Программирование на С++ в Visual Studio® 2010 Express

Глава 1. Первые шаги

© Прохоренок Н.А., 2010 г., unicross@mail.ru

Этот DjVu-файл предоставляется КАК ЕСТЬ. Автор не несет никакой ответственности за прямые или косвенные проблемы, связанные с использованием данного файла. ВЫ ИСПОЛЬЗУЕТЕ ЕГО НА СВОЙ СТРАХ И РИСК.

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Запрещено:
- перепечатывать всю книгу или отдельные главы без письменного разрешения Автора;
- декомпилировать данный файл и преобразовывать его в любой другой формат.
Создание пустого проекта

В процессе изучения основ языка C++ мы будем создавать консольные приложения. Консольное приложение — это программа, отображающая текстовую информацию и позволяющая вводить символы с клавиатуры. Консольное приложение позволит не отвлекаться на изучение среды разработки, а полностью сосредоточить свое внимание на изучении синтаксиса языка.

Создать консольное приложение можно двумя способами. Первый способ заключается в создании пустого проекта и написании кода с нуля. Именно этим способом мы будем пользоваться на протяжении всей книги. Второй способ позволяет создать консольное приложение с помощью мастера. В этом случае мастер создаст все необходимое файлы и заполнит их шаблонами кода.

Чтобы создать пустой проект в меню Файл выбираем пункт Создать | Проект. В открывшемся окне выделяем пункт Консольное приложение Win32. Вводим название проекта (например, "helloworld") в поле Имя. Введенное название автоматически копируется в поле Имя решения. В поле Расположение указываем путь к каталогу, в котором будет сохранен проект, и устанавливаем флажок Создать каталог для решения. Нажимаем кнопку ОК. В результате открывается окно Мастер приложений Win32. Нажимаем кнопку Далее. В группе переключателей Тип приложения выбираем Консольное приложение. Снимаем флажок Предварительно скомпилированный заголовок и устанавливаем флажок Пустой проект. Нажимаем кнопку Готово. В результате проект будет создан и открыт в главном окне.

Создать пустой проект можно гораздо быстрее. Для этого в окне Создать проект вместо пункта Консольное приложение Win32 следует выделить пункт Пустой проект. В этом случае проект будет создан сразу после нажатия кнопки ОК.

После создания пустого проекта необходимо добавить основной файл. Для этого в окне Обозреватель решений щелкаем правой кнопкой мыши на пункте Файлы исходного кода и из контекстного меню выбираем пункт Добавить | Создать элемент. В результате открывается окно Добавление нового элемента. Выделяем пункт Файл C++ (.cpp), вводим название файла (например, "main") и нажимаем кнопку Добавить. В результате файл будет добавлен в папку проекта и его название отобразится в окне Обозреватель решений, а сам файл будет открыт на отдельной вкладке.

При изучении языков программирования принято начинать с программы, выводящей надпись "Hello, world!" в окно консоли. Не будем нарушать традицию и продемонстрируем, как это выглядит на C++ (лисинг 1.1).

Листинги на странице http://unicross.narod.ru/cpp/
Листинг 1.1. Первая программа

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

В первой строке программы с помощью директивы `include` подключается файл `iostream`, в котором объявлен объект `cout`, предназначенный для вывода данных в окно консоли. Далее создается функция `main()`, внутри которой расположены все остальные инструкции, ограниченные фигурными скобками. Именно функция с названием `main()` будет автоматически вызываться при запуске программы. Перед названием функции указывается тип возвращаемого значения. Ключевое слово `int` означает, что функция возвращает целое число. Возвращаемое значение указывается после ключевого слова `return` в самом конце функции `main()`. Число 0 в данном случае означает нормальное завершение программы. Если указано другое число, то это свидетельствует о некорректном завершении программы. Согласно стандарту, внутри функции `main()` ключевое слово `return` можно не указывать. В этом случае компилятор должен самостоятельно вставить инструкцию, возвращающую значение 0.

Вывод строки "Hello, world!" осуществляется с помощью объекта `cout`. Чтобы исключить конфликт имен все стандартные идентификаторы в языке C++ определены в пространстве имен `std`. Поэтому перед именем объекта `cout` необходимо указать название пространства имен. Между названием пространства имен и названием объекта указываются два двоеточия (`std::cout`). И, наконец, константа `endl` (сокращение от "end line") вставляет символ перевода строки.

Теперь попробуем скомпилировать и запустить программу. Для этого добавляем содержимое листинга в созданный файл, а затем из меню Построение выбираем пункт Построить решение или нажимаем клавишу <F7>. В результате в окне Вывод отображается следующая информация:

```
1>------- Построение начато: проект: helloworld, Конфигурация: Debug Win32 -------
1> main.cpp
1> helloworld.vcxproj -> C:\book\helloworld\Debug\helloworld.exe
======== Построение: успешно: 1, с ошибками: 0, без изменений: 0, пропущено: 0 ========
```

Листинги на странице http://unicross.narod.ru/cpp/
Данный результат свидетельствует об успешном создании файла `helloworld.exe` в конфигурации `Debug` (отладка) в папке `C:\book\helloworld\Debug\`. Если при наборе были допущены ошибки, то их описание будет отображено в окне `Вывод`. В качестве примера уберем точку с запятой после константы `endl` и попробуем произвести компиляцию. Результат будет выглядеть так:

1>------- Построение начато: проект: helloworld, Конфигурация: Debug
Win32 -------
1> main.cpp
1>c:\book\helloworld\helloworld\main.cpp(5): error C2143: синтаксическая ошибка: отсутствие ";" перед "return"
======= Построение: успешно: 0, с ошибками: 1, без изменений: 0, пропущено: 0 ======

Фраза "успешно: 0, с ошибками: 1" говорит о наличии ошибки в программе. Описание самой ошибки приводится строкой ранее. Указывается название файла, номер ошибки ("C2143"), а также подробное описание ошибки, в данном случае на русском языке. После исправления ошибки необходимо заново скомпилировать проект.

Чтобы посмотреть результат выполнения программы из меню `Отладка` выбираем пункт `Запуск без отладки` или нажимаем комбинацию клавиш `<Ctrl>`+<F5>. В результате откроется окно консоли с приветствием, а также со строкой "Для продолжения нажмите любую клавишу". Нажатие любой клавиши приведет к закрытию окна консоли. Запустить программу можно также в режиме `Отладка`. Для этого из меню `Отладка` выбираем пункт `Начать отладку` или нажимаем клавишу <F5>. В этом случае окно консоли откроется, затем сразу закроется, а в окне `Вывод` отобразится отладочная информация. Как сделать, чтобы окно консоли сразу не закрывалось, мы рассмотрим далее в этой главе.

Следует заметить, что при нажатии клавиши <F5> или комбинации клавиш `<Ctrl>`+<F5> производится проверка актуальности скомпилированного файла. Если файл устарел (в программе сделаны изменения), то будет выведено окно, в котором предлагается произвести повторную компиляцию. Чтобы выполнить повторную компиляцию нажимаем кнопку `Да`. Таким образом, можно не производить предварительную компиляцию, а сразу запускать программу.

**Примечание**
Если в меню `Отладка` нет пункта `Запуск без отладки`, то предварительно в меню `Сервис` необходимо выбрать пункт `Параметры | Расширенные параметры`.

Создание консольного приложения

Чтобы создать консольное приложение с помощью мастера в меню Файл выбираем пункт Создать | Проект. В открывшемся окне выделяем пункт Консольное приложение Win32. Вводим название проекта (например, "welcome") в поле Имя. Введенное название автоматически копируется в поле Имя решения. В поле Расположение указываем путь к каталогу, в котором будет сохранен проект, и устанавливаем флажок Создать каталог для решения. Нажимаем кнопку ОК. В результате откроется окно Мастер приложений Win32. Нажимаем кнопку Далее. В группе переключателей Тип приложения выбираем Консольное приложение. Устанавливаем флажок Предварительно скомпилированный заголовок. Флажок Пустой проект должен быть снят. Нажимаем кнопку Готово.

В результате будут созданы следующие файлы:

→ welcome.cpp — основной файл проекта;

→ stdafx.h — файл для подключения заголовочных файлов, используемых в проекте. Этот файл необходимо подключать во всех файлах проекта;

→ stdafx.cpp — файлы stdafx.h и stdafx.cpp используются для построения файла предкомпилированного заголовка (welcome.pch) и файла предкомпилированных типов (stdafx.obj);

→ targetver.h — в этом файле подключается файл SDKDDKVer.h, который обеспечивает определение последней доступной платформы Windows.

Файлы stdafx.cpp и targetver.h оставляем без изменений. В конец файла stdafx.h вставляем строку, позволяющую использовать в программе объект cout, предназначенный для вывода данных в окно консоли:

```cpp
#include <iostream>
```

Содержимое файла stdafx.h приведено в листинге 1.2.

<table>
<thead>
<tr>
<th>Листинг 1.2. Содержимое файла stdafx.h</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma once</td>
</tr>
<tr>
<td>#include &quot;targetver.h&quot;</td>
</tr>
<tr>
<td>#include &lt;stdio.h&gt;</td>
</tr>
<tr>
<td>#include &lt;tchar.h&gt;</td>
</tr>
<tr>
<td>#include &lt;iostream&gt;</td>
</tr>
</tbody>
</table>

Листинги на странице http://unicross.narod.ru/cpp/
Директива `pragma со значением once` указывает, что содержимое файла может быть включено только один раз. Директивы `include` подключают файлы, в которых содержатся объявления идентификаторов. Такие файлы называются заголовочными файлами. Они содержат только объявление идентификаторов без описания их реализации. Обратите внимание на то, что в директиве `include` заголовочные файлы подключаются различными способами. Поиск файлов, указанных внутри угловых скобок, производится в системных каталогах, а поиск файлов, указанных внутрь кавычек, производится в каталоге проекта. Таким образом, названия файлов стандартной библиотеки необходимо указывать внутри угловых скобок, а пользовательских файлов внутрь кавычек.

Внимательный читатель наверняка обратил внимание на то, что три первых заголовочных файла содержат расширение .h, а в последней директиве расширение файла не указано. Наличие расширения файла принято в стандартной библиотеке языка C. В стандартной библиотеке языка C++ расширение файла принято не указывать. Так как язык C++ наследует все библиотеки языка C, то файлы можно подключать как в стиле языка C, так и в стиле языка C++. Например, файл `string.h` из стандартной библиотеки языка C доступен в языке C++ под названием `cstring`, а файл `math.h` под названием `cmath`. Отличие между этими способами подключения заключается в импортировании идентификаторов. В языке C при подключении файла (например, `math.h`) все идентификаторы импортируются в глобальное пространство имен, а в языке C++ при подключении файла (например, `cmath`) идентификаторы определяются в пространстве имен под названием `std`. Поэтому перед идентификатором необходимо указать название пространства имен (например, `std::cout`). Использование пространств имен позволяет избежать конфликта имен в программе.

Чтобы посмотреть содержимое заголовочного файла (например, `stdio.h`) щелкните правой кнопкой мыши на строке "#include `<stdio.h>`" и из контекстного меню выберите пункт Открыть документ «stdio.h». В результате заголовочный файл `stdio.h` будет открыт на отдельной вкладке. Следует заметить, что файл `stdio.h` содержит объявления идентификаторов, предназначенных для операций ввода/вывода в языке C. Так как для вывода мы используем объект `cout`, объявленный в файле `iostream`, то подключение файла `stdio.h` можно вообще удалить из файла `stdafx.h`. Производить операции ввода/вывода мы будем только в стиле языка C++.

Теперь рассмотрим содержимое файла `welcome.cpp` (листинг 1.3).
#include "stdafx.h"
using namespace std;

int _tmain(int argc, _TCHAR* argv[]) {
    cout << "Welcome to C++!" << endl;
    return 0;
}

Сравните этот листинг с предыдущим примером (листинг 1.1). Первое, что должно броситься в глаза — это название функции (вместо названия main() используется название _tmain()) и наличие двух параметров. Параметры расположены внутри круглых скобок и разделены запятой. Параметр состоит из объявлений типа данных и названия. Первый параметр (argc) имеет тип int, который соответствует целочисленному значению. Через этот параметр доступно количество аргументов, переданных в командной строке. Следует учитывать, что первым аргументом является название исполняемого файла. Поэтому значение параметра argc не может быть меньше единицы. Через второй параметр (argv) доступны все аргументы в виде строки (тип _TCHAR*). Квадратные скобки после названия второго параметра означают, что доступен массив строк. Более подробно передачу аргументов через командную строку мы рассмотрим далее в этой главе.

Для вывода строки "Welcome to C++!", как и в предыдущем примере, используется объект cout, объявленный в файле iostream. Файл iostream мы подключили в файле stdafx.h, поэтому повторно подключать его не нужно. Достаточно подключить только файл stdafx.h в начале программы.

Обратите внимание на то, что перед объектом cout и константой endl не указано название пространства имен std, так как все идентификаторы, объявленные в пространстве имен std были импортированы в основную программу с помощью инструкции:

    using namespace std;

Хотя это очень удобно, следует избегать импортирования всех идентификаторов в глобальное пространство имен, так как возможны конфликты имен. Вместо этого способа лучше импортировать определенные идентификаторы. Например, импортировать только идентификаторы cout и endl можно так:

    using std::cout;

Листинги на странице http://unicross.narod.ru/cpp/
using std::endl;
Если необходимо импортировать все идентификаторы, то следует ограничить область видимости. Например, указать инструкцию using внутри функции. В этом случае область видимости ограничена рамками функции и в глобальное пространство имен идентификаторы не попадут.
Скомпилировать и запустить программу на исполнение можно также как и в предыдущем разделе. Для запуска без отладки нажимаем комбинацию клавиш <Ctrl>+<F5>. Если нужно просто скомпилировать проект, то нажимаем клавишу <F7>. Запуск с отладкой осуществляется нажатием клавиши <F5>. Запомните эти комбинации клавиш наизусть.

Структура программы
Запускать программу мы научились, теперь можно приступать к изучению языка C++. Как видно из предыдущих примеров, программа состоит из инструкций, расположенных в текстовом файле. Структура обычной программы выглядит так:

<Подключение заголовочных файлов>
<Определение глобальных переменных>
<Определение функций, классов и др.>
int main([ int argc, char *argv[] ]) {
    <Инструкции>
    return 0;
}

<Определение функций и классов>
В самом начале программы подключаются заголовочные файлы, в которых содержатся объявления идентификаторов без их реализации. Например, чтобы иметь возможность вывести данные в окно консоли необходимо подключить файл iostream, в котором содержится объявление объекта cout. Подключение осуществляется с помощью директивы include:
#include <iostream>
С помощью директивы include можно подключать как стандартные заголовочные файлы, так и пользовательские файлы.
После подключения файлов производится объявление глобальных переменных. Переменные предназначены для хранения значений определенного типа. Глобальные переменные видны во всей программе, включая функции. Если объявить переменную внутри функции, то область видимости переменной будет ограничена рамками функции.

Листинги на странице http://unicross.narod.ru/cpp/
и в других частях программы использовать переменную нельзя. Такие переменные называются локальными. Объявить целочисленную переменную можно так:

```c
int x;
```

В этом примере мы объявили переменную с названием "x" и указали, что переменная может хранить данные, имеющие тип `int`. Ключевое слово `int` означает, что переменная предназначена для хранения целого числа. Попытка присвоить переменной значение, имеющее другой тип приведет к предупреждению или ошибке. Исключением являются ситуации, в которых компилятор может произвести преобразование типов данных без потери точности. Например, целое число без проблем преобразуется в вещественное число, однако попытка преобразовать вещественное число в целое приведет к выводу предупреждающего сообщения, даже если после точки стоит ноль.

Пример сообщения:

```c
warning C4244: =: преобразование "double" в "int", возможна потеря данных
```

Хотя компилятор предупреждает о потере данных, программа все равно будет скомпилирована. Поэтому обращайте внимание на сообщения, которые выводятся в окне **Вывод** на вкладке **Построение** при компиляции. Иначе программа будет работать, но результат выполнения будет некорректным.

Обратите внимание на то, что объявление переменной заканчивается точкой с запятой. Большинство инструкций в языке C++ должно заканчиваться точкой с запятой. Если точку с запятой не указать, то компилятор сообщает о синтаксической ошибке. Пример сообщения:

```c
error C2144: синтаксическая ошибка: перед "int" требуется ";"
```

Чтобы увидеть строку, о которой сообщает компилятор, сделайте двойной щелчок мышью на сообщении в окне **Вывод**. В результате строка с ошибкой в программе станет активной и слева от нее отобразится маркер. В нашем случае активной будет строка, расположенная сразу после объявления глобальной переменной.

При объявлении можно сразу присвоить начальное значение переменной. Присваивание значения переменной при объявлении называется **инициализацией переменной**. Чтобы произвести инициализацию переменной после ее названия указывается оператор `=`, а затем значение. Пример:

```c
int x = 10;
```

Если глобальной переменной не присвоено значение при объявлении, то она будет иметь значение 0. Если локальной переменной не присвоено значение, то переменная будет содержать произвольное значение. Как говорят в таком случае переменная содержит "мусор".

Листинги на странице http://unicross.narod.ru/cpp/
Обратите внимание на то, что перед оператором = и после него вставлены пробелы. Количество пробелов может быть произвольным или пробелов может не быть вовсе. Кроме того, допускается вместо пробелов использовать символ перевода строки или символ табуляции. Например, эти инструкции вполне допустимы:

```
int x=21;
int y = 85;
int z = 56;
```

Тем не менее, следует придерживаться единообразия в коде и обрамлять операторы одним пробелом. Если присваивание производится в объявлении функции (задается значение параметра по умолчанию), то пробелы принято не указывать вовсе. Следует учитывать, что это не строгие правила, а лишь рекомендации по оформлению кода.

Как видно из предыдущего примера, инструкция может быть расположена на нескольких строках. Концом инструкции является точка с запятой, а не конец строки. Например, на одной строке может быть несколько инструкций:

```
int x = 21; int y = 85; int z = 56;
```

Из этого правила есть исключения. Например, после директив препроцессора точка с запятой не указывается. В этом случае концом инструкции является конец строки. Директиву препроцессора можно узнать по символу # перед названием директивы. Типичным примером директивы препроцессора является директива include, которая используется для подключения заголовочных файлов:

```
#include <iostream>
```

После объявления глобальных переменных могут располагаться объявления функций и классов. Такие объявления называются прототипами. Схема прототипа функции выглядит следующим образом:

```
<Тип возвращаемого значения> <Название функции>(([[<Тип> [<Параметр1>]
                  [, ..., <Тип> [<ПараметрN>]]])))
```

Например, прототип функции, которая складывает два целых числа и возвращает их сумму выглядит так:

```
int summa(int x, int y);
```

После объявления функции необходимо описать ее реализацию, которая называется определением функции. Определение функции обычно располагается после определения функции main(). Обратите внимание на то, что объявлять прототип функции main()
не нужно, так как определение этой функции обычно расположено перед определением других функций. Пример определения функции `summa()`:

```cpp
int summa(int x, int y) {
    return x + y;
}
```

Как видно из примера, первая строка в определении функции `summa()` совпадает с объявлением функции. Следует заметить, что в объявлении функции можно не указывать названия параметров. Достаточно указать информацию о типе данных. Таким образом, описание функции можно записать так:

```cpp
int summa(int, int);
```

После объявления функции ставится точка с запятой. В определении функции внутри фигурных скобок должна быть описана реализация функции. Так как в объявлении указано, что функция возвращает целочисленное значение, следовательно после описания реализации необходимо вернуть значение. Возвращаемое значение указывается после ключевого слова `return`. Если функция не возвращает никакого значения, то перед названием функции вместо типа данных указывается ключевое слово `void`. Пример объявления функции, которая не возвращает значения:

```cpp
void print(int);
```

Пример определения функции `print()`:

```cpp
void print(int x) {
    std::cout << x << std::endl;
}
```

Вся реализация функции должна быть расположена внутри фигурных скобок. Открывающая скобка может находиться на одной строке с определением функции, как в предыдущем примере, или в начале следующей строки. Пример:

```cpp
void print(int x)
{
    std::cout << x << std::endl;
}
```

Многие программисты считают такой стиль наиболее приемлемым, так как открывающая и закрывающая скобки расположены друг под другом. На мой же взгляд образуется лишняя пустая строка. Так как размеры экрана ограничены, при наличии пустой строки на экран помещается меньше кода и приходится чаще пользоваться полосой прокрутки. Если размещать инструкции с равным отступом, то блок кода выделяется визуально и следить за положением фигурных скобок просто излишне. Тем

Листинги на странице http://unicross.narod.ru/cpp/
более, что редактор кода позволяет подсветить папные скобки. Чтобы найти пару фигурных скобок следует поместить указатель ввода перед скобкой. В результате скобки будут подсвечены. Какой стиль использовать, зависит от личного предпочтения программиста или от правил оформления кода, принятых в определенной фирме. Главное, чтобы стиль оформления внутри одной программы был одинаковым.

Перед инструкциями внутри фигурных скобок следует размещать одинаковый отступ. В качестве отступа можно использовать пробелы или символ табуляции. При использовании пробелов размер отступа равняется трем или четырем пробелам для блока первого уровня. Для вложенных блоков количество пробелов умножают на уровень вложенности. Если для блока первого уровня вложенности использовалось три пробела, то для блока второго уровня вложенности должно использоваться шесть пробелов, для третьего уровня — девять пробелов и т. д. В одной программе не следует использовать пробелы и пробелы в табуляцию в качестве отступа. Необходимо выбрать что-то одно и пользоваться этим во всей программе.

Закрывающая фигурная скобка обычно размещается в конце блока на отдельной строке. После скобки точка с запятой не указывается. Однако наличие точки с запятой ошибкой не является, так как инструкция может не содержать выражений вообще. Например, такая инструкция вполне допустима, хотя она ничего не делает:

;

Самой главной функцией в программе является функция main(). Именно функция с названием main() будет автоматически вызываться при запуске программы. Функция имеет два прототипа:

```c
int main();
int main(int argc, char *argv[]);
```

Первый прототип функции имеет синоним:

```c
int main(void);
```

Значение void означает, что функция не принимает параметры. Подобный синтаксис используется в языке программирования C. В языке C++ указание ключевого слова void является излишним. Достаточно указать пустые скобки.

Второй прототип применяется для получения значений, указанных при запуске программы из командной строки. Количество значений доступно через параметр argc, а сами значения через параметр argv.

Перед названием функции указывается тип возвращаемого значения. Ключевое слово int означает, что функция возвращает целое число. Возвращаемое значение указывается после ключевого слова return в самом конце функции main(). Число 0

Листинг на странице http://unicross.narod.ru/cpp/
означает нормальное завершение программы. Если указано другое число, то это свидетельствует о некорректном завершении программы. Согласно стандарту, внутри функции main() ключевое слово return можно не указывать. В этом случае компилятор должен самостоятельно вставить инструкцию, возвращающую значение 0. Возвращаемое значение передается операционной системе и может использоваться для определения корректности завершения программы.

В листинге 1.4 приведен пример программы, структуру которой мы рассмотрели в этом разделе. Не расстраивайтесь, если что-то показалось непонятым. Все это мы будем изучать более подробно в следующих главах книги. На этом этапе достаточно лишь представлять структуру программы и самое главное уметь ее скомпилировать и запустить на выполнение. Напомню, что для выполнения программы следует создать пустой проект, а не консольное приложение.

**Листинг 1.4. Пример программы**

```cpp
#include <iostream>

int x = 21;
int y = 85;

int summa(int x, int y);
void print(int x);

int main() {
    int z;
    z = summa(x, y);
    print(z);
    return 0;
}

int summa(int x, int y) {
    return x + y;
}

void print(int x) {
    std::cout << x << std::endl;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
Комментарии в программе

Комментарии предназначены для вставки пояснений в текст программы и компилятор полностью их игнорирует. Внутри комментария может располагаться любой текст, включая инструкции, которые выполнять не следует. Помните, комментарии нужны программисту, а не компилятору. Вставка комментариев в код позволит через некоторое время быстро вспомнить предназначение фрагмента кода.

В языке C++ присутствуют два типа комментариев: одностroчный и многострочный.

Одностroчный комментарий начинается с символов // и заканчивается в конце строки. Вставлять одностroчный комментарий можно как в начале строки, так и после инструкции. Если символ комментария разместить перед инструкцией, то она не будет выполнена. Если символы // расположены внутри кавычек, то они не являются признаком начала комментария. Примеры одностroчных комментариев:

```
// Это комментарий
std::cout << "Hello, world!"; // Это комментарий
// std::cout << "Hello, world!"; // Инструкция выполнена не будет
char s[] = "// Это НЕ комментарий!!!";
```

Многострочный комментарий начинается с символов /* и заканчивается символами */. Комментарий может быть расположен как на одной строке, так и на нескольких. Кроме того, многострочный комментарий можно размещать внутри выражения, хотя это и нежелательно. Следует иметь в виду, что многострочные комментарии не могут быть вложенными, поэтому при комментировании больших блоков следует проверять, что в них не встречается закрывающая комментарий комбинация символов */. Тем не менее, одностroчный комментарий может быть расположен внутри многострочного комментария. Примеры многострочных комментариев:

```
/*
Многострочный комментарий
*/
std::cout << "Hello, world!"; /* Это комментарий */
/* std::cout << "Hello, world!"; // Инструкция выполнена не будет */
int x;
x = 10 /* Комментарий */ + 50 /* внутри выражения */;
char s[] = "/* Это НЕ комментарий!!! */";
```

Листинги на странице http://unicross.narod.ru/cpp/
Редактор в VC++ позволяет быстро закомментировать фрагмент кода. Для этого необходимо выделить одну инструкцию (или сразу несколько инструкций), а затем из меню Правка выбрать пункт Дополнительно | Преобразовать выделенный фрагмент в комментарий. Можно также нажать кнопку Закомментировать выделенные строки на панели инструментов Текстовый редактор. Если панель инструментов не отображена, в меню Вид выбираем пункт Панели инструментов | Текстовый редактор. Кроме того, можно воспользоваться клавишами быстрого доступа. Для этого следует нажать комбинацию клавиш <Ctrl>+<K>, а затем комбинацию клавиш <Ctrl>+<C>. Если строка выделена полностью, то перед ней будет вставлен односторочный комментарий. Если выделен лишь фрагмент строки, то он будет обернут в многострочный комментарий.

Чтобы убрать комментарий следует выделить фрагмент, а затем из меню Правка выбрать пункт Дополнительно | Отменить преобразование в комментарий. Можно также нажать кнопку Отменить преобразование выделенных строк в комментарий на панели инструментов Текстовый редактор. Кроме того, можно воспользоваться клавишами быстрого доступа. Для этого следует нажать комбинацию клавиш <Ctrl>+<K>, а затем комбинацию клавиш <Ctrl>+<U>.

**Вывод данных в языке C++**

Для вывода данных в языке C++ предназначены объекты cout, cerr и clog, объявленные в файле iostream. Объект cout используется для вывода обычных сообщений в окно консоли. Объекты cerr и clog применяются для вывода сообщений об ошибках. Также как и объект cout объекты cerr и clog первоначально связаны с окном консоли, однако возможно перенаправить поток на другое устройство или в файл. Для понимания всех возможностей этих объектов требуется знание принципов объектно-ориентированного программирования (ООП). Так как ООП мы еще не изучали, в этом разделе будут рассматриваться только основные возможности объектов. В последующих главах мы вернемся к изучению этих объектов.

Прежде чем использовать объекты необходимо подключить файл iostream с помощью директивы include:

```c++
#include <iostream>
```

Обратите внимание на то, что название файла указывается внутри угловых скобок без расширения h, так как файл iostream входит в состав стандартной библиотеки C++.

После подключения файла все объекты будут доступны через пространство имен std, поэтому при обращении к объекту необходимо указать название пространства имен и

Листинги на странице http://unicross.narod.ru/cpp/
два символа двоеточия перед названием объекта. Например, вывести строку "Hello, world!" можно так:

```cpp
std::cout << "Hello, world!";
```

Если каждый раз указывать название пространства имен лень, то можно импортировать все идентификаторы в глобальное пространство имен с помощью инструкции:

```cpp
using namespace std;
```

В этом случае название пространства имен указывать не нужно:

```cpp
cout << "Hello, world!";
```

Хотя это очень удобно, следует избегать импортирования всех идентификаторов в глобальное пространство имен, так как возможны конфликты имен. Вместо этого способа лучше импортировать определенные идентификаторы. Например, импортировать только объект `cout` можно так:

```cpp
using std::cout;
```

После названия объекта указывается оператор `<<`, который как бы указывает направление вывода. При разработке собственных объектов можно перегрузить этот оператор и тем самым управлять выводом. После оператора `<<` передается объект, который будет выведен в окне консоли. В качестве объекта можно указать число, строку, а также объект, в котором оператор `<<` перегружен. Пример:

```cpp
std::cout << 10;       // Целое число
std::cout << 81.5;     // вещественное число
std::cout << "Hello, world!";  // строка
char str[] = "Hello, world!";
std::cout << str;      // строка
```

Результат выполнения этого примера будет таким:

```
1081.5Hello, world!Hello, world!
```

Как видно из результата, все данные выводятся на одной строке. Чтобы данные выводились на отдельных строках можно воспользоваться константой `endl` (окончание от "end line"). Пример:

```cpp
std::cout << "String1";
std::cout << std::endl;
std::cout << "String2";
```

Результат в окне консоли:

```
String1
String2
```

Листинги на странице http://unicross.narod.ru/cpp/
На самом деле endl не совсем константа. Можно сказать, что это функция (называемая манипулятором), внутри которой производится вывод символа перевода строки и возвращается ссылка на поток вывода. Благодаря возврату ссылки можно строить цепочки из операторов <<. Например, предыдущий пример можно записать так:

```cpp
std::cout << "String1" << std::endl << "String2";
```

Для перевода строки можно также воспользоваться комбинацией символов "\n", которые обозначают символ перевода строки. Пример:

```cpp
std::cout << "String1" << "\n" << "String2";
```

Для вывода строк, состоящих из двухбайтных символов, предназначены объекты wcout, wcerr и wclog, объявленные в файле iostream. Объект wcout предназначен для вывода обычных сообщений, а объективы wcerr и wclog для вывода сообщений об ошибках. Пример:

```cpp
std::wcout << L"String";
```

С помощью стандартного вывода можно создать индикатор выполнения процесса в окне консоли. Чтобы реализовать такой индикатор нужно вспомнить, что символ перевода строки в Windows состоит из двух символов \r (перевод каретки) и \n (перевод строки). Таким образом, используя только символ перевода каретки \r можно перемещаться в начало строки и перезаписывать ранее выведенную информацию. Пример индикатора процесса показан в листинге 1.5.

## Листинг 1.5. Индикатор выполнения процесса

```cpp
#include <iostream>

int main() {
    int i, j;
    std::cout << "... 08";
    for (i=5; i<101; i+=5) {
        for (j=0; j<500000000; ++j); // Имитация процесса
            std::cout << "\r... " << i << "8";
    }
    std::cout << std::endl;
    return 0;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
Так как данные перед выводом могут помещаться в буфер, вполне возможно потребуется сбросить буфер явным образом. Сделать это можно с помощью метода flush():
std::cout.flush();

**Вывод данных в языке C**

Язык C++ поддерживает также операции вывода из языка C, объявленные в файле stdio.h. В языке C++ вместо этого файла можно подключать файл cstdio:

```cpp
#include <cstdio>
```

Для вывода одиночного символа применяется функция putchar(). Прототип функции:

```cpp
int putchar(int ch);
```

Пример вывода символа:

```cpp
std::putchar('w'); // w
std::putchar(119); // w
```

Вывести строку позволяет функция puts(). Прототип функции:

```cpp
int puts(const char *Str);
```

Функция выводит строку Str и вставляет символ перевода строки. Пример:

```cpp
std::puts("String1");
std::puts("String2");
```

Результат выполнения:

String1
String2

Для форматированного вывода используется функция printf(). Прототип функции:

```cpp
int printf(const char *Format, ...);
```

В параметре Format указывается строка специального формата. Внутри этой строки можно указать обычные символы и спецификаторы формата, начинающиеся с символа %. Вместо спецификаторов формата подставляются значения, указанные в качестве параметров. Количество спецификаторов должно совпадать с количеством переданных параметров. В качестве значения функция возвращает количество выведенных символов. Пример вывода строки и числа:

```cpp
std::printf("String\n");
std::printf("Count %d\n", 10);
std::printf("%s %d\n", "Count", 10);
```

Листинги на странице http://unicross.narod.ru/cpp/
Результат выполнения:

`String`
`Count 10`
`Count 10`

В первом примере строка формата не содержит спецификаторов и выводится как есть. Во втором примере внутри строки формата используется спецификатор `%d`, предназначенный для вывода целого числа. Вместо этого спецификатора подставляется число 10, переданное во втором параметре. В третьем примере строка содержит сразу два спецификатора `%s` и `%d`. Спецификатор `%s`, предназначен для вывода строки, а спецификатор `%d` — для вывода целого числа. Вместо спецификатора `%s` будет подставлена строка "Count", а вместо спецификатора `%d` — число 10. Обратите внимание на то, что тип данных переданных значений должен совпадать с типом спецификатора. Если в качестве значения для спецификатора `%s` указать число, то это приведет в ошибке времени исполнения. Никакой проверки соответствия типа на этапе компиляции не производится.

Спецификаторы имеют следующий синтаксис:

`%[<Модификаторы>][<Ширина>][.<<Точность>>]<тип>`

В параметре `<тип>` могут быть указаны следующие символы:

- `c` — символ:
  ```
  std::printf("%c", 'w'); // w
  std::printf("%c", 119); // w
  ```

- `s` — строка:
  ```
  std::printf("%s", "String"); // String
  ```

- `d` или `i` — десятичное целое число со знаком:
  ```
  std::printf("%d %i", 10, 30); // 10 30
  std::printf("%d %i", -10, -30); // -10 -30
  ```

- `u` — десятичное целое число без знака:
  ```
  std::printf("%u", 10); // 10
  ```

- `o` — восьмеричное число без знака:
  ```
  std::printf("%o %o", 10, 077); // 12 77
  std::printf("%#o %#o", 10, 077); // 012 077
  ```

- `x` — шестнадцатеричное число без знака в нижнем регистре:

std::printf("%x %x", 10, 0xff);  // a ff
std::printf("%#x %#x", 10, 0xff); // 0xa 0xff

→ x — шестнадцатеричное число без знака в верхнем регистре:
  std::printf("%x %x", 10, 0xff);  // A FF
  std::printf("%#x %#x", 10, 0xff); // 0XA 0xFF

→ f — вещественное число в десятичном представлении:
  std::printf("%f %f", 18.65781452, 12.5); // 18.657815 12.500000
  std::printf("%f", -18.65781452); // -18.657815
  std::printf("%#.0f %#.0f", 100.0, 100.0); // 100. 100

→ e — вещественное число в экспоненциальной форме (буква "e" в нижнем регистре):
  std::printf("%e", 18657.81452); // 1.865781e+004
  std::printf("%e", 0.000081452); // 8.145200e-005

→ E — вещественное число в экспоненциальной форме (буква "E" в верхнем регистре):
  std::printf("%E", 18657.81452); // 1.865781E+004

→ g — эквивалентно f или e (выбирается более короткая запись числа):
  std::printf("%g %g %g", 0.086578, 0.000086578, 1.865E-005);
  // 0.086578 8.6578e-005 1.865e-005

→ G — эквивалентно f или E (выбирается более короткая запись числа):
  std::printf("%G %G %G", 0.086578, 0.000086578, 1.865E-005);
  // 0.086578 8.6578E-005 1.865E-005

→ p — вывод адреса переменной:
    int x = 10;
    std::printf("%p", &x); // 0012FF60
    std::printf("%#p", &x); // 0x0012FF60

→ % — символ процента (%):
    std::printf("10%%"); // 10%

Параметр <Ширина> задает минимальную ширину поля. Если строка меньше ширины поля, то она дополняется пробелами. Если строка не помещается в указанную ширину, то значение игнорируется и строка выводится полностью:

Листинги на странице http://unicross.narod.ru/cpp/
Задать минимальную ширину можно не только для строк, но и для других типов:
std::printf("%3s", "string"); // 'string'
std::printf("%10s", "string"); // ' string'
Параметр <Точность> задает количество знаков после точки для вещественных чисел.
Перед этим параметром обязательно должна стоять точка. Пример:
std::printf("%10.5f", 3.14159265359); // 3.14159
std::printf("%.3f", 3.14159265359); // 3.142
Если параметр <Точность> используется применительно к целому числу, то он задает
минимальное количество цифр. Если число содержит меньшее количество цифр, то
вначале числа добавляются нули. Пример:
std::printf("%7d", 100);       // ' 100'
std::printf("%.7d", 100);      // '0000100'
std::printf("%.7d", 123456789); // '123456789'
Если параметр <Точность> используется применительно к строке, то он задает
максимальное количество символов. Символы, которые не помещаются будут
отброшены. Пример:
std::printf("%5.7s", "Hello, world!"); // 'Hello,'
std::printf("%15.20s", "Hello, world!"); // ' Hello, world!
Вместо минимальной ширины и точности можно указать символ * . В этом случае
значения передаются через параметры функции printf() в порядке указания символов
в строке формата. Примеры:
std::printf("%*.5f", 10, 5, 3.14159265359); // 3.14159
std::printf("%*.f", 3, 3.14159265359);       // 3.142
std::printf("%*s", 10, "string");           // ' string'
В первом примере вместо первого символа * подставляется число 10, указанное во
втором параметре, а вместо второго символа * подставляется число 5, указанное в
третьем параметре. Во втором примере вместо символа * подставляется число 3,
которое задает количество цифр после точки. В третьем примере символ * заменяется
числом 10, которое задает минимальную ширину поля.
В параметре <Модификаторы> могут быть указаны следующие символы:

# — для восьмеричных значений добавляется в начало символ "0", для
шестнадцатеричных значений добавляет комбинацию символов "0x" (если

Листинги на странице http://unicross.narod.ru/cpp/
используется тип "x" или "0x" (если используется тип "X"), для вещественных чисел указывает всегда выводить дробную точку, даже если задано значение 0 в параметре <Точность>:

```cpp
std::printf("%#o %#o", 10, 077); // 012 077
std::printf("%#x %#x", 10, 0xff); // 0xa 0xff
std::printf("%#X %#X", 10, 0xff); // 0XA 0xFF
std::printf("%#.0f %.0f", 100.0, 100.0); // 100. 100
```

0 — задает наличие ведущих нулей для числового значения:

```cpp
std::printf("'%7d'", 100); // '   100'
std::printf("'%07d'", 100); // '0000100'
```

— задает выравнивание по левой границе области. По умолчанию используется выравнивание по правой границе. Пример:

```cpp
std::printf("'%5d' '5d'", 3, 3); // '   3' '3'
std::printf("'%05d' '5d'", 3, 3); // '00003' '3'
```

 пробел — вставляет пробел перед положительным числом. Перед отрицательным числом будет стоять минус. Пример:

```cpp
std::printf("'% d' ' d'", -3, 3); // '-3' ' 3'
```

+ — задает обязательный вывод знака, как для отрицательных, так и для положительных чисел. Пример:

```cpp
std::printf("'%+d' '+d'", -3, 3); // '-3' '+3'
```

h — предназначен для вывода значения переменной, имеющей тип short int. Пример:

```cpp
short int x = 32767;
std::printf("%hd", x); // 32767
```

l (буква "эль") — предназначен для вывода значения переменной, имеющей тип long int. Пример:

```cpp
long int x = 2147483647;
std::printf("%ld", x); // 2147483647
```

Модификатор l можно использовать совместно с типами c и s, для вывода двухбайтного символа и строки, состоящей из двухбайтных символов соответственно. Пример:

Листинги на странице http://unicross.narod.ru/cpp/
wchar_t str[] = L"string";
std::printf("%ls", str); // string

L — предназначен для вывода значения переменной, имеющей тип long double.
Пример:
long double x = 8e+245;
std::printf("%Le", x); // 8.000000e+245

Следует учитывать, что функции printf(), puts() и printf() применяются в языке
C. Хотя их можно использовать и в языке C++, тем не менее для вывода данных стоит
отдать предпочтение объекту cout. Описание функций printf(), puts() и printf() приведено
в этой книге лишь для того, чтобы вы могли разобраться в чужом коде. В
оставшейся части книги мы будем использовать только объект cout.

Ввод данных в языке C++

Для ввода данных в языке C++ предназначен объект cin, объявленный в файле iostream.
Объект cin позволяет ввести данные любого встроенного типа, например, число,
символ или строку. В этом разделе мы рассмотрим лишь основные возможности этого
объекта, так как для понимания всех возможностей требуется знание принципов
объектно-ориентированного программирования (ООП). В последующих главах мы
вернемся к изучению объекта cin.

Прежде чем использовать объект cin необходимо подключить файл iostream с
помощью директивы include:

#include <iostream>

В качестве примера использования объекта cin произведем суммирование двух целых
чисел, введенных пользователем (листиг 1.6).

Листинг 1.6. Суммирование двух введенных чисел

#include <iostream>

int main() {
   int x = 0, y = 0;
   std::cout << "x = ";
   std::cin >> x;
   std::cout << "y = ";
}

Листинги на страницце http://unicross.narod.ru/cpp/
std::cin >> y;
std::cout << "Summa = " << x + y << std::endl;
return 0;

В первой строке производится подключение файла iostream. Далее внутри функции
main() объявляются две локальные переменные: x и y. Ключевое слово int в начале
строки означает, что объявляются целочисленные переменные. При объявлении
переменным сразу присваивается начальное значение (0) с помощью оператора =. Если
значение не присвоить, то переменная будет иметь произвольное значение, так
называемый "мусор". Как видно из примера, на одной строке можно объявить сразу
несколько переменных, разделив их запятыми.

В следующей строке с помощью объекта cout выводится подсказка для пользователя
("x = "). Благодаря этой подсказке пользователь будет знать, что от него требуется.
После ввода числа и нажатия клавиши <Enter> значение будет присвоено переменной x.
Ввод значения производится с помощью объекта cin. После названия объекта
указывается оператор >>, который как бы указывает направление ввода. При разработке
существенных объектов можно перегрузить этот оператор и тем самым управлять
вводом. После оператора >> передается название переменной, в которой будут
сохранены введенные данные. Так как объекты cout и cin объявлены в пространстве
имен std, название пространства имен необходимо указать перед объектами.

Далее производится вывод подсказки и получение второго числа, которое сохраняется в
переменной y. В следующей строке производится сложение двух чисел и вывод
результата. Процесс ввода двух чисел и получения суммы выглядит так:
x = 10
y = 20
Summa = 30

При вводе данных производится попытка преобразования к типу переменной. В нашем
примере строка "10" будет преобразована в число 10, а строка "20" в число 20. Однако,
вместо числа пользователь может ввести буквы. В этом случае преобразование
закончится неудачей, переменной не будет присвоено значение, а введенное значение
останется в буфере и будет автоматически считано следующей операцией ввода.
Проверить отсутствие ошибки при преобразовании можно с помощью метода good()
объекта cin. Прототип метода:

bool good() const;

Листинги на странице http://unicross.narod.ru/cpp/
Метод good() возвращает логическое значение true, которое соответствует числу 1, если ошибок не произошло, или значение false, соответствующее числу 0, в противном случае. Вывести сообщение об ошибке и прервать выполнение программы можно так:

```cpp
std::cout << "x = ";
std::cin >> x;
if (std::cin.good() == false) {
    std::cout << "Error" << std::endl;
    return 1; // Выходим из функции main()
}
```

В этом примере для проверки условия используется оператор ветвления if. После названия оператора внутри круглых скобок указывается проверяемое выражение. Если выражение возвращает логическое значение true (истина), то будут выполнены инструкции внутри фигурных скобок. Если выражение возвращает значение false (ложь), то инструкции внутри фигурных скобок игнорируются и управление передается инструкции, расположенной сразу после закрывающей фигурной скобки. Проверка значения, возвращаемого методом good(), осуществляется с помощью оператора ==. При ошибке метод good() вернет значение false, следовательно условие false == false будет истинным. В этом случае выводим сообщение об ошибке и выходим из функции main() (и следовательно из программы), возвращая значение 1.

Обратите внимание на то, что оператор проверки на равенство содержит два символа =. Указание одного символа = является логической ошибкой, так как этот оператор используется для присваивания значение переменной, а не для проверки условия. Использовать оператор присваивания внутри логического выражения допускается, поэтому компилятор не выведет сообщение об ошибке, однако программа может выполняться некорректно. Подобную ошибку часто допускают начинающие программисты. Например, в следующем примере вместо проверки равенства числу 11, производится операция присваивания:

```cpp
int x = 10;
if (x = 11) {
    std::cout << "x == 11" << std::endl;
}
```

Любое число не равное 0 трактуется как логическое значение true, поэтому производится проверка условия true == true, которое является истинным. Проверка соответствия условия значению true производится по умолчанию.

Листинги на странице http://unicross.narod.ru/cpp/
Прерывать выполнение программы в случае ошибки мы научились, теперь рассмотрим возможность обработки ошибки (листинг 1.7).

**Листинг 1.7. Обработка ошибок ввода**

```cpp
#include <iostream>

int main() {
    int x = 0, y = 0;
    bool flag = false;
    do {
        std::cout << "x = ";
        std::cin >> x;
        if (!std::cin.good()) {
            std::cout << std::endl << "Error" << std::endl;
            std::cin.clear(); // Сбрасываем флаг ошибки
            std::cin.ignore(255, '\n'); // Очищаем буфер
        }
        else flag = true;
    } while (!flag);
    flag = false;
    do {
        std::cout << "y = ";
        std::cin >> y;
        if (!std::cin.good()) {
            std::cout << std::endl << "Error" << std::endl;
            std::cin.clear(); // Сбрасываем флаг ошибки
            std::cin.ignore(255, '\n'); // Очищаем буфер
        }
        else flag = true;
    } while (!flag);
    std::cout << "Summa = " << x + y << std::endl;
    return 0;
}
```

В этом примере внутри функции `main()` объявляются три локальные переменные: `x`, `y` и `flag`. Переменные `x` и `y` являются целочисленными, а переменная `flag` имеет

Листинги на странице http://unicross.narod.ru/cpp/
логический тип bool. Тип bool позволяет хранить в переменной только два значения: true (истина) и false (ложь). Значение true соответствует числу 1, а значение false — числу 0. В переменной flag мы будем сохранять текущий статус обработки ошибок.

При возникновении ошибки ее необходимо обработать, а затем вывести повторный запрос на ввод числа. Вполне возможно эту процедуру нужно будет выполнить несколько раз. Причем количество повторов заранее неизвестно. Для выполнения одних и тех же инструкций несколько раз предназначены циклы. В нашем примере используется цикл do...while. Инструкции внутри фигурных скобок будут выполняться до тех пор, пока логическое выражение после ключевого слова while является истинным. Проверка условия производится после выполнения инструкций внутри фигурных скобок, поэтому инструкции выполняются минимум один раз. Обратите внимание на логическое выражение !flag. Восклицательный знак, расположенный перед переменной инвертирует значение. Например, если переменная содержит значение true, то выражение возвращает значение false. Так как проверка значения на истинность производится по умолчанию, выражение может не содержать операторов сравнения.

Внутри цикла выводим подсказку пользователю с помощью объекта cout, а затем получаем значение с помощью объекта cin и сохраняем его в переменной. Далее проверяем наличие ошибки с помощью метода good() объекта cin. Если произошла ошибка, то выводим сообщение "Error". Так как ошибка была обработана необходимо сбросить flag ошибки. Для этого предназначен метод clear() объекта cin. Прототип метода:

```c++
void clear(iostate State=goodbit, bool Reraise=false);
```

В прототипе всем параметрам присвоено начальное значение, поэтому, если значения по умолчанию устраивают, параметры можно не указывать.

Сбросить flag ошибки не достаточно, так как ошибочное значение по-прежнему находится в буфере. Если буфер не очистить, то значение будет автоматически передано следующей операции ввода, что породит повторную ошибку. В результате цикл будет выполняться бесконечно. Пригнорировать значение, расположенное в буфере, можно с помощью метода ignore() объекта cin. Прототип метода:

```c++
istream &ignore(streamsize Count=1, int_type Metadelim=BOF);
```

Первый параметр задает максимальное количество считываемых из буфера символов (по умолчанию один символ). Второй параметр задает символ-разделитель. Если

Листинги на странице http://unicross.narod.ru/cpp/
символ-разделитель встретится раньше, чем будет считано указанное количество символов, то считывание прекращается. Метод возвращает ссылку на поток.

В нашем примере очистка выполняется пока не будут считаны 255 символов, либо пока не встретится символ перевода строки (\n). Символ перевода строки также удаляется из буфера. После очистки буфера проверяется значение переменной flag. Так как значение переменной является ложным, цикл будет выполнен еще раз.

Если ошибки нет, выполняются инструкции, расположенные после ключевого слова else. В нашем случае переменной flag присваивается значение true. Это значение является условием выхода из цикла.

Далее таким же способом получаем второе число. Так как в предыдущем цикле значение переменной flag было изменено, перед циклом производим восстановление ложного значения. Иначе выход из цикла произойдет даже в случае ошибки. Если второе число успешно получено, производим вывод суммы чисел.

После объекта cin можно указать сразу несколько переменных, разделяв их операторами >>. В этом случае при вводе все значения можно указать на одной строке. Между собой значения должны разделяться пробелами, табуляцией или символом перевода строки. Эти символы не будут присвоены переменным. Например, программу суммирования двух целых чисел можно записать так:

```cpp
int x = 0, y = 0;
std::cin >> x >> y;
std::cout << "Summa = " << x + y << std::endl;
```

При вводе строки "10 20" число 10 будет присвоено переменной x, а число 20 — переменной y. Пробел игнорируется.

Числа мы вводить научились, теперь попробуем ввести строку:

```cpp
char str[255];
std::cout << "str = ";
std::cin >> str;
std::cout << str << std::endl;
```

В первой строке объявляется символьный массив, состоящий из 255 символов. Строки в языке С++ представляют собой последовательность (массив) символов, последним элементом которого является нулевой символ (\0). Обратите внимание на то, что нулевой символ (нулевой байт) не имеет никакого отношения к числу 0. Коды этих символов разные. В следующей строке выводится подсказка пользователю. Далее объекту cin передается указатель на первый элемент массива str. Если в окне консоли

Листинги на странице http://unicross.narod.ru/cpp/
ввести строку "Hello, world!" и нажать клавишу <Enter>, то массив str будет содержать значение "Hello,\0". Фрагмент "world!" останется в буфере, так как символ пробела является символом-разделителем. Чтобы получить определенное количество символов необходимо воспользоваться методом getline() объекта cin. Прототипы метода:

```cpp
istream &getline(char *Str, streamsize Count);
istream &getline(char *Str, streamsize Count, char Delim);
```

В первом параметре задается указатель на строку, а во втором параметре максимальное количество символов. В параметре delim можно дополнительно указать символ-ограничитель. Метод, соответствующий первому прототипу, производит считывание count-1 символов. Если раньше встретится символ перевода строки (\n), то считывается фрагмент до этого символа. Символ перевода строки считывается, но не записывается в массив. В конце массива автоматически вставляется нулевой символ. В качестве значения метода возвращает ссылку на поток ввода. Пример использования метода getline() приведен листинге 1.8.

Метод, соответствующий второму прототипу, производит считывание count-1 символов. Если раньше встретится символ delim, то считывается фрагмент до этого символа. Символ delim считывается, но не записывается в массив. В конце массива автоматически вставляется нулевой символ. В качестве значения метода возвращает ссылку на поток ввода.

### Листинг 1.8. Ввод строки

```cpp
#include <iostream>

int main() {
    char str[255];
    std::cout << "str = ";
    std::cin.getline(str, 255);
    std::cout << str << std::endl;
    return 0;
}
```

**Ввод данных в языке C**

Язык C++ поддерживает также операции ввода из языка C, объявленные в файле stdio.h. В языке C++ вместо этого файла можно подключать файл cstdio:

Листинги на странице http://unicross.narod.ru/cpp/
#include <stdio>

Для ввода одного символа предназначена функция getchar(). Прототип функции:
int getchar (void);

В качестве значения функция возвращает код введенного символа. Чтобы символ был
считан необходимо после ввода символа нажать клавишу <Enter>. Если было введено
несколько символов, то будет считан первый символ, а остальные останутся в буфере.
Ключевое слово void внутри круглых скобок означает, что функция не принимает
никаких параметров. Пример получения символа:
char ch;
std::printf("ch = ");
ch = std::getchar();

Для ввода строки предназначена функция gets(), однако применять ее в программе не
следует, так как функция не производит никакой проверки длины строки, что может
привести к переполнению буфера. Лучше получать строку позимвольно с помощью
функции getchar(). Прототип функции:
char *gets(char *Buffer);

Для получения и автоматического преобразования данных в конкретный тип (например,
в целое число) предназначена функция scanf(). При вводе строки функция не
производит никакой проверки длины строки, что может привести к переполнению
буфера. Поэтому лучше отказаться от использования этой функции. Прототип функции:
int scanf(const char *Format, ...);

В первом параметре указывается строка специального формата, внутри которой
задаются спецификаторы, аналогичные применяемым в функции printf(), а также
некоторые дополнительные спецификаторы. В последующих параметрах передаются
ссылки на переменные. Пример получения целого числа:
int x;
std::printf("x = ");
std::scanf("%d", &x); // Символ & обязательно!!!

Обратите внимание на то, что перед названием переменной x указан символ &. В этом
случае передается адрес переменной x, а не ее значение. При вводе строки символ & не
указывается, так как название переменной без квадратных скобок является ссылкой на
первый элемент массива. Пример ввода строки:
char *str[255];
std::printf("str = ");
std::scanf("%254s", str); // Символ & не указывается!!!

Листинги на странице http://unicross.narod.ru/cpp/
Считывание символов производится до первого символа-разделителя, например, пробела, табуляции или символа перевода строки. Поэтому после ввода строки "Hello, world!" переменная str будет содержать лишь фрагмент "Hello," а не всю строку.

Следует учитывать, что функции getchar(), gets() и scanf() применяются в языке C. Хотя их можно использовать и в языке C++, тем не менее для ввода данных стоит отдать предпочтение объекту cin. Описание функций getchar(), gets() и scanf() приведено в этой книге лишь для того, чтобы вы могли разобраться в чужом коде. В оставшейся части книги мы будем использовать только объект cin.

**Интерактивный ввод символов**

Объект cin позволяет получить символ только после нажатия клавиши <Enter>. Если необходимо получить символ сразу после нажатия клавиши на клавиатуре, то в VC++ можно воспользоваться функциями _getche() и _getch(), которые объявлены в файле conio.h. Прототипы функций:

```c
#include <conio.h>
int _getche(void);
int _getch(void);
```

**Примечание**
В некоторых компиляторах функции _getche() и _getch() называются getche() и getch() соответственно.

Функция _getche() возвращает код символа и выводит его на экран. Обратите внимание на то, что нормально выводятся только буквы английского алфавита. При вводе русских букв возможно некорректное отображение букв в консоли. При нажатии клавиши на клавиатуре функция _getch() возвращает код символа, но сам символ на экран не выводится. Это обстоятельство позволяет использовать функцию _getch() для получения конфиденциальных данных (например, пароля). Следует учитывать, что код символа возвращается и при нажатии некоторых служебных клавиш, например, <Home>, <End> и др.

В качестве примера использования функции _getch() создадим программу для проверки правильности ввода пароля (листинг 1.9). Пароль может содержать до 16 символов (цифры от 0 до 9 и латинские буквы от "a" до "z" в любом регистре). При нажатии клавиши вместо символа будем выводить звездочку. Если символ не входит в допустимый набор, выведем сообщение об ошибке и завершим программу. Проверку введенного пароля произведем после нажатия клавиши <Enter>.

Листинги на странице http://unicross.narod.ru/cpp/
Листинг 1.9. Ввод пароля

```cpp
#include <iostream>
#include <conio.h>  // Для _getch()
#include <cstring>  // Для strcmp()

int main() {
    char passwd[17], ch;
    bool flag = false;
    int i = 0;
    std::cout << "Password: ";
    do {
        ch = _getch();
        if (i > 15 || ch == \'\r\' || ch == \'\n\') {
            flag = true;
            passwd[i] = \'\0\';
        } else if (
            (ch > 47 && ch < 58)  // Цифры от 0 до 9
            || (ch > 64 && ch < 91)  // Буквы от A до Z
            || (ch > 96 && ch < 123))  // Буквы от a до z
        {
            passwd[i] = ch;
            std::cout << '*';
            ++i;
        }
        else {  // Если недопустимый символ, то выходим
            std::cout << std::endl << "Error" << std::endl;
            return 0;
        }
    } while (!flag);

    // Сравнение паролей
    if (std::strcmp(passwd, "test") == 0) {
        std::cout << std::endl << "OK" << std::endl;
    } else {
        std::cout << std::endl << "Error" << std::endl;
    }

    Листинги на странице http://unicross.narod.ru/cpp/
```
В начале программы производим подключение файлов iostream, conio.h и cstring. Файл iostream необходим для использования объекта cout, файл conio.h — для функции _getch(), а файл cstring — для функции strcmp(), предназначенной для сравнения двух строк. Объект cout и функция strcmp() находятся внутри пространства имен std, в то время как функция _getch() находится в глобальном пространстве имен, поэтому перед функцией _getch() имя пространства не указывается. Далее внутри функции main() объявляем четыре переменные. В символьный массив passwd, содержащий 17 элементов (16 символов + \0), будут записываться введенные символы. В символьную переменную ch будем записывать код нажатой клавиши. Логическая переменная flag предназначена для хранения статуса ввода (если содержит значение true, то ввод пароля закончен), а целочисленная переменная i предназначена для хранения текущего индекса массива passwd. Обратите внимание на то, что первый элемент массива имеет индекс 0, а не 1.

После объявления переменных выводим подсказку пользователю с помощью объекта cout. Затем внутри цикла do...while получаем текущий символ с помощью функции _getch() и сохраняем его код в переменной ch. В следующей строке проверяем выход индекса за границы массива (i > 15) и нажатие клавиши <Enter>. При нажатии клавиши <Enter> передается последовательность символов \r\n (перевод каретки плюс перевод строки), однако функция _getch() может получить только один символ, поэтому проверяется наличие или символа \r или символа \n. Между собой логические выражения разделяются с помощью оператора || (логическое ИЛИ). Чтобы все выражение вернуло истину достаточно, чтобы хотя бы одно из трех логических выражений было истинным. Если выражение i > 15 является ложным, то проверяется выражение ch == '\r', в противном случае все выражение считается истинным и дальнейшая проверка не осуществляется. Если выражение ch == '\r' является ложным, то проверяется выражение ch == '\n', в противном случае все выражение считается истинным и дальнейшая проверка не осуществляется. Если выражение ch == '\n' является ложным, то все выражение считается ложным, в противном случае все выражение считается истинным.

Если выражение является истинным, то ввод пароля закончен. Чтобы выйти из цикла присваиваем переменной flag значение true. Кроме того, вставляем нулевой символ в конец массива. Он будет обозначать конец строки с введенным паролем. После этих инструкций управление передается в конец цикла и производится проверка условия.

Листинг на странице http://unicross.narod.ru/cpp/
flag. Так как условие вернет false (!true) происходит выход из цикла. После выхода из цикла производим проверку правильности пароля с помощью функции strcmp() и выводим соответствующее сообщение. Функция strcmp() принимает в качестве параметров две строки (указатель на первый элемент символьного массива) и возвращает число 0, если строки равны. Сравнение производится с учетом регистра символов. Сравнивать две строки с помощью оператора == нельзя, так как в этом случае производится сравнение адресов переменных, а не символов в строке.

Если ввод пароля не закончен, то производим проверку вхождения символа в допустимый диапазон значений (цифры от 0 до 9 и латинские буквы от "a" до "z" в любом регистре). Выполнить проверку допустимости символа необходимо, так как функция _getch() возвращает также коды некоторых служебных клавиш, например, <Home>, <End> и др. Если условие не соблюдается, то выводим сообщение об ошибке и возвращаем значение 0, тем самым завершая выполнение программы.

Проверяемое условие содержит более сложное выражение, нежели в предыдущем условии. Выражение разбито на несколько мелких выражений с помощью круглых скобок. Внутри первых круглых скобок проверяется соответствие символа диапазону чисел от 0 до 9. Цифра 0 соответствует коду 48, а цифра — коду 57. Коды остальных цифр находятся в диапазоне между этими кодами. Чтобы не перечислять в выражении коды всех цифр используется проверка двух условий, разделенных оператором && (логическое И). Выражение (ch >= 47 && ch <= 58) следует читать так: "если переменная ch содержит символ, имеющий код больше 47 и меньше 58, то вернуть значение true, в противном случае — значение false". Если условие внутри первых скобок является истинным, то все выражение является истинным и дальнейшая проверка не производится. Если условие внутри первых круглых скобок является ложным, то проверяется условие внутри вторых круглых скобок. Условие внутри третьих круглых скобок проверяется только если условие внутри вторых круглых скобок является ложным. Вместо кодов символов можно указать сами символы внутри апострофов. В этом случае условие будет выглядеть так:

```
else if ( (ch >= '0') && (ch <= '9') ) // Цифры от 0 до 9
   || (ch >= 'A' && (ch <= 'Z')) // Буквы от A до Z
   || (ch >= 'a' && (ch <= 'z')) // Буквы от a до z
```

Если символ входит в допустимый диапазон, то сохраняем символ в массиве, выводим звездочку в окно консоли, а затем увеличиваем значение в переменной i на единицу, тем самым перемещая указатель текущей позиции на следующий элемент массива. Выражение ++i соответствует выражению i = i + 1. После этих инструкций

Листинг на странице http://unicross.narod.ru/cpp/
управление передается в конец цикла и производится проверка условия !flag. Так как условие вернет true (!false) инструкции внутри цикла будут выполнены еще раз.

Получение данных из командной строки

Передать данные можно в командной строке после названия файла. Чтобы получить эти данные в программе используется следующий формат функции main():

```c
int main(int argc, char *argv[]) {  
    // Инструкции  
    return 0;  
}
```

Через первый параметр (argc) доступно количество аргументов, переданных в командной строке. Следует учитывать, что первым аргументом является название исполняемого файла. Поэтому значение параметра argc не может быть меньше единицы. Через второй параметр (argv) доступны все аргументы в виде строки (тип char *). Квадратные скобки после названия второго параметра означают, что доступен массив строк. Рассмотрим получение данных из командной строки на примере (листинг 1.10).

Листинг 1.10. Получение данных из командной строки

```c
#include <iostream>

int main(int argc, char *argv[]) {  
    std::cout << "argc = " << argc << std::endl;  
    for (int i=0; i<argc; ++i) {  
        std::cout << argv[i] << std::endl;  
    }  
    return 0;  
}
```

Теперь скопилируем программу в конфигурации Debug (можно и в Release, но тогда необходимо изменить путь к исполняемому файлу) и запустим программу на выполнение с помощью командной строки. Запускаем командную строку. Для этого в меню Пуск выбираем пункт Выполнить. В открывшемся окне набираем команду cmd и нажимаем кнопку ОК. Откроется черное окно, в котором будет приглашение для ввода команд. Переходим в папку C:\book\test\Debug (в моем случае проект с названием "test" расположен в папке C:\book). Для этого набираем команду:

Листинги на странице http://unicross.narod.ru/cpp/
cd C:\book\test\Debug

В командной строке должно быть приглашение:
C:\book\test\Debug>

Для запуска программы вводим команду:

test.exe -param1 -param2

В этой команде мы передаем программе test.exe некоторые данные ("-param1 -param2"). Результат выполнения программы будет выглядеть так:

argc = 3
test.exe
-param1
-param2

Первый элемент массива (argv[0]) не всегда будет содержать только название исполняемого файла. Если в командной строке запуск производится следующим образом:

C:\book\test\Debug\test.exe -param1 -param2
tо элемент будет содержать не только название файла, но и путь к нему:

argc = 3
C:\book\test\Debug\test.exe
-param1
-param2

Предотвращение закрытия окна консоли

При запуске программы с помощью комбинации клавиш <Ctrl>+<F5> в конце автоматически вставляется строка "Для продолжения нажмите любую клавишу". Нажатие любой клавиши приводит к закрытию окна консоли. Однако конечный пользователь таким образом запускать программу не будет. Если запуск производится из командной строки, то пользователь сможет увидеть результат, но если запуск производится с помощью двойного щелчка на значке файла, то окно консоли откроется, а затем сразу закроется. Чтобы окно не закрывалось необходимо вставить инструкцию, ожидаяющую нажатие клавиши. Сделать это можно несколькими способами.

Первый способ заключается в использовании функции _getch() (листинг 1.11). Возвращаемое функцией значение можно пропорционально. Прежде чем использовать функцию необходимо подключить файл conio.h. Прототип функции:

Листинги на странице http://unicross.narod.ru/cpp/
#include <conio.h>

int _getch(void);

Листинг 1.11. Использование функции _getch()

#include <iostream>
#include <conio.h> // Для _getch()

int main() {
    // Инструкции
    std::cout << "Press <Enter> . . . ";
    _getch();
    return 0;
}

Второй способ заключается в использовании функции system() (листинг 1.12). Эта функция позволяет передать команду операционной системе. Для вывода строки "Для продолжения нажмите любую клавишу" и ожидания нажатия клавиши предназначена команда pause. Прежде чем использовать функцию необходимо подключить файл cstdlib (или stdlib.h). Прототип функции:

#include <cstdlib>

int system(const char *Command);

Листинг 1.12. Использование функции system()

#include <iostream>
#include <cstdlib> // Для system()

int main() {
    // Инструкции
    std::system("pause");
    return 0;
}

Два предыдущих способа требовали подключения дополнительных файлов. Кроме того, функция _getch() может не поддерживаться компилятором (возможно потребуется использовать функцию getch()), а функция system() выполняет много лишних операций. Вместо этих способов лучше использовать метод get() объекта cin. Этот метод позволяет получить введенный символ. Ввод осуществляется после нажатия

Листинги на странице http://unicross.narod.ru/cpp/
класси <Enter>. Возвращаемое функцией значение можно проигнорировать. Прототип метода:
#include <iostream>
int get();
При использовании метода get() следует учитывать один нюанс. Если в программе производился ввод данных, то в буфере могут оставаться символы. В этом случае первый символ автоматически будет передан методу get() и окно консоли сразу закроется. Поэтому после ввода данных необходимо дополнительно использовать метод ignore(). Прототип метода:
#include <iostream>
istream &ignore(streamsize Count=1, int_type Metadelim=EOF);
Первый параметр задает максимальное количество считываемых из буфера символов (по умолчанию один символ). Второй параметр задает символ-разделитель. Если символ-разделитель встретится раньше, чем будет считано указанное количество символов, то считывание прекращается. Метод возвращает ссылку на поток вывода.
В большинстве случаев достаточно игнорировать один символ (обычно символ перевода строки), указав название метода без параметров:
std::cin.ignore().get();
Однако в буфере могут остаться и другие символы. Поэтому лучше проигнорировать сразу несколько символов, указав их количество (например, 200) в первом параметре, а символ перевода строки во втором:
std::cin.ignore(200, '
').get();
Если ввода данных не было, то достаточно использовать только метод get(). Пример использования методов get() и ignore() приведен в листинге 1.13.

Листинг 1.13. Использование методов get() и ignore()

#include <iostream>

int main() {
   // Инструкция
   std::cout << "Press <Enter> ... ";
   std::cin.get(); // Если не было ввода данных
   // std::cin.ignore (200, '\n').get(); // Если был ввод данных
   return 0;

   Листинги на странице http://unicross.narod.ru/cpp/
Настройка отображения русских букв в консоли

До сих пор во всех операциях вывода мы использовали латинские буквы. Для этого были основания. Например, попытаемся вывести текст на русском языке:

```cpp
std::cout << "Добро пожаловать в C++!";
```

В результате в окне консоли отобразится нечитаемый текст:

`-юсёю люцрыютъ т C++!`

Причина искажения русских букв заключается в том, что по умолчанию в окне консоли используется кодировка cp866, а в программе мы выводим текст в кодировке Windows-1251. Коды русских букв в этих кодировках отличаются, поэтому происходит искажение. При использовании командной строки пользователь может сменить кодировку вручную, выполнив команду:

```bash
chcp 1251
```

Однако этого не достаточно. Кроме смены кодировки необходимо изменить названия шрифта, так как по умолчанию используются точечные шрифты, которые не поддерживают кодировку Windows-1251. Для нормального отображения русских букв следует в свойствах окна выбрать шрифт Lucida Console.

Все эти действия вы можете произвести на своем компьютере, однако пользователям не знают в какой кодировке выводятся данные из вашей программы. Следовательно необходимо предусмотреть вывод русских букв в текущей кодировке консоли.

Изменить кодировку консоли можно с помощью функции `system()`, однако сменить шрифт довольно проблематично. Проще изменить кодировку текста с Windows-1251 на cp866. При использовании Visual C++ преобразование кодировки производится автоматически после настройки локали (локальных настроек компьютера). Настроить локаль позволяет функция `setlocale()`. Для использования этой функции необходимо подключить файл `locale` (или `locale.h`). Прототип функции:

```cpp
#include <locale>
char *setlocale(int Category, const char *Locale);
```

В первом параметре указывается категория в виде числа от 0 до 5. Вместо чисел можно использовать макроопределения `LC_ALL`, `LC_COLLATE`, `LC_CTYPE`, `LC_MONETARY`, `LC_NUMERIC` и `LC_TIME`. При компиляции макроопределения будут заменены соответствующими числами. Во втором параметре задается название локали в виде строки (например, "Russian_Russia.1251" или ".1251"). Вместо названия можно указать

Листинги на странице http://unicross.narod.ru/cpp/
пустую строку. В этом случае будет использоваться локаль, настроенная в системе. Пример настройки локали и вывода русских букв приведен в листинге 1.14.

**Листинг 1.14. Настройка локали**

```cpp
#include <iostream>
#include <locale>

int main() {
    std::setlocale(LC_ALL, "Russian_Russia.1251");
    std::cout << "Добро пожаловать в C++!" << std::endl;
    std::cout << "Нажмите <Enter> для закрытия окна...";
    std::cin.get();
    return 0;
}
```

При использовании Visual C++ эта программа корректно выводит русские буквы вне зависимости от того, используется в консоли кодировка Windows-1251 или cp866. Однако при использовании других компиляторов (например, MinGW) русские буквы по-прежнему будут отображаться некорректно. Поэтому напишем программу позволяющую вывести текст в кодировке cp866. Кроме того, предусмотрим функцию для преобразования строки из кодировки Windows-1251 в cp866 и обратно (листинг 1.15).

**Листинг 1.15. Файл converts.cpp**

```cpp
#include <iostream> // Для cout и endl
#include <cstring>  // Для strlen(), strcpy() и strcpy_s()
#include <new>      // Для bad_alloc
#include <windows.h> // Для CharToOemA()
#include "converts.h"

// Вывод строки в кодировке CP866
void print_cp866(const char *str, short int x) {
    char *buf;
    try {
        buf = new char[std::strlen(str) + 1];
    }
}
```

Листинги на странице http://unicross.narod.ru/cpp/
catch (std::bad_alloc err) {
    return;
}
CharToOemA(str, buf);
switch (x) {
    case 0: std::cout << buf; break;
    case 1: std::cout << buf << std::endl; break;
    case 2: std::cout << buf << " "; break;
    default: std::cout << buf; break;
}
delete [] buf;

// Преобразование строки в кодировку CP866
void convert_to_cp866(char *str) {
    char *buf;
    try {
        buf = new char[strlen(str) + 1];
    }
    catch (std::bad_alloc err) {
        return;
    }
    CharToOemA(str, buf);
    // std::strcpy(str, buf); // Для MinGW
    strcpy_s(str, std::strlen(str) + 1, buf); // Для VC++
    delete [] buf;
}

// Преобразование строки в кодировку Windows-1251
void convert_to_cp1251(char *str) {
    char *buf;
    try {
        buf = new char[strlen(str) + 1];
    }
    catch (std::bad_alloc err) {
        return;
    }
    OemToCharA(str, buf);

Листинги на странице http://unicross.narod.ru/cpp/
// std::strcpy(str, buf); // Для MinGW
strcpy_s(str, std::strlen(str) + 1, buf); // Для VC++
delete [] buf;
}

Как вы уже знаете, прежде чем идентификатор использовать, его необходимо объявить в программе. Ранее мы объявляли функции перед определением функции main(). В этом примере все функции вынесены в отдельный файл, называемый модулем. При использовании модулей определения функций размещают в файле с расширением crr, а все объявления выносят в отдельный файл, имеющий расширение h. Причем названия файлов должны совпадать. Содержимое файла converts.h приведено в листинге 1.16.

Листинг 1.16. Файл converts.h

#ifndef CONVERTS_H
#define CONVERTS_H

const short int PRINT_ENDL = 1;
const short int PRINT_SPACE = 2;

void print_cp866(const char *str, short int x=0);
void convert_to_cp866(char *str);
void convert_to_cp1251(char *str);
#endif

Все содержимое заголовочного файла расположено внутри условия, которое проверяется перед компиляцией. Условие выглядит следующим образом:
#ifndef CONVERTS_H
// Инструкции
#endif

Это условие следует читать так: "если не существует макроопределения CONVERTS_H, то вставить инструкции в то место, где подключается файл". Условие начинается с директивы #ifndef и заканчивается директивой #endif. Все это необходимо, чтобы объявления идентификаторов не вставлялись дважды. Иначе будут вставлены два одинаковых объявлений, а это является ошибкой. Чтобы объявления не вставлялись дважды в первой инструкции внутри условия объявляется макроопределение CONVERTS_H с помощью директивы #define. Теперь повторная проверка условия вернет ложное значение.

Файл converts.h содержит объявления двух констант (PRINT_ENDL и PRINT_SPACE) и трех функций:

Листинги на странице http://unicross.narod.ru/cpp/
→ print_cp866() — преобразует строку из кодировки Windows-1251 в кодировку cp866 и выводит ее в окно консоли. Исходная строка не изменяется;
→ convert_to_cp866() — преобразует исходную строку из кодировки Windows-1251 в кодировку cp866;
→ convert_to_cp1251() — преобразует исходную строку из кодировки cp866 в кодировку Windows-1251.

Внутри функции print_cp866() объявляется указатель buf, который может ссылаться на данные имеющие тип char. Указатели имеют такое же предназначение, что и переменные, но вместо данных указатели содержат адрес данных в памяти компьютера. Длина передаваемой в качестве первого параметра строки заранее неизвестна, поэтому память выделяется динамически и указателю buf присваивается адрес первого элемента символьного массива, размер которого указан в квадратных скобках. Для получения длины строки используется функция strlen(). Эта функция возвращает длину без учета нулевого символа (который служит признаком конца строки), поэтому к длине строки прибавляется единица.

Выделение динамической памяти может закончиться неудачей, если размера доступной памяти не достаточно. Поэтому инструкция расположена внутри конструкции try...catch, которая позволяет перехватить и обработать исключение. Если внутри блока try возникнет исключение, то управление передается в блок catch, соответствующий типу исключения. При нежвакте памяти исключение имеет тип bad_alloc. Этот тип указывается перед названием переменной (err) внутри круглых скобок после ключевого слова catch. Внутри блока catch можно произвести обработку исключения. В нашем случае просто производится выход из функции. Если исключение не возникло или произошло исключение другого типа, то инструкции внутри блока catch не выполняются.

После успешного выделения динамической памяти производится преобразование кодировок с помощью функции charToOemA(). В первом параметре передается строка в кодировке Windows-1251, а во втором параметре передается указатель на динамическую область памяти, которая была выделена ранее. В эту область будет записана строка в кодировке cp866.

Далее в зависимости от значения переменной x производится вывод строки в кодировке cp866 в окно консоли. Значение проверяется с помощью оператора switch. Если значение переменной x равно 0 (case 0) или переменная имеет значение отличное от 1 и 2 (блок default), то производится вывод только строки. Если переменная имеет значение 1 (case 1), то после строки выводится символ перевода строки. Если

Листинги на странице http://unicross.narod.ru/cpp/
переменная имеет значение 2 (case 2), то после строки выводится пробел. Обратите внимание на то, что в конце каждого блока расположен оператор break, который передает управление инструкции, расположенной после закрывающей фигурной скобки. При вызове функции значения 0, 1 и 2 не несут никакой смысловой нагрузки. Поэтому значение 0 определено по умолчанию в объявлении функции и его можно вообще не указывать при вызове. Значение 1 закреплено за константой PRINT_ENDL, а значение 2 — за константой PRINT_SPACE. Теперь при вызове функции вместо чисел можно использовать константы. Например, вывод строки и дальнейший перевод на новую строку выглядит так:

```c
print_cp866("Строка", PRINT_ENDL);
```
Согласитесь, это понятнее, чем:

```c
print_cp866("Строка", 1);
```
После выделения динамической памяти программист должен самостоятельно позаботиться о возвращении памяти операционной системе. Если этого не сделать и присвоить указателю другой адрес, то область памяти будет потеряна. Несколько подобных промашек приведет к утечке памяти и один прекрасный момент свободная память закончится. Восстановить память можно будет только после перезагрузки компьютера. Вернуть область памяти операционной системе из программы позволяет оператор delete, после которого передается указатель на область. Если ранее выделялась память для массива, то между оператором delete и указателем вставляются квадратные скобки. После применения оператора delete, использовать указатель нельзя.

Внутри функции convert_to_cp866() также выделяется динамическая память, а затем производится преобразование кодировок с помощью функции charToOemA(). Далее с помощью функции strcpy_s() (для MinGW с помощью функции strcpy()) производится копирование символов из динамической памяти в исходную строку, а затем область возвращается операционной системе. Таким образом изменяется кодировка в исходной строке.

Функция convert_to_cp1251() аналогична функции convert_to_cp866(), но преобразование кодировки производится с помощью функции OemToCharA().

Теперь можно подключить эти файлы к проекту. Для создания файла converts.cpp в окне Обозреватель решений щелкаем правой кнопкой мыши на пункте Файлы исходного кода и из контекстного меню выбираем пункт Добавить | Создать элемент. В результате откроется окно Добавление нового элемента. Выделяем пункт Файл C++ (.cpp), вводим название файла ("converts") и нажимаем кнопку Добавить. В результате

Листинг 1 на странице http://unicross.narod.ru/cpp/
файл будет добавлен в папку проекта и его название отобразится в окне Обозреватель решений, а сам файл будет открыт на отдельной вкладке. Вставляем в этот файл код из листинга 1.15.

Для создания файла converts.h в окне Обозреватель решений щелкаем правой кнопкой мыши на пункте Заголовочные файлы и из контекстного меню выбираем пункт Добавить | Создать элемент. В результате откроется окно Добавление нового элемента. Выделяем пункт Заголовочный файл (.h), вводим название файла ("converts") и нажимаем кнопку Добавить. Вставляем в этот файл код из листинга 1.16.

Теперь, чтобы использовать эти функции, достаточно подключить файл converts.h в основной программе. Пример использования функций приведен в листинге 1.17.

Листинг 1.17. Файл main.cpp

```
#include <iostream>
#include "converts.h"

int main() {
    print_cp866("Добро пожаловать", PRINT_SPACE);
    print_cp866("в C++!", PRINT_ENDL);
    char s[] = "Нажмите <Enter> для закрытия окна...");
    convert_to_cp866(s);
    std::cout << s;
    std::cin.get(); // Ожидаем нажатия <Enter>
    return 0;
}
```

Результат выполнения в окне консоли:
Добро пожаловать в C++!
Нажмите <Enter> для закрытия окна...

Примечание
В Visual C++ нельзя использовать функцию print_cp866(), если предварительно была произведена настройка локалы с помощью функции setlocale(), так как перекодировка будет произведена дважды.

Листинги на странице http://unicross.narod.ru/cpp/
Преждевременное завершение выполнения программы

В некоторых случаях может возникнуть условие, при котором дальнейшее выполнение программы лишено смысла, например, отсутствует свободная память при использовании динамической памяти. В этом случае имеет смысл вывести сообщение об ошибке и прервать выполнение программы досрочно. Для этого предназначена функция exit(). Прежде чем использовать функцию необходимо предварительно подключить файл cstdlib (или stdlib.h в языке C). Прототип функции:

```c
#include <cstdlib>
void exit(int Code);
```

В качестве параметра функция принимает число, которое является статусом завершения. Число 0 означает нормальное завершение программы, а любое другое число — некорректное завершение. Эти числа передаются операционной системе. Вместо чисел можно использовать макроопределения EXIT_SUCCESS (нормальное завершение) и EXIT_FAILURE (аварийное завершение). Объявления макросов:

```c
#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1
```

Помимо функции exit() для аварийного завершения программы предназначена функция abort(). В этом случае завершение программы осуществляется операционной системой (выводится окно с сообщением "обнаружена ошибка. Приложение будет закрыто"). Прежде чем использовать функцию необходимо предварительно подключить файл cstdlib (или stdlib.h в языке C). Прототип функции:

```c
#include <cstdlib>
void abort(void);
```

В качестве примера произведем деление двух вещественных чисел, введенных пользователем, и выведем результат. При этом обработаем ошибки ввода и деление на нуль (листинг 1.18). Все сообщения об ошибках выведем на русском языке.

```
Листинг 1.18. Преждевременное завершение выполнения программы
```

```c
#include <iostream>
#include <locale>
#include <cstdlib>

int main() {
    double x = 0.0, y = 0.0;
    
    Листинги на странице http://unicross.narod.ru/cpp/
```
std::setlocale(LC_ALL, "Russian_Russia.1251");
std::cout << "x = ";
std::cin >> x;
std::cout << "y = ";
std::cin >> y;
if (!std::cin.good()) {
    std::cout << std::endl << "Ошибка при вводе чисел!" << std::endl;
    // Сбрасываем флаг ошибки, иначе окно закроется
    std::cin.clear();
    std::cin.ignore(200, '\n').get();
    std::exit(1); // Завершаем выполнение программы
}
else if (y == 0.0) {
    std::cout << "На нуль делить нельзя!" << std::endl;
    std::cin.ignore(200, '\n').get();
    std::exit(EXIT_FAILURE); // Аналогично std::exit(1);
}
std::cout << "x / y = " << x / y << std::endl;
std::cin.ignore(200, '\n').get();
return 0;
}

В этом примере вместо функции exit() можно было воспользоваться инструкцией return, так как завершение программы выполнялось внутри функции main(). Однако в больших программах основная часть кода расположена вне функции main() и в этом случае инструкцией return для завершения всей программы уже не обойтись.

Листинги на странице http://unicross.narod.ru/cpp/
Программирование на C++ в Visual Studio® 2010 Express

Глава 2.
Переменные и типы данных

© Прохоренок Н.А., 2010 г., unicross@mail.ru

Этот DjVu-файл предоставляется КАК ЕСТЬ. Автор не несет никакой
ответственности за прямые или косвенные проблемы, связанные с
использованием данного файла. ВЫ ИСПОЛЬЗУЕТЕ ЕГО НА СВОЙ
СТРАХ И РИСК.

Все названия программных продуктов являются зарегистрированными
торговыми марками соответствующих фирм.

Запрещено:
- перепечатывать всю книгу или отдельные главы без письменного разрешения
  Автора;
- декомпилировать данный файл и преобразовывать его в любой другой формат.
Переменные — это участки памяти, используемые программой для хранения данных. Прежде чем использовать переменную, ее необходимо предварительно обьявить глобально (вне функций) или локально (внутри функции). В большинстве случаев объявление переменной является сразу и ее определением. Глобальные переменные видны внутри всех функций в файле, а локальные переменные видны только внутри той функции, в которой они объявлены. Для объявления переменной используется следующий формат:

```
[<Спецификатор>][<Модификатор>] <Тип> <Переменная1> [=<Значение1>] 
[, . . . , <ПеременнаяN> [=<ЗначениеN>]];
```

Именование переменных

Какая переменная должна иметь уникальное имя, состоящее из латинских букв, цифр и знака подчеркивания, причем имя переменной не может начинаться с цифры. При указании имени переменной важно учитывать регистр букв: х и Х — разные переменные.

В качестве имени переменной нельзя использовать ключевые слова. Список ключевых слов приведен в табл. 2.1. Следует учитывать, что в некоторых компиляторах могут быть определены дополнительные ключевые слова. Запоминать все ключевые слова нет необходимости, так как в редакторе эти слова подсвечиваются. Любая попытка использования ключевого слова вместо названия переменной приведет к ошибке при компиляции. Помимо ключевых слов следует избегать совпадений со встроенными идентификаторами.

Правильные имена переменных: х, у, str_name, strName.

Неправильные имена переменных: 1у, ИмяПеременной.

Последнее имя неправильное, так как в нем используются русские буквы. Хотя на самом деле такой вариант также будет работать, но лучше русские буквы все же не применять.

Листинги на странице http://unicross.narod.ru/cpp/
Таблица 2.1. Ключевые слова языка C++

<table>
<thead>
<tr>
<th>asm</th>
<th>else</th>
<th>new</th>
<th>this</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
<td>enum</td>
<td>operator</td>
<td>throw</td>
</tr>
<tr>
<td>bool</td>
<td>explicit</td>
<td>private</td>
<td>true</td>
</tr>
<tr>
<td>break</td>
<td>export</td>
<td>protected</td>
<td>try</td>
</tr>
<tr>
<td>case</td>
<td>extern</td>
<td>public</td>
<td>typedef</td>
</tr>
<tr>
<td>catch</td>
<td>false</td>
<td>register</td>
<td>typeid</td>
</tr>
<tr>
<td>char</td>
<td>float</td>
<td>reinterpret_cast</td>
<td>typename</td>
</tr>
<tr>
<td>class</td>
<td>for</td>
<td>return</td>
<td>union</td>
</tr>
<tr>
<td>const</td>
<td>friend</td>
<td>short</td>
<td>unsigned</td>
</tr>
<tr>
<td>const_cast</td>
<td>goto</td>
<td>signed</td>
<td>using</td>
</tr>
<tr>
<td>continue</td>
<td>if</td>
<td>sizeof</td>
<td>virtual</td>
</tr>
<tr>
<td>default</td>
<td>inline</td>
<td>static</td>
<td>void</td>
</tr>
<tr>
<td>delete</td>
<td>int</td>
<td>static_cast</td>
<td>volatile</td>
</tr>
<tr>
<td>do</td>
<td>long</td>
<td>struct</td>
<td>wchar_t</td>
</tr>
<tr>
<td>double</td>
<td>mutable</td>
<td>switch</td>
<td>while</td>
</tr>
<tr>
<td>dynamic_cast</td>
<td>namespace</td>
<td>template</td>
<td></td>
</tr>
</tbody>
</table>

Типы данных

В языке C++ определены следующие элементарные типы данных:

- **bool** — логический тип данных. Может содержать значения true (соответствует числу 1) или false (соответствует числу 0). Пример объявления переменной:
  ```
  bool is_int;
  ```

- **char** — код символа. Диапазон значений от -128 до 127. Пример объявления переменной:
  ```
  char ch;
  ```

- **wchar_t** — двухбайтовый символ. Пример объявления переменной:

  Листинги на странице http://unicross.narod.ru/cpp/
wchar_t wch;

- int — целое число со знаком. Диапазон значений зависит от разрядности операционной системы. В 16-битной операционной системе диапазон от -32 768 до 32 767. В 32-битной операционной системе диапазон от -2 147 483 648 до 2 147 483 647. Пример объявления переменной:
  ```
  int x;
  ```

- float — вещественное число. Пример объявления переменной:
  ```
  float y;
  ```

- double — вещественное число двойной точности. Пример объявления переменной:
  ```
  double z;
  ```

Перед элементарным типом данных могут быть указаны следующие модификаторы или их комбинация:

- signed — указывает, что символьный или целочисленный типы могут содержать отрицательные значения. Тип signed char соответствует типу char и может содержать значения от -128 до 127. Пример объявления переменной:
  ```
  signed char ch;
  ```

Тип signed int соответствует типу int. Пример объявления переменной:
  ```
  signed int x;
  ```

- unsigned — указывает, что символьный или целочисленный типы не могут содержать отрицательные значения. Тип unsigned char может содержать значения от 0 до 255. Пример объявления переменной:
  ```
  unsigned char ch;
  ```

Диапазон значений для типа unsigned int зависит от разрядности операционной системы. В 16-битной операционной системе диапазон от 0 до 65 535. В 32-битной операционной системе диапазон от 0 до 4 294 967 295. Пример объявления переменной:
  ```
  unsigned int x;
  ```

- short — может быть указан перед целочисленным типом. Диапазон значений у типов short int и signed short int от -32 768 до 32 767. Пример объявления переменных:
  ```
  short int x;
  ```

Листинги на странице http://unicross.narod.ru/cpp/
signed short int y;

Диапазон значений у типа unsigned short int от 0 до 65535. Пример объявления переменной:
unsigned short int x;

→ long — может быть указан перед целочисленным типом и типом double. Диапазон значений у типов long int и signed long int от -2147483648 до 2147483647. Пример объявления переменных:
long int x;
signed long int y;

Диапазон значений у типа unsigned long int от 0 до 4294967295. Пример объявления переменной:
unsigned long int x;

Ключевое слово long перед целым типом может быть указано дважды. Диапазон значений у типов long long int и signed long long int от -9223372036854775808 до 9223372036854775807. Пример объявления переменных:
long long int x;
signed long long int y;

Диапазон значений у типа unsigned long long int может быть от 0 до 18446744073709551615. Пример объявления переменной:
unsigned long long int x;

Ключевое слово long можно указать также перед типом double. Пример объявления переменной:
long double z;

При использовании модификаторов тип int подразумевается по умолчанию, поэтому тип int можно не указывать. Пример объявления переменных:
short x; // Эквивалентно: short int x;
long y; // Эквивалентно: long int y;
signed z; // Эквивалентно: signed int z;
unsigned k; // Эквивалентно: unsigned int k;

В одной инструкции можно объявить сразу несколько переменных, указав из через запятую после названия типа данных:

Листинги на странице http://unicross.narod.ru/cpp/
int x, y, z;
Если переменная может изменять свое значение извне, то перед модификатором указывается ключевое слово volatile. Это ключевое слово предотвращает проведение оптимизации программы, при котором предполагается, что значение переменной может быть изменено только в программе.

После объявления переменной под нее выделяется определенная память, размер которой зависит от используемого типа данных и разрядности операционной системы. Например, тип int в 16-битной системе занимает 16 бит, в 32-битной — 32 бита, а в 64-битной — 64 бита. Чтобы сделать код машинонезависимым следует определять размер типа с помощью оператора sizeof. Оператор имеет два формата:

<Размер> = sizeof <Переменная>;
<Размер> = sizeof (тип данных);

Пример:

```cpp
int x;
std::cout << sizeof x << std::endl;
std::cout << sizeof (x) << std::endl;
std::cout << sizeof (bool) << std::endl;
```

Обратите внимание на то, что тип данных обязательно должен быть указан внутри круглых скобок, в то время как название переменной можно указать как внутри скобок, так без них.

В качестве примера использования оператора sizeof выведем текущий размер всех типов данных (листиг 2.1).

```
Листинг 2.1. Использование оператора sizeof

#include <iostream>
using namespace std;

int main() {
    cout << "bool = " << sizeof (bool) << endl;
    cout << "char = " << sizeof (char) << endl;
    cout << "wchar_t = " << sizeof (wchar_t) << endl;
    cout << "int = "   << sizeof (int)   << endl;
    cout << "short int = " << sizeof (short int) << endl;
    cout << "long int = " << sizeof (long int) << endl;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
cout << "long long int = " << sizeof (long long int) << endl;
cout << "float = " << sizeof (float) << endl;
cout << "double = " << sizeof (double) << endl;
cout << "long double = " << sizeof (long double) << endl;
cin.get();
return 0;
}

Результат выполнения программы в Visual C++ 2010 Express в 32-битной операционной системе Windows XP (размер указан в байтах):
bool = 1
char = 1
wchar_t = 2
int = 4
short int = 2
long int = 4
long long int = 8
float = 4
double = 8
long double = 8

Инициализация переменных

При объявлении переменной её можно сразу присвоить начальное значение, указав его после оператора =. Эта операция называется инициализацией переменных. Пример укзания значения:
int x, y = 10, z = 30, k;
Переменная становится видимой сразу после объявления, поэтому на одной строке с объявлением (после запятой) эту переменную уже можно использовать для инициализации других переменных:
int x = 5, y = 10, z = x + y; // z равно 15
При инициализации допустимо указывать значение не только после оператора =, но и внутри круглых скобок после имени переменной:
int x(10); // x равно 10
Инициализация глобальных (объявленных вне функций) переменных производится только один раз. Локальные (объявленные внутри функций) переменные инициализируются при каждом вызове функции, а статические (сохраняющие свое

Листинги на странице http://unicross.narod.ru/cpp/
значение между вызовами) локальные переменные — один раз при первом вызове функции. Если при объявлении переменной значение не было присвоено, то:

- глобальные переменные автоматически получают значение 0;
- локальным переменным значение не присваивается. Переменная будет содержать произвольное значение, так называемый "мусор";
- статические локальные переменные автоматически получают значение 0.

Присвоить значение переменной можно уже после объявления, указав его после оператора =. Эта операция называется присваиванием. Пример присваивания:

```c
int x;
x = 10;
```

Переменной, имеющей тип bool, можно присвоить значения true или false. Значение true соответствует числу 1, а значение false — числу 0. Числовые значения можно указать вместо значений true и false. Пример:

```c
bool a, b, c, d;
a = true;
b = false;
c = 1; // Эквивалентно: c = true;
d = 0; // Эквивалентно: d = false;
```

Переменной, имеющей тип char, можно присвоить числовое значение (код символа) или указать символ внутри апострофов. Обратите внимание на то, что использовать кавычки нельзя, так как в этом случае вместо одного символа будет два: собственно сам символ плюс нулевой символ. Пример:

```c
char ch1, ch2;
ch1 = 119; // Буква w
ch2 = 'w'; // Буква w
```

Внутри апострофов можно указать специальные символы. Специальные символы — это комбинации знаков, обозначающих служебные или непечатаемые символы. Перечислим специальные символы, доступные в C++:

- \0 — нулевой символ;
- \n — перевод строки;
- \r — возврат каретки;
- \t — горизонтальная табуляция;

Листинги на странице http://unicross.narod.ru/cpp/
→ \\v — вертикальная табуляция;
→ \a — звуковой сигнал;
→ \b — возврат на один символ;
→ \f — перевод формата;
→ \' — апостроф;
→ \" — кавычка;
→ \? — знак вопроса;
→ \\ — обратная косая черта;
→ \n — восьмеричное значение N;
→ \xn — шестнадцатеричное значение N.

Пример указания восьмеричного и шестнадцатеричного значений:
```c
char ch1, ch2;
ch1 = '\167'; // Буква w (восьмеричное значение)
ch2 = '\x77'; // Буква w (шестнадцатеричное значение)
```

Переменной, имеющей тип wchar_t, значение присваивается также, как и переменной, имеющей тип char, только перед отрывающим апострофом указывается буква L:
```c
wchar_t ch;
ch = L'w';
```

Целочисленное значение задается в десятичной, восьмеричной или шестнадцатеричной форме. Восьмеричные числа начинаются с нуля и содержат цифры от 0 до 7. Шестнадцатеричные числа начинаются с комбинации символов 0x (или 0X) и могут содержать числа от 0 до 9 и буквы от A до F (регистр букв не имеет значения). Восьмеричные и шестнадцатеричные значения преобразуются в десятичное значение. Пример:
```c
int x, y, z;
x = 119; // Десятичное значение
y = 0167; // Восьмеричное значение
z = 0x77; // Шестнадцатеричное значение
```

Присвоить переменной нулевое значение можно указав круглые скобки после названия типа:
```c
int x = int(); // x равно 0
```

Листинги на странице http://unicross.narod.ru/cpp/
По умолчанию целочисленные константы имеют тип `signed int`. Если необходимо изменить тип на другой, то после числа указываются следующие буквы:

- L (или l) — тип `long int`. Пример указания значения: `10L`;
- U (или u) — тип `unsigned int`. Пример указания значения: `10u`;
- если буквы U и L указаны одновременно, то тип будет `unsigned long int`. Пример указания значения: `10UL`.

Вещественное число может содержать точку и (или) экспоненту, начинающуюся с буквы e (регистр не имеет значения):

```c
float x, y;
double z, k;
x = 20.0;
y = 12.1e5;
z = .123;
k = 47.E-5;
```

Присвоить переменной нулевое значение можно указав круглые скобки после названия типа:

```c
float x = float();
double y = double();
```

По умолчанию вещественные константы имеют тип `double`. Если необходимо изменить тип на другой, то после числа указываются следующие буквы:

- f (или f) — тип `float`. Пример указания значения: `12.3f`;
- u (или u) — тип `long double`. Значение должно содержать точку и (или) экспоненту, иначе тип будет `long int`. Пример указания значения: `12.3L`.

**Оператор `typedef`**

Оператор `typedef` позволяет создать псевдоним для существующего типа данных. В дальнейшем псевдоним можно указывать при объявлении переменной. Оператор имеет следующий формат:

```c
typedef <Существующий тип> <Псевдоним>;
```

В качестве примера создадим псевдоним для типа `long int`:

```c
typdef long int lint;
lint x = 5, y = 10;
```

Листинги на странице http://unicross.narod.ru/cpp/
После создания псевдонима, его имя можно использовать при создании другого псевдонима:

typedef long int lint;
typedef lint newint;
newint x = 5, y = 10;

Псевдонимы предназначены для создания машиннозависимых программ. При переносе программы на другой компьютер достаточно будет изменить одну строку. Подобный подход часто используется в стандартной библиотеке. Например, прототип функции strlen(), позволяющей получить длину строки, выглядит так:

size_t strlen(const char *Str);

В этом прототипе тип данных size_t, возвращаемый функцией strlen(), является псевдонимом для типа unsigned int, а не новым типом.

Динамическое определение типа данных

Для определения типа данных при выполнении программы предназначен оператор typeid. Прежде чем использовать оператор, необходимо подключить файл typeinfo:

```cpp
#include <typeinfo>
```

Оператор имеет следующий формат:

```cpp
<Экземпляр класса type_info> = typeid(<Данные>);
```

В качестве значения оператор typeid возвращает экземпляр класса type_info. Этот экземпляр можно сравнить с другим экземпляром того же класса, используя операторы == (равно) и != (не равно). Получить название типа данных позволяет метод name(). Пример использования оператора typeid приведен в листинге 2.2.

### Листинг 2.2. Динамическое определение типа данных

```cpp
#include <iostream>
#include <typeinfo>

int main() {
    long int x = 5;
    int y = 12;
    double z = 3.14;
    std::cout << typeid(x).name() << std::endl; // long
    std::cout << typeid(y).name() << std::endl; // int
}
```

Листинги на странице http://unicross.narod.ru/cpp/
std::cout << typeid(z).name() << std::endl; // double
std::cout << typeid(12.1f).name() << std::endl; // float
if (typeid(x) == typeid(y)) {
    std::cout << "==" << std::endl;
} else {
    std::cout << "!=" << std::endl; // !=
}
if (typeid(x) != typeid(y)) {
    std::cout << "!=" << std::endl; // !=
} else {
    std::cout << "==" << std::endl;
}
std::cin.get();
return 0;

Константы

Константы — это переменные, значения в которых не должны изменяться во время работы программы. В более широком смысле под константой понимают любое значение, которое нельзя изменить, например, 10, 12.5, 'W', "string".

При объявлении константы перед типом данных указывается ключевое слово const:

```cpp
class int X = 5;
```

Обратите внимание на то, что в названии константы принято использовать буквы только в верхнем регистре. Если название константы состоит из нескольких слов, то между словами указывается символ подчеркивания. Это позволяет отличить внутри программы константу от обычной переменной. После объявления константы, ее можно использовать в выражениях:
```cpp
class int X = 5;

int y;

y = X + 20;
```

Присвоить значение константе можно только при объявлении. Любая попытка изменения значения в программе приведет к ошибке при компиляции: "error C3892: невозможно присваивать значения переменной, которая объявлена как константа".

Листинги на странице http://unicross.narod.ru/cpp/
Создать константу можно также с помощью директивы `#define`. Значение, указанное в этой директиве, подставляется в выражение до компиляции. Название, указанное в директиве `#define`, принято называть макроопределением или макросом. Директива имеет следующий формат:

`#define <Название макроса> <Значение>`

Пример использования директивы `#define` приведен в листинге 2.3.

Листинг 2.3. Использование директивы `#define`

```cpp
#include <iostream>
#define X 5

int main() {
    int y;
    y = X + 20;
    std::cout << y << std::endl;
    std::cin.get();
    return 0;
}
```

Обратите внимание на то, что оператор `=` не используется и в конце инструкции точка с запятой не указывается. Если точку с запятой указать, то значение вместе с ней будет вставлено в выражение. Например, если определить макрос так:

`#define X 5;`

tо после подстановки значения, инструкция

```
y = X + 20;
```

будет выглядеть следующим образом:

```
y = 5; + 20;
```

Точка с запятой после цифры 5 является концом инструкции, поэтому переменной `y` будет присвоено значение 5, а не 25. Подобная ситуация приводит к ошибкам, которые трудно найти, так как в этом случае инструкция `+ 20;` не возбуждает ошибку при компиляции.

В качестве значения макроса можно указать целое выражение, например:

`#define X 5 + 5`
Это значение также может привести к недоразумениям. Никакого вычисления выражения не происходит. Все выражение целиком подставляется вместо названия макроса. Если инструкция выглядит так:

```
y = x * 20;
```
то после подстановки значения, инструкция примет следующий вид:

```
y = 5 + 5 * 20;
```
Приоритет оператора умножения выше приоритета оператора сложения, поэтому число 5 будет умножено на 20, а затем к результату прибавлено число 5. Таким образом, результат будет 105, а не 200, как это было бы при использовании константы:

```c
const int x = 5 + 5;
int y;
y = x * 20; // 200
```
В качестве значения макроса можно указать строку в кавычках:

```
#define ERR "Сообщение об ошибке"
```
При указании длинной строки следует учитывать, что определение макроса должно быть расположено на одной строке. Если нужно разместить значение на нескольких строках, то в конце строки необходимо добавить обратную косую черту. После косой черты не должно быть никаких символов, в том числе и комментариев. Пример:

```
#define ERR "Сообщение об ошибке \nна нескольких \nстроках"
```
В языке C++ существуют встроенные макросы (до и после названия два символа подчеркивания):

- `__FILE__` — имя файла;
- `__LINE__` — номер текущей строки;
- `__DATE__` — дата компиляции файла;
- `__TIME__` — время компиляции файла.
Помимо этих макросов существует множество других, объявленных в различных заголовочных файлах. Эти макросы мы рассмотрим в соответствующих разделах книги. Выведем текущие значения встроенных макросов (листинг 2.4).
Листинг 2.4. Встроенные макросы

#include <iostream>

int main() {
    std::cout << __LINE__  << std::endl;
    std::cout << __FILE__  << std::endl;
    std::cout << __DATE__  << std::endl;
    std::cout << __TIME__  << std::endl;
    std::cin.get();
    return 0;
}

Примерный результат выполнения:
4
c:\book\test\test\main.cpp
Aug 30 2010
15:58:35

Спецификаторы хранения

Перед модификатором и типом могут быть указаны следующие спецификаторы:

- auto — так как локальные переменные по умолчанию являются автоматическими, ключевое слово auto в C++ практически не используется. Поддерживается только в целях совместимости с языком C. В VC++ 2010 ключевое слово auto по умолчанию имеет несколько иной смысл, который определен в новом стандарте C++0x. В этом стандарте ключевое слово auto можно использовать вместо типа данных в объявлении переменной. Тип такой переменной определяется автоматически по значению, указанному при инициализации. Пример:
  auto x = 10;
  std::cout << typeid(x).name()  << std::endl; // int
  auto y = 10.2f;
  std::cout << typeid(y).name()  << std::endl; // float
  auto z = 10.2;
  std::cout << typeid(z).name()  << std::endl; // double

Листинги на странице http://unicross.narod.ru/cpp/
`register` — является подсказкой компилятору, что переменная будет использоваться интенсивно. Для ускорения доступа, значение такой переменной сохраняется в регистрах процессора. Компилятор может проигнорировать это объявление и сохранить значение в памяти. Ключевое слово может использоваться при объявлении переменной внутри блока или в параметрах функции. К глобальным переменным не применяется. Пример объявления переменной:
```
register int i;
```

`extern` — сообщает компилятору, что переменная определена в другом месте, например, в другом файле. Ключевое слово лишь объявляет переменную, а не определяет ее. Таким образом, память под переменную повторно не выделяется. Если при объявлении производится инициализация переменной, то объявление становится определением переменной. В качестве примера объявим переменную внутри функции `main()`, а определение переменной разместим после функции:
```
#include <iostream>

int main() {
    extern int i; // Определение в другом месте
    std::cout << i << std::endl; // 10
    std::cin.get();
    return 0;
}
```

`static` — если ключевое слово указано перед локальной переменной, то значение будет сохраняться между вызовами функции. Инициализация статических локальных переменных производится только при первом вызове функции. При следующих вызовах используется сохраненное ранее значение. Создадим функцию со статической переменной. Внутри функции увеличим значение переменной на единицу, а затем выведем значение в окно консоли. Далее вызовем эту функцию несколько раз:
```
int i = 10; // Определение переменной i
```

Листинги на странице http://unicross.narod.ru/cpp/
#include <iostream>

void func();

int main() {
    func(); // 1
    func(); // 2
    func(); // 3
    std::cin.get();
    return 0;
}

void func() {
    static int x = 0; // Статическая переменная
    ++x; // Увеличиваем значение на 1
    std::cout << x << std::endl;
}

При каждом вызове функции func() значение статической переменной x будет увеличиваться на единицу. Если убрать ключевое слово static, то при каждом вызове будет выводиться число 1, так как локальные переменные инициализируются при входе в функцию и уничтожаются при выходе из нее.

Если ключевое слово static указано перед глобальной переменной, то ее значение будет видимо только в пределах файла. Вместо статических глобальных переменных лучше использовать неименованные пространства имен.

Области видимости переменных

Прежде чем использовать переменную, ее необходимо предварительно объявить. До объявления переменной, она не видна в программе. Объявить переменную можно глобально (вне функций) или локально (внутри функции или блока).

Глобальные переменные — это переменные, объявленные в программе вне функций. Глобальные переменные видны в любой части программы, включая функции. Инициализация таких переменных производится только один раз. Если при объявлении

Листинги на странице http://unicross.narod.ru/cpp/
переменной не было присвоено начальное значение, то производится автоматическая
инициализация нулевым значением.

Локальные переменные — это переменные, которые объявлены внутри функции или
блока (области, ограниченной фигурными скобками). Локальные переменные видны
только внутри функции или блока. Инициализация таких переменных производится при
каждом вызове функции или входе в блок. После выхода из функции или блока
локальная переменная уничтожается. Если при объявлении переменной не было
присвоено начальное значение, то переменная будет содержать произвольное значение,
так называемый "мусор". Исключением являются статические локальные переменные,
которым автоматически присваивается нулевое значение и которые сохраняют значение
при выходе из функции.

Если имя локальной переменной совпадает с именем глобальной переменной, то все
операции внутри функции осуществляются с локальной переменной, а значение
gлобальной не изменяется. Чтобы в этом случае получить доступ к глобальной
переменной, необходимо перед названием переменной указать два двоеточия,
например, ::x.

Область видимости глобальных и локальных переменных показана в листинге 2.5.

```
Листинг 2.5. Область видимости переменных

#include <iostream>

int x = 10; // Глобальная переменная
void func();

int main() {
    func();
    // Переменные x и у из функции func() здесь не видны
    // Вывод значения глобальной переменной x
    std::cout << x << std::endl; // 10
    { // Блок
        int z = 30;
        std::cout << z << std::endl; // 30
    }
    // Переменная z здесь уже не видна!!!
    for (int i=0; i<10; ++i) {

    }

Листинги на странице http://unicross.narod.ru/cpp/
```
std::cout << i << std::endl;
}
// Переменная i здесь уже не видна!!!
std::cin.get();
return 0;
}

void func() {
  // Переменная z здесь не видна
  int x = 5, y = 20;  // Локальные переменные
  // Вывод значения локальной переменной x
  std::cout << x << std::endl;  // 5
  // Вывод значения глобальной переменной x
  std::cout << ::x << std::endl;  // 10
}

Очень важно учитывать, что переменная, объявленная внутри блока, видна только в пределах блока (внутри фигурных скобок). Например, присвоим значение переменной в зависимости от некоторого условия:

// Неправильно!!!
if (x == 10) {  // Какое-то условие
  int y;
  y = 5;
}
else {
  int y;
  y = 25;
}

// Переменная y здесь не определена!!!

В этом примере переменная y видна только внутри блока. После условного оператора if переменной не существует. Чтобы переменная была видна внутри блока и после выхода из него, необходимо поместить объявление переменной перед блоком:

// Правильно
int y;
if (x == 10) {  // Какое-то условие
  y = 5;
}

Листинги на странице http://unicross.narod.ru/cpp/
else {
    y = 25;
}

Если обращение к переменной внутри функции или блока производится до объявления одноименной локальной переменной, то до объявления будет использоваться глобальная переменная, а после объявления — локальная переменная (листинг 2.6). Если глобальной переменной с таким названием не существует, то будет ошибка при компиляции.

**Листинг 2.6. Обращение к переменной до объявления внутри функции**

```cpp
#include <iostream>

int x = 10; // Глобальная переменная

int main() {
    // Выводится значение глобальной переменной
    std::cout << x << std::endl; // 10
    int x = 5; // Локальная переменная
    // Выводится значение локальной переменной
    std::cout << x << std::endl; // 5
    std::cin.get();
    return 0;
}
```

**Пространства имен**

Пространства имен предназначены для предотвращения конфликтов идентификаторов внутри программы. Например, в вашей программе объявлена функция с названием `sum()`. В один прекрасный момент вам понадобились возможности библиотеки стороннего разработчика, а в этой библиотеке также объявлена функция с названием `sum()`. При подключении библиотеки произойдет конфликт имен. Чем меньше идентификаторов в глобальной области видимости, тем меньше конфликтов будет в программе.

Подобные конфликты возникали и при использовании стандартной библиотеки. Поэтому идентификаторы стандартной библиотеки C++ размещаются в пространстве имен std. Чтобы получить доступ к этим идентификаторам необходимо указать название

Листинги на странице http://unicross.narod.ru/cpp/
пространства имен перед идентификатором. Между названием пространства имен и
идентификатором указывается оператор ::. Пример:

```cpp
std::cout << "Текст, выводимый в окно консоли"; // Объявления идентификаторов
```

Следует учитывать, что заголовочные файлы из стандартной библиотеки, имеющие
расширение h, по-прежнему помещают все идентификаторы в глобальное пространство
имен.

Для объявления пространства имен предназначено ключевое слово namespace. Формат
объявления пространства имен:

```cpp
namespace <Название пространства имен> { // Объявления идентификаторов
    int x = 10;
}
```

Обратите внимание на то, что пространство имен должно быть объявлено в глобальной
области видимости. Пример объявления пространства имен NS и переменной x внутри:

```cpp
namespace NS {
    int x = 10;
}
```

Для доступа к переменной x из программы необходимо перед названием переменной
указать название пространства имен и оператор ::. Пример:

```cpp
std::cout << NS::x << std::endl;
```

Допускается объявление одного пространства имен несколько раз, а также вложение
одного пространства имен в другое. Пример:

```cpp
namespace NS {
    namespace NS2 {
        int y = 30;
    }
}
```

Если необходимо получить доступ к идентификаторам, объявленным во вложенном
пространстве имен, то перед названием идентификатора перечисляются названия
нескольких пространств имен:

```cpp
std::cout << NS::NS2::y << std::endl;
```

Чтобы получить доступ к идентификаторам, объявленным в глобальном пространстве
имен, то перед идентификатором указывается только оператор ::. Пример:

```cpp
std::cout << ::z << std::endl;
```

Листинги на странице http://unicross.narod.ru/cpp/
С помощью ключевого слова `using` можно импортировать все или только определенные идентификаторы из пространства имен в глобальную или локальную область видимости. Импорт всех идентификаторов выглядит так:

```cpp
using namespace <Название пространства имен>;
```

При импортировании отдельных идентификаторов используется следующий формат:

```cpp
using <Название пространства имен>::<Идентификатор>;
```

Пример использования пространств имен приведен в листинге 2.7.

```
Листинг 2.7. Пространства имен

#include <iostream>

namespace NS {
    int x = 10;
    void print() {
        /* Перед x указывать название пространства имен
        * не нужно, так как x находится в той же
        * области видимости
        */
        std::cout << x << std::endl;
    }
}

namespace NS { // Пространство объявлено дважды
    int y = 20;
    // Вложенное пространство имен
    namespace NS2 {
        int z = 30;
    }
}

int main() {
    std::cout << NS::x << std::endl; // 10
    NS::print(); // 10
    // Доступ к идентификаторам вложенного пространства имен
    std::cout << NS2::z << std::endl; // 30
    // Импорт только идентификатора y
}
```

Листинги на странице http://unicross.narod.ru/cpp/
using NS::y;
std::cout << y << std::endl; // 20
// Импорт всех идентификаторов
using namespace NS;
std::cout << x << std::endl; // 10
print(); // 10
std::cin.get();
return 0;
}

В языке C++ существуют также неименованные пространства имен. Идентификаторы, объявленные в таком пространстве имен, видны только в пределах файла. Формат объявления неименованного пространства имен:

```cpp
namespace {
  // Объявления идентификаторов
}
```

Обращение к идентификаторам неименованного пространства имен осуществляется также как к глобальным идентификаторам. Пример использования неименованного пространства имен приведен в листинге 2.8.

Листинг 2.8. Неименованные пространства имен

```cpp
#include <iostream>

// Неименованное пространство имен
// Область видимости ограничена файлом
namespace {
  int x = 10;
}

int main() {
  std::cout << x << std::endl; // 10
  std::cin.get();
  return 0;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
Массивы

Массив — это нумерованный набор переменных одного типа. Переменная в массиве называется элементом, а ее позиция в массиве задается индексом. Все элементы массива располагаются в смежных ячейках памяти. Например, если объявлен массив из трех элементов, имеющих тип long int (занимает 4 байта), то адрес первого элемента массива (в 32-битной операционной системе) будет 0x0041A040, второго — 0x0041A044, а третьего — 0x0041A048. Объем памяти (в байтах), занимаемый массивом, определяется так:

<Объем памяти> = sizeof (<Тип>) * <Количество элементов>

Объявление массива выглядит следующим образом:

<Tип> <Переменная>[<Количество элементов>];

Пример объявления массива из трех элементов, имеющих тип long int:

long arr[3];

При объявлении элементам массива можно присвоить начальные значения. Для этого после объявления указывается оператор =, а далее значения через запятую внутри фигурных скобок. После закрывающей фигурной скобки обязательно указывается точка с запятой. Пример инициализации массива:

long arr[3] = {10, 20, 30};

Количество значений внутри фигурных скобок может быть меньше количества элементов массива. В этом случае значения присваиваются соответствующим элементам с начала массива. Пример:

long arr[3] = {10, 20};

В этом примере первому элементу массива присваивается значение 10, второму — значение 20, а третьему элементу будет присвоено значение 0.

Если при объявлении массива указываются начальные значения, то количество элементов внутри квадратных скобок можно не указывать. Размер будет соответствовать количеству значений внутри фигурных скобок. Пример:

long arr[] = {10, 20, 30};

Если при объявлении массива начальные значения не указаны, то:

→ элементам глобальных массивов автоматически присваивается значение 0;
→ элементы локальных массивов будут содержать произвольные значения, так называемый "мусор".

Листинги на странице http://unicross.narod.ru/cpp/
Обращение к элементам массива осуществляется с помощью квадратных скобок, в которых указывается индекс элемента. Обратите внимание на то, что нумерация элементов массива начинается с 0, а не с 1, поэтому первый элемент имеет индекс 0. С помощью индексов можно присвоить начальные значения элементам массива уже после объявления:

```cpp
long arr[3];
arr[0] = 10; // Первый элемент имеет индекс 0!!!
arr[1] = 20; // Второй элемент
arr[2] = 30; // Третий элемент
```

Следует учитывать, что проверка выхода указанного индекса за пределы диапазона на этапе компиляции не производится. Таким образом, можно перезаписать значение в смежной ячейке памяти и тем самым нарушить работоспособность программы или даже повредить операционную систему. Помните, что контроль корректности индекса входит в обязанности программиста.

После определения массива выделяется необходимый размер памяти, а в переменной сохраняется адрес первого элемента массива. При указании индекса внутри квадратных скобок производится вычисление адреса соответствующего элемента массива. Зная адрес элемента массива можно получить значение или перезаписать его. Иными словами, с элементами массива можно производить такие же операции, как и с обычными переменными. Пример:

```cpp
long arr[3] = {10, 20, 30};
long x = 0;
x = arr[1] + 12;
arr[2] = x - arr[2];
std::cout << x << std::endl; // 32
std::cout << arr[2] << std::endl; // 2
```

Массивы в языке C++ могут быть многомерными. Объявление многомерного массива имеет следующий формат:

```
<Tип> <Переменная>[<Количество элементов1>][<Количество элементов2>][<Количество элементовN>];
```

На практике наиболее часто используются двухмерные массивы, позволяющие хранить значения ячеек таблицы, содержащей определенное количество строк и столбцов. Объявление двухмерного массива выглядит так:

```
<Tип> <Переменная>[<Количество строк>][<Количество столбцов>];
```

Пример объявления двухмерного массива, содержащего две строки и четыре столбца:

```cpp
int arr[2][4]; // Две строки из 4-х элементов каждая
```

Листинги на странице http://unicross.narod.ru/cpp/
Все элементы двухмерного массива располагаются в памяти друг за другом. Вначале элементы первой строки, затем второй и т. д. При инициализации двухмерного массива элементы указываются внутри фигурных скобок через запятую. Пример:

```cpp
int arr[2][4] = {
    1, 2, 3, 4,
    5, 6, 7, 8
};
```

Чтобы сделать процесс инициализации наглядным мы расположили элементы на отдельных строках. Количество элементов на строке совпадает с количеством столбцов в массиве. Если элементы одной строки разместить внутри фигурных скобок, то процесс станет еще нагляднее:

```cpp
int arr[2][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8}
};
```

Если при объявлении производится инициализация, то количество строк можно не указывать, оно будет определено автоматически:

```cpp
int arr[][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8}
};
```

Получить или задать значение элемента можно указав два индекса (не забывайте, что нумерация начинается с нуля):

```cpp
std::cout << arr[0][0] << std::endl; // 1
std::cout << arr[1][0] << std::endl; // 5
std::cout << arr[1][3] << std::endl; // 8
```

**Строки**

Строка является массивом символов, последний элемент которого содержит нулевой символ (\0). Обратите внимание на то, что нулевой символ (нулевой байт) не имеет никакого отношения к числу 0. Коды этих символов разные. Такие строки достались в "наследство" от языка C, поэтому их часто называют C-строками. В языке C++ помимо C-строк можно также использовать экземпляры класса string. Класс string мы рассмотрим немного позже.

Объявлена С-строка также как и массив элементов:

```c
char str[7];
```

При инициализации можно перечислить символы внутри фигурных скобок:

```c
char str[7] = {'S', 't', 'r', 'i', 'n', 'g', '\0'};
```

или указать строку внутри двойных кавычек:

```c
char str[7] = "String";
```

При использовании двойных кавычек следует учтывать, что длина строки на один символ больше, так как в конец будет автоматически вставлен нулевой символ. Если это не предусмотреть и объявить массив из шести элементов, вместо семи, то при компиляции будет выведено сообщение об ошибке.

Если размер массива при объявлении не указать, то он будет определен автоматически в соответствии с длиной строки:

```c
char str[] = "String";
```

Обратите внимание на то, что присваивать строку в двойных кавычках можно только при инициализации. Попытка присвоить строку позже приведет к ошибке:

```c
char str[7];
str = "String"; // Ошибка!!!
```

Внутри строки в двойных кавычках можно указывать специальные символы (например, \n, \t и др.), которые мы уже рассматривали в разделе "Инициализация переменных". Если внутри строки встречается кавычка, то ее необходимо экранировать с помощью обратного слеша:

```c
char str[] = "Третья "Кино\n"n";
```

Объявить массив строк можно следующим образом:

```c
char str[][20] = {"String1", "String2", "String3"};
```

```c
std::cout << str[0] << std::endl; // String1
std::cout << str[1] << std::endl; // String2
std::cout << str[2] << std::endl; // String3
```

**Указатели**

Указатель — это переменная, которая предназначена для хранения адреса другой переменной. В языке C++ указатели часто используются в следующих случаях:

- для управления динамической памятью;
- чтобы иметь возможность изменить значение переменной внутри функции;

Листинги на странице http://unicross.narod.ru/cpp/
для эффективной работы с массивами и др.

Объявление указателя имеет следующий формат:
<Tип> *<Переменная>;

Пример объявления указателя на тип int:
int *p;
Очень часто символ * указывается после типа, а не перед именем переменной:
int* p;
С точки зрения компилятора эти два объявления ничем не отличаются. Однако следует учитывать, что при объявлении нескольких переменных в одной инструкции, символ * относится к переменной, перед которой он указан, а не к типу данных. Например, следующая инструкция объявляет указатель и переменную, а не два указателя:
int* p, x; // Переменная x указателем не является!!!
Поэтому более логично указывать символ * перед именем переменной:
int *p, x;
Чтобы указателю присвоить адрес переменной, необходимо при присваивании значения перед названием переменной указать оператор &. Типы данных переменной и указателя должны совпадать. Это нужно, чтобы при адресной арифметике был известен размер данных. Пример присвоения адреса:
int *p, x = 10;
p = &x;
Чтобы получить или изменить значение, расположенное по адресу на который ссылается указатель, необходимо выполнить операцию разыменования указателя. Для этого перед названием переменной указывается оператор *. Пример:
std::cout << *p << std::endl; // 10
*p = *p + 20;
std::cout << *p << std::endl; // 30
Основные операции с указателями показаны в листинге 2.9.

<table>
<thead>
<tr>
<th>Листинг 2.9. Указатели</th>
</tr>
</thead>
</table>

```cpp
#include <iostream>

int x = 10;
int *p = 0; // Нулевой указатель
```

Листинги на странице http://unicross.narod.ru/cpp/
int main() {
   // Присваивание указателю адреса переменной x
   p = &x;
   // Вывод адреса
   std::cout << p << std::endl; // Например: 0041A01C
   // Вывод значения
   std::cout << *p << std::endl; // 10
   std::cin.get();
   return 0;
}

В этом примере при объявлении указателя ему присваивается значение 0. В данном случае это можно было и не делать, так как глобальные и статические локальные указатели автоматически получают значение 0. Однако указатели, которые объявлены в локальной области видимости, будут иметь произвольное значение. Если попытаться записать какое-либо значение через такой указатель, то можно повредить операционную систему. Поэтому согласно соглашению, указатели, которые не на что не указывают, должны иметь значение 0. Указатель, которому присвоено значение 0, называется нулевым указателем. Вместо числового значения 0 можно указать макрос NULL, который определен в заголовочном файле stdio.h. Определение макроса выглядит так:

#define NULL 0

Пример использования макроса:

int *p = NULL; // Эквивалентно int *p = 0;

Значение одного указателя можно присвоить другому указателю. При этом важно учитывать, что типы указателей должны совпадать. Пример:

int *p1 = 0, *p2 = 0, x = 10;
p1 = &x;
p2 = p1; // Копирование адреса переменной x
std::cout << *p2 << std::endl; // 10
*p2 = 40; // Изменение значения в переменной x
std::cout << *p1 << std::endl; // 40

В этом примере мы просто скопировали адрес переменной x из одного указателя в другой. Помимо копирования адреса можно создать указатель на указатель. Для этого при объявлении перед названием переменной указываются два оператора *:

Листинги на странице http://unicross.narod.ru/cpp/
int **p = 0;
Чтобы получить адрес указателя используется оператор &., а для получения значения переменной, на которую ссылается указатель, применяются два оператора * . Пример:

int *p1 = 0, **p2 = 0, x = 10;
p1 = &x;
p2 = &p1; // Указатель на указатель
std::cout << **p2 << std::endl; // 10
**p2 = 40; // Изменение значения в переменной x
std::cout << *p1 << std::endl; // 40

Количество вложений указателей неограничено. При каждом вложении указывается дополнительная звездочка:

int *p1 = 0, **p2 = 0, ***p3 = 0, x = 10;
p1 = &x;
p2 = &p1;
p3 = &p2;
std::cout << ***p3 << std::endl; // 10
***p3 = 40; // Изменение значения в переменной x
std::cout << **p2 << std::endl; // 40

Подобный синтаксис трудно понять и очень просто сделать ошибку, поэтому обычно ограничиваются использованием указателя на указатель.

При инициализации указателя ему можно присвоить не только числовое значение, но и строку. Пример:

char *str = "String";
std::cout << str << std::endl; // String

Указатели можно сохранять в массиве. При объявлении массива указателей используется следующий синтаксис:

<Tип> *<Название массива>[<Количество элементов>];

Пример использования массива указателей:

int *p[3]; // Массив указателей из трех элементов
int x = 10, y = 20, z = 30;
p[0] = &x;
p[1] = &y;
p[2] = &z;
std::cout << *p[0] << std::endl; // 10

Листинги на странице http://unicross.narod.ru/cpp/
```cpp
std::cout << *p[1] << std::endl; // 20
std::cout << *p[2] << std::endl; // 30

Опъявлявание массива строк и вывод значений выглядит так:

const char *str[] = {"String1", "String2", "String3"};
std::cout << str[0] << std::endl; // String1
std::cout << str[1] << std::endl; // String2
std::cout << str[2] << std::endl; // String3

Указатели очень часто используются для обращения к элементам массива, так как адресная арифметика выполняется эффективнее, чем доступ по индексу. В качестве примера создадим массив из трех элементов, а затем выведем значения (листинг 2.10).

### Листинг 2.10. Перебор элементов массива

```cpp
#include <iostream>

int main() {
    const short SIZE = 3;
    int i, *p = 0, arr[SIZE] = {10, 20, 30};
    // Устанавливаем указатель на первый элемент массива
    p = arr; // Оператор & не укажется
    for (i=0; i<SIZE; ++i) {
        std::cout << *p << std::endl;
        ++p; // Перемещаем указатель на следующий элемент
    }
    p = arr; // Восстанавливаем положение указателя
    // Выполняем какие-либо инструкции
    std::cin.get();
    return 0;
}
```

В первой строке внутри функции main() объявляется константа SIZE, в которой сохраняется количество элементов в массиве. Если массив используется часто, то лучше сохранить его размер как константу, так как количество элементов нужно будет указывать при каждом переборе массива. Если в каждом цикле устанавливать конкретное число, то при изменении размера массива придется вручную вносить изменения во всех циклах. При объявлении константы достаточно будет изменить ее значение один раз при инициализации.

В следующей строке объявляется переменная i, предназначенная для использования внутри цикла for. Эту переменную можно объявить в первом параметре цикла. В этом случае она будет видна только внутри цикла for. Пример объявления:

```c
for (int i=0; i<SIZE; ++i) { /* инструкции */ }
```

Далее объявляется указатель на тип int и массив. Количество элементов массива задается константой SIZE. При объявлении массив инициализируется начальными значениями.

После объявления переменных указателю присваивается адрес первого элемента массива. Обратите внимание на то, что перед названием массива не указывается оператор &, так как название переменной содержит адрес первого элемента. Если использовать оператор &, то необходимо дополнительно указать индекс внутри квадратных скобок:

```c
p = &arr[0]; // эквивалентно: p = arr;
```

Для перебора элементов массива используется цикл for. В первом параметре цикла задается начальное значение (i=0), во втором — условие (i<SIZE), а в третьем — приращение на единицу (++i) на каждой итерации цикла. Инструкции внутри цикла будут выполняться пока условие является истинным (значение переменной i меньше количества элементов массива).

Внутри цикла выводится значение элемента на который ссылается указатель, а затем значение указателя увеличивается на единицу (++p;). Обратите внимание на то, что изменяется адрес, а не значение элемента массива. При увеличении значения указателя используются правила адресной арифметики, а не правила обычной арифметики. Увеличение значения указателя на единицу означает, что значение будет увеличено на размер типа. Например, если тип int занимает 4 байта, то при увеличении значения на единицу указатель вместо адреса 0x0012FF30 будет содержать адрес 0x0012FF34. Значение увеличилось на 4, а не на 1. В нашем примере вместо двух инструкций внутри цикла можно использовать одну:

```c
std::cout << *p++ << std::endl;
```

Выражение p++ возвращает текущий адрес, а затем увеличивает его на единицу. Символ * позволяет получить доступ к значению элемента по указанному адресу. Последовательность выполнения соответствует следующей расстановке скобок:

```c
std::cout << *(p++) << std::endl;
```

Если скобки расставить так:

```c
std::cout << (*p)++ << std::endl;
```

Листинги на странице http://unicross.narod.ru/cpp/
то, вначале будет получен доступ к элементу массива и выведено его текущее значение, а затем производится увеличение значения элемента массива. Перемещение указателя на следующий элемент не производится.

Получить доступ к элементу массива можно несколькими способами. Первый способ заключается в указании индекса внутри квадратных скобок. Во втором способе используется адресная арифметика совместно с разыменованием указателя. В третьем способе внутри квадратных скобок указывается название массива, а перед квадратными скобками — индекс элемента. Этот способ может показаться странным. Однако, если учесть, что выражение \( 1[\text{arr}] \) воспринимается компилятором как \( *(1 + \text{arr}) \), то все встанет на свои места. Таким образом, все эти инструкции являются эквивалентными:

```cpp
int arr[3] = {10, 20, 30};
std::cout << arr[1] << std::endl; // 20
std::cout << *(arr + 1) << std::endl; // 20
std::cout << *(1 + arr) << std::endl; // 20
std::cout << 1[arr] << std::endl; // 20
```

С указателем можно выполнять следующие арифметические и логические операции:

- прибавлять целое число. Число умножается на размер базового типа указателя, а затем результат прибавляется к адресу;
- вычитать целое число. Вывести значения элементов массива в обратном порядке можно так:

```cpp
p = &arr[2]; // Устанавливаем указатель на последний элемент
for (i=SIZE-1; i>=0; --i) {
    std::cout << *p-- << std::endl;
}
```
- вычитать один указатель из другого. Это позволяет получить количество элементов базового типа между двумя указателями;
- сравнивать указатели между собой.

При использовании ключевого слова `const` применительно к указателям важно учитывать местоположение ключевого слова `const`. Например, следующие объявления не эквивалентны:

```cpp
const char *p = str;
char const *p = str;
char * const p = str;
```

Листинги на странице http://unicross.narod.ru/cpp/
const char * const p = str;
Первые два объявления являются эквивалентными. В этом случае изменить значение,
на которое ссылается указатель, нельзя, но указателю можно присвоить другой адрес:
char str1[] = "String", str2[] = "New";
const char *p = str1;
p = str2; // Нормально
p[0] = 's'; // Ошибка
При третьем объявлении изменить значение, на которое ссылается указатель, можно, но
указателю нельзя присвоить другой адрес:
char str1[] = "String", str2[] = "New";
char * const p = str1;
p = str2; // Ошибка
p[0] = 's'; // Нормально
Четвертое объявление запрещает изменение значения, на которое ссылается указатель,
и присвоение другого адреса:
char str1[] = "String", str2[] = "New";
const char * const p = str1;
p = str2; // Ошибка
p[0] = 's'; // Ошибка

Ссылки

Ссылка — это переменная, которая является псевдонимом другой переменной. Присвоить значение ссылке можно только при ее объявлении. Тип ссылки должен совпадать с типом переменной. Объявление выглядит следующим образом:
<Tип> &<Название ссылки> = <Переменная>;
После создания ссылки ее название можно использовать в выражениях вместо названия переменной. Пример использования независимых ссылок приведен в листинге 2.11.

Листинг 2.11. Независимые ссылки

#include <iostream>

int main() {
    int x = 10;
    int &ref = x; // Объявление ссылки
ref = 20;
std::cout << x << std::endl;    // 20
x = 30;
std::cout << ref << std::endl;   // 30
std::cin.get();
return 0;
}

Особого смысла в применении независимых ссылок нет. Все удобство использования ссылок проявляется при передаче параметров в функцию. Прежде чем рассматривать передачу параметров с помощью ссылок вначале необходимо продемонстрировать использование указателей (листинг 2.12).

**Листинг 2.12. Передача параметров по ссылке**

```
#include <iostream>

void func(int *x);

int main() {
    int y = 10;
    func(&y);               // Передаем адрес, а не значение
    std::cout << y << std::endl; // 20
    std::cin.get();
    return 0;
}
void func(int *x) {
    *x = *x * 2;
}
```

В этом примере объявлена функция `func()`, внутри которой производится увеличение значения переменной в два раза. Если использовать передачу по значению (применяется по умолчанию), то создается копия значения и все операции производятся с этой копией. Так как локальные переменные видны только внутри тела функции, после завершения выполнения функции копия удаляется. Любые изменения значения копии не затронут значения оригинала. Чтобы изменить значение самой переменной используется передача значения по ссылке. В этом случае в функцию передается адрес переменной:

Листинги на странице http://unicross.narod.ru/cpp/
Внутри функции адрес присваивается указателю. Используя операцию разыменования указателя можно изменить значение самой переменной, а не значение копии. Достигается это с помощью следующей инструкции:

```
*x = *x * 2;
```

При использовании ссылки оператор `&` указывается перед параметром в объявлении функции. В этом случае при вызове функции передается не адрес, а значение. Таким образом внутри функции мы работаем с псевдонимом переменной, а не с указателем, поэтому выполнять разыменование указателя не нужно. Пример использования ссылок приведен в листинге 2.13.

```cpp
#include <iostream>

void func(int &x);

int main() {
    int y = 10;
    func(y); // Передаем значение, а не адрес
    std::cout << y << std::endl; // 20
    std::cin.get();
    return 0;
}

void func(int &x) {
    x = x * 2;
}
```

Согласитесь, что инструкция `x = x * 2;` выглядит более привлекательно, чем инструкция `*x = *x * 2;`.

**Динамическое выделение памяти**

Как вы уже знаете, при объявлении переменной необходимо указать тип данных, а для массива дополнительно задать точное количество элементов. На основе этой информации при запуске программы автоматически выделяется необходимый объем памяти. После завершения программы память автоматически освобождается. Иными словами, объем памяти необходимо знать до выполнения программы. Во время

Листинги на странице http://unicros.sarod.ru/cpp/
выполнении программы создать новую переменную или увеличить размер существующего массива нельзя.

Чтобы произвести увеличение массива во время выполнения программы необходимо выделить достаточный объем памяти с помощью оператора new, перенести существующие элементы, а лишь затем добавить новые элементы. Управление динамической памятью полностью лежит на плечах программиста, поэтому после завершения работы с памятью необходимо самим возвратить память операционной системе с помощью оператора delete. Если память не возвратить операционной системе, то участок памяти станет недоступным для дальнейшего использования. Подобные ситуации приводят к утечке памяти. Сделать эти участи опять доступными можно только после перезагрузки компьютера.

Для выделения памяти под один объект предназначен следующий синтаксис:

```c
<Указатель> = new <Тип данных>(){<Начальное значение>};
```

Оператор new выделяет объем памяти, необходимый для хранения значения указанного типа, записывает в эту память начальное значение (если оно задано) и возвращает адрес. Работать в дальнейшем с этим участком памяти можно с помощью упаковщика. Пример выделения памяти:

```c
int *p = new int; // Без начального значения
int *p = new int(10); // С начальным значением
```

При выделении памяти может возникнуть ситуация нехватки памяти. В ранних версиях C++ в этом случае возвращался нулевой указатель. Согласно новой версии стандарта в случае ошибки оператор new должен возбуждать исключение bad_alloc (объект исключения определен в файле new). Обработать это исключение можно с помощью конструкции try...catch. Пример выделения памяти с обработкой исключения:

```c
#include <new>

// ... Фрагмент опущен ...
int *p = 0; // Создаем указатель
try {
    p = new int; // Выделяем память
}
catch (std::bad_alloc err) {
    // Обработка исключения
}
```

Выделение памяти производится внутри блока try. Если при этом возникнет исключение bad_alloc, то управление будет передано в блок catch. После
выполнения инструкций в блоке catch управление передается инструкции, расположененной сразу после блока. Иными словами, компилятор считает, что вы обработали исключение и можно продолжить выполнение программы. Следует учитывать, что пользоваться указателем после обработки нельзя, поэтому внутри блока catch обычно выводят сообщение об ошибке и завершают выполнение программы. Если исключение не обработано, то программа аварийно завершается. Если исключение не возникло, то инструкции внутри блока catch не выполняются.

Обратите внимание на то, что объявление указателя производится вне блока try. Если объявление разместить внутри блока, то область видимости переменной будет ограничена этим блоком. После выхода из блока переменная автоматически уничтожается, а выделенная память операционной системе не возвращается. Поэтому, объявление указателя должно находиться перед блоком, а не внутри него.

Возвратить ранее выделенную память операционной системе позволяет оператор delete. Оператор имеет следующий формат:

delte <Указатель>;

После использования оператора delete указатель по-прежнему будет содержать прежний адрес. Поэтому после использования оператора delete указатель принято обнулять. Пример выделения памяти под один объект приведен в листинге 2.14.

### Листинг 2.14. Динамическое выделение памяти под один объект

```cpp
#include <iostream>
#include <cstdlib>    // Для exit()
#include <new>        // Для bad_alloc

int main() {
    int *p = 0;       // Создаем указатель
    try {
        p = new int;  // Выделяем память
    }
    catch (std::bad_alloc err) {
        std::cout << "Error" << std::endl;
        std::exit(1);    // Выходим при ошибке
    }
    *p = 20;           // Пользуемся памятью
    std::cout << *p << std::endl;  // 20
}
```

Листинги на странице http://unicross.narod.ru/cpp/
Выделение памяти под массив производится следующим образом:
\[
<\text{Указатель}> = \text{new} <\text{Тип данных}>[<\text{Количество элементов}>];
\]
Освободить выделенную память можно так:
\[
\text{delete} \ [\ <\text{Указатель}>];
\]
Обратите внимание на то, что при освобождении памяти количество элементов не указывается. Пример выделения памяти под массив приведен в листинге 2.15.

**Листинг 2.15. Динамическое выделение памяти под массив**

```cpp
#include <iostream>
#include <cstdlib>  // Для exit()
#include <new>      // Для bad_alloc

int main() {
    const short SIZE = 3;  // Размер массива
    int *p = 0;            // Создаем указатель
    try {
        p = new int[SIZE];  // Выделяем память
    }
    catch (std::bad_alloc err) {
        std::cout << "Error" << std::endl;
        std::exit(1);        // Выходим при ошибке
    }
    p[0] = 10;             // Пользуемся памятью
    p[1] = 20;
    p[2] = 30;
    for (int i=0; i< SIZE; ++i) {
        std::cout << p[i] << std::endl;
    }
    delete [] p;          // Возвращаем память
    p = 0;                // Обнуляем указатель
}
```

Листинги на странице http://unicross.narod.ru/cpp/
В ранних версиях C++ при нехватке памяти возвращался нулевой указатель. Такая же
ситуация возникает в языке C при использовании функции malloc(). Чтобы оператор
new, соответствующий современному стандарту, возвращал нулевой указатель, а не
genерировал исключение используется следующий синтаксис:

<Указатель> = new (std::nothrow) <Тип данных>({<Начальное значение>});
<Указатель> = new (std::nothrow) <Тип данных>[<Количество элементов>];
Для использования nothrow требуется подключить файл new. После выделения памяти
следует проверить указатель на отсутствие нулевого значения. Пример выделения
памяти без генерации исключения приведен в листинге 2.16.

Листинг 2.16. Динамическое выделение памяти без генерации исключения

```cpp
#include <iostream>
#include <cstdlib> // Для exit()
#include <new> // Для nothrow

int main() {
    int *p = new (std::nothrow) int; // Выделяем память
    if (!p) { // Проверяем на корректность
        std::cout << "Error" << std::endl;
        std::exit(1); // Выходим при ошибке
    }
    *p = 20; // Пользуемся памятью
    std::cout << *p << std::endl; // 20
    delete p; // Возвращаем память
    p = 0; // Обнуляем указатель
    std::cin.get();
    return 0;
}
```
Язык C++ поддерживает также функции malloc() и free() из языка C. Функция
malloc() предназначена для динамического выделения памяти, а функция free() —
для освобождения ранее выделенной памяти. Для использования функций необходимо
подключить файл cstdlib (или stdlib.h). Прототипы функций:

Листинги на странице http://unicross.narod.ru/cpp/
#include <cstdlib>
void *malloc(size_t Size);
void free(void *Memory);

Функция malloc() принимает в качестве параметра размер памяти в байтах и возвращает указатель, имеющий тип void *. Если память выделить не удалось, то функция возвращает нулевой указатель. В языке C++ перед присвоением значения указателю необходимо выполнить явное приведение к используемому типу. Кроме того, чтобы программа была машиннезависимой следует применять оператор sizeof для вычисления размера памяти, требуемого для определенного типа. Пример:

```c
int *p = (int *) std::malloc(sizeof(int));
```

Функция free() принимает в качестве параметра указатель на ранее выделенную память. Важно учитывать, что указатель должен быть корректным. Пример использования функций malloc() и free() приведен в листинге 2.17.

Листинг 2.17. Динамическое выделение памяти с помощью функций malloc() и free()

```c
#include <iostream>
#include <cstdlib> // Для exit(), malloc() и free()

int main() {
    // Выделяем память
    int *p = (int *) std::malloc(sizeof(int));
    if (!p) { // Проверяем на корректность
        std::cout << "Error" << std::endl;
        std::exit(1); // Выходим при ошибке
    }
    *p = 20; // Пользуемся памятью
    std::cout << *p << std::endl; // 20
    std::free((void *)p); // Возвращаем память
    p = 0; // Обнуляем указатель
    std::cin.get();
    return 0;
}
```

Следует учитывать, что функции malloc() и free() применяются в языке C. Хотя их можно использовать и в языке C++, тем не менее стоит отдать предпочтение оператору new и явной обработке исключения. Описание функций malloc() и free(), а также
ключевого слова noexcept, приведено в этой книге лишь для того, чтобы вы могли разобраться в чужом коде. Необходимо также заметить, что в одной программе не стоит одновременно использовать оператор new и функцию malloc().

Структуры

Структура — это совокупность переменных (называемых членами, элементами или полями), объединенных под одним именем. Объявление структуры выглядит следующим образом:

```c
struct [<Название структуры>] {  
   <Тип данных> <Название поля1>;  
   ...  
   <Тип данных> <Название поляN>;  
} [<Объявления переменных через запятую>];
```

Допустимо не задавать название структуры, если после закрывающей фигурной скобки указано объявление переменной. Точка с запятой в конце объявления структуры является обязательной.

Объявление структуры только описывает новый тип данных, а не определяет переменную, поэтому память под нее не выделяется. Чтобы объявить переменную, ее название указывается после закрывающей фигурной скобки при объявлении структуры или отдельно, используя название структуры в качестве типа данных:

```c
[struct ]<Название структуры> <Названия переменных через запятую>;
```

В языке С++ ключевое слово struct при объявлении переменной можно не указывать. После объявления переменной компилятор выделяет необходимый размер памяти. Для получения размера структуры внутри программы следует использовать оператор sizeof:

```c
<Размер> = sizeof <Переменная>;
<Размер> = sizeof (<Название структуры>);
```

Пример одновременного объявления структуры и переменной:

```c
struct Point {  
   int x;  
   int y;  
} point1;
```

Пример отдельного объявления переменной:

```c
Point point2;
```

Листинги на странице http://unicross.narod.ru/cpp/
Присвоить или получить значение поля можно с помощью точечной нотации:

```cpp
<Переменная>.<Название поля> = <Значение>;
<Значение> = <Переменная>.<Название поля>;
```

Пример:

```cpp
point1.x = 0;
point1.y = 100;
```

Одну структуру можно присвоить другой структуре с помощью оператора `=`. В этом случае копируются значения всех полей структуры. Пример:

```cpp
point2 = point1;
```

Структуры можно вкладывать. При обращении к полю вложенной структуры дополнительно указывается название структуры родителя. В качестве примера объявим структуру `Point` (точка), а затем используем ее для описания координат прямоугольника (листинг 2.18). Структуру, описывающую прямоугольник, объявим без названия.

### Листинг 2.18. Использование вложенных структур

```cpp
#include <iostream>

struct Point { // Объявление именованной структуры
    int x;
    int y;
};

struct { // Объявление структуры без названия
    Point top_left;
    Point bottom_right;
} rect;

int main() {
    rect.top_left.x = 0;
    rect.top_left.y = 0;
    rect.bottom_right.x = 100;
    rect.bottom_right.y = 100;
    std::cout << rect.top_left.x << " " << rect.top_left.y << std::endl
              << rect.bottom_right.x << " " << std::endl;
    Листинги на странице http://unicross.narod.ru/cpp/
<< rect.bottom_right.y << std::endl;
    std::cin.get();
    return 0;
}

Адрес структуры можно сохранить в указателе. Объявление указателя на структуру
производится также как и на любой другой тип данных. Для получения адреса
структуры используется оператор &, а для доступа к полю структуры вместо точки
применяется оператор ->. Пример использования указателя на структуру приведен в
листинге 2.19.

Листинг 2.19. Использование указателя на структуру

#include <iostream>

struct Point {   // Объявление структуры и переменной
    int x;
    int y;
} point1;
Point *p = &point1;   // Объявление указателя

int main() {
    p->x = 10;
    p->y = 20;
    std::cout << p->x << " " << p->y << std::endl;
    std::cin.get();
    return 0;
}

В языке C++ структуры имеют функционал, аналогичный классам. Единственное
отличие состоит в том, что доступ к членам структуры является открытым, а доступ к
членам класса по умолчанию является закрытым. И в том и другом случае поведение по
умолчанию можно изменить. Например, объявление класса Point с открытым доступом
k членам выглядит так:

class Point { // Объявление класса и переменной
    public:
        int x;
        int y;

Листинги на странице http://unicross.narod.ru/cpp/
} point;

Как видно из примера, объявление класса производится также как и объявление структуры, только вместо ключевого слова struct используется слово class. Отличие только в указании ключевого слова public перед объявлением членов класса, которое открывает доступ к членам класса извне. Как уже говорилось, по умолчанию доступ к членам класса извне закрыт. Классы мы рассмотрим подробно при изучении объектно-ориентированного программирования.

**Битовые поля**

Логический тип данных bool может содержать только значения true (соответствует числу 1) или false (соответствует числу 0). Эти значения помещаются в один бит, однако тип данных bool занимает в памяти целый байт (восемь бит). Остальные семь бит не содержат значений. Язык C++ поддерживает битовые поля, которые предоставляют доступ к отдельным битам, позволяя тем самым хранить в одной переменной несколько значений, занимающих указанное количество бит. Тем не менее следует учитывать, что минимальный размер битового поля будет соответствовать типу int (в 32-битной операционной системе занимает 4 байта). Объявление битового поля имеет следующий формат:

```cpp
struct [/<Название битового поля>/] {  
    <Тип данных> [/<Название поля1>]:<Длина в битах>;  
    ...
    <Тип данных> [/<Название поляN>]:<Длина в битах>;
} [/<Объявления переменных через запятую>/];
```

В одной структуре можно использовать одновременно битовые поля и обычные поля. Обратите внимание на то, что название битового поля можно не указывать, кроме того, если длина поля составляет один бит, то перед названием поля следует указать ключевое слово unsigned. Пример объявления битового поля и переменной:

```cpp
struct Status {  
    unsigned :3;  
    unsigned a:1;  
    unsigned b:1;  
    unsigned c:1;
} status;
```

Доступ к полю осуществляется также как и к полю структуры:

```cpp
status.a = 1;
```

Листинги на странице http://unicross.narod.ru/cpp/
status.b = 0;
status.c = 1;

Объединения

Объединение — это область памяти, используемая для хранения данных разных типов. В один момент времени в этой области могут храниться данные только одного типа. Размер объединения будет соответствовать размеру более сложного типа данных. Например, если внутри объединения определены переменные, имеющие типы int, float и double, то размер объединения будет соответствовать размеру типа double. Объявление объединения имеет следующий формат:

```plaintext
union [ <Название объединения> ] {
    <Тип данных> <Название члена1>;
    ...
    <Тип данных> <Название членаN>;
} [ <Объявления переменных через запятую>] ;
```

Точка с запятой в конце объявления является обязательной. Объявление только описывает новый тип данных, а не определяет переменную, поэтому память под нее не выделяется. Чтобы объявить переменную, ее название указывается после закрывающей фигурной скобки при объявлении объединения или отдельно, используя название объединения в качестве типа данных:

```plaintext
[ union ] <Название объединения> <Название переменных через запятую> ;
```

В языке C++ ключевое слово union при объявлении переменной можно не указывать. После объявления переменной компилятор выделяет необходимый размер памяти. Пример объявления объединения и переменной:

```plaintext
union Uni {
    int x;
    float y;
    double z;
} union1;
```

Приписать или получить значение можно с помощью точечной нотации:

```plaintext
<Переменная>.<Название члена> = <Значение> ;
<Значение> = <Переменная>.<Название члена> ;
```

Пример:

```plaintext
union1.x = 152;
```

Листинги на странице http://unicross.narod.ru/cpp/
std::cout << union1.x << std::endl; // 152
В языке C++ объединения могут содержать функции, а также конструктор и деструктор. Объявления функций размещаются внутри объявления объединения, а определения функций — вне объявления. Перед названием функции указывается название объединения и оператор ::. Внутри функции к членам объединения можно обращаться без указания названия объединения. Пример использования объединений приведен в листинге 2.20.

**Листинг 2.20. Объединения**

```cpp
#include <iostream>

union Uni {
    int x;
    double y;
    void print_x();
    void print_y();
};

void Uni::print_x() {
    std::cout << x << std::endl;
}
void Uni::print_y() {
    std::cout << y << std::endl;
}

int main() {
    Uni union1;
    union1.x = 152;
    union1.print_x(); // 152
    union1.y = 1.5e-5;
    union1.print_y(); // 1.5e-005
    std::cin.get();
    return 0;
}
```

Допускается не задавать одновременно название объединения и объявления переменных. В этом случае объединение называется безымянным или анонимным.

Листинги на странице http://unicross.narod.ru/cpp/
Доступ к членам безымянных объединений осуществляется непосредственно по именам членов. Таким образом, названия членов объединений не должны конфликтовать с именами переменных, объявленных в том же пространстве имен. Безымянные объединения расположенные в глобальной области видимости или внутри пространства имен должны объявляться с помощью ключевого слова static. Пример использования безымянных объединений приведен в листинге 2.21.

Листинг 2.21. Безымянные (анонимные) объединения

```cpp
#include <iostream>

static union { // static указывать обязательно
    int x;
    double y;
};

int main() {
    union { // Слово static не указывается
        int a;
        double b;
    };
    x = 12;
    std::cout << x << std::endl; // 12
    y = 11.5;
    std::cout << y << std::endl; // 11.5
    a = 458;
    std::cout << a << std::endl; // 458
    b = 1.7e5;
    std::cout << b << std::endl; // 170000
    std::cin.get();
    return 0;
}
```

Перечисления

Перечисление — это совокупность целочисленных констант, описывающих все допустимые значения переменной. Если переменной присвоить значение не
совпадающее с перечисленными константами, то компилятор выведет сообщение об ошибке. Объявление перечисления имеет следующий формат:

```
enum [ <Название перечисления>] {  
   <Список констант через запятую>  
} [<Объявления переменных через запятую>];
```

Точка с запятой в конце объявления является обязательной. Чтобы объявить переменную, ее название указывается после закрывающей фигурной скобки при объявлении перечисления или отдельно, используя название перечисления в качестве типаданных:

```
[ enum ]<Название перечисления> <Названия переменных через запятую>;
```

В языке C++ ключевое слово `enum` при объявлении переменной можно не указывать. Пример одновременного объявления перечисления и переменной:

```
enum Color {  
   red, blue, green, black  
} color1;
```

Пример отдельного объявления переменной:

```
Color color2;
```

Константам `red`, `blue`, `green` и `black` автоматически присваиваются целочисленные значения, начиная с нуля. Значение каждой последующей константы будет на единицу больше предыдущей. Нумерация производится слева направо. Таким образом, константа `red` будет иметь значение `0`, `blue` — `1`, `green` — `2`, `black` — `3`.

При объявлении перечисления константе можно присвоить другое значение. В этом случае последующая константа будет иметь значение на единицу больше этого другого значения. Пример:

```
enum Color {  
   red=3, blue, green=7, black  
} color1;
```

В этом примере, константа `red` будет иметь значение `3`, а не `0`, `blue` — `4`, `green` — `7`, `black` — `8`. Присвоить значение переменной можно так:

```
color1 = black;
```

Допустимо не задавать одновременно название перечисления и объявление переменной. В этом случае переменные, указанные внутри фигурных скобок, используются как константы. Пример:

```
enum {
```

Листинги на странице http://unicross.narod.ru/cpp/
```cpp
red, blue = 10, green, black
}
std::cout << blue << std::endl; // 10
Пример использования перечисления приведен в листинге 2.22.

Листинг 2.22. Перечисления

#include <iostream>

enum Color { // Объявление перечисления
    red, blue, green=5, black
};

int main() {
    Color color; // Объявление переменной
    color = black;
    std::cout << color << std::endl; // 6
    if (color == black) { // Проверка значения
        std::cout << "color == black" << std::endl;
    }
    std::cin.get();
    return 0;
}

Приведение типов

Как вы уже знаете, при объявлении переменной необходимо указать определенный тип данных. Далее над переменной можно производить операции, предназначенные для этого типа данных. Если в выражении используются переменные, имеющие разный тип данных, то тип результата выражения будет соответствовать наиболее сложному типу. Например, если производится сложение переменной, имеющей тип int, с переменной имеющей тип double, то целое число будет автоматически преобразовано в вещественное. Результатом этого выражения будет значение, имеющее тип double.

При попытке присвоить переменной значение, имеющее несовместимый тип, компилятор выведет сообщение об ошибке (если автоматическое преобразование невозможно) или предупреждающее сообщение (при усечении значения). Например, попытка присвоить целочисленной переменной строку (int x = "string";) приведет

Листинги на странице http://unicross.narod.ru/cpp/
к нейсправной ошибке: "error C2440: инициализация невозможно преобразовать
"const char [7]" в "int"". Попытка присвоить вещественное число целочисленной переменной (int x = 10.5; ) приведет к следующему предупреждающему сообщению: "warning C4244: инициализация: преобразование "double" в "int", возможна потеря данных". В этом случае дробная часть просто отбрасывается, переменной присваивается число 10, а выполнение программы будет продолжено. Подобное сообщение будет выведено даже если после точки стоит нуль. Чтобы избавиться от предупреждающих сообщений необходимо выполнить явное приведение типов. В этом случае компилятор считает, что вы знаете что делаете.

Для приведения типов в языке C++ используются следующие операторы:

- **static_cast** — применяется для стандартного приведения типов. Формат оператора:

  ```cpp
  static_cast< <Тип результата> > (<Выражение>)
  ```

  Например, деление целых чисел возвращает целое число. Дробная часть при этом просто отбрасывается. Чтобы деление целых чисел возвращало вещественное число, необходимо преобразовать одно из целых чисел в вещественное:

  ```cpp
  int x = 10, y = 3;
  std::cout << x / y << std::endl; // 3
  std::cout << static_cast<double> (x) / y
               << std::endl; // 3.33333
  ```

- **reinterpret_cast** — используется для приведения указателя одного типа в абсолютно другой, даже в несовместимый. Формат оператора:

  ```cpp
  reinterpret_cast< <Тип результата> > (<Выражение>)
  ```

- **const_cast** — отменяет действие ключевых слов const и volatile. Формат оператора:

  ```cpp
  const_cast< <Тип результата> > (<Выражение>)
  ```

  Пример приведения константного указателя в обычный внутри функции:

  ```cpp
  void func(const int *x) {
      int *p = const_cast<int *> (x);
      // *x = 50; // Ошибка!!!
      *p = 30; // Теперь можно изменить значение
  }
  ```

  Листинги на странице http://unicross.narod.ru/cpp/
Пример удаления действия ключевого слова `const` у ссылки внутри функции:
```c++
void func(const int &x) {
    // x = 50; // Ошибка!!!
    // Теперь можно изменить значение
    const_cast<int &>(x) = 30;
}
```

> `dynamic_cast` — выполняет приведение типов указателей или ссылок. Применяется для приведения полиморфных типов. Если приведение указателя окончилось неудачей, оператор возвращает нулевой указатель, а если проблема со ссылкой, то генерируется исключение `bad_cast` (объект исключения объявлен в файле `typeinfo`). Формат оператора:

```c++
dynamic_cast< тип результата > ( выражение )
```

Более подробно оператор `dynamic_cast` мы рассмотрим при изучении объектно-ориентированного программирования.

Язык C++ поддерживает также приведение типов, используемое в языке C. Формат приведения:

```c
(<тип результата>) выражение
```

Пример преобразования целого числа в вещественное, имеющее тип `double`:
```c++
int x = 10, y = 3;
std::cout << x / y << std::endl; // 3
std::cout << (double)x / y << std::endl; // 3.33333
```

Кроме того, можно указать выражение внутри круглых скобок после названия типа:
```c++
std::cout << double(10) / 3 << std::endl; // 3.33333
```

Листинги на странице `http://unicross.narod.ru/cpp/`
Программирование на C++
в Visual Studio® 2010 Express

Глава 3.
Операторы и циклы

© Прохоренок Н.А., 2010 г., unicross@mail.ru
Этот DjVu-файл предоставляется КАК ЕСТЬ. Автор не несет никакой
ответственности за прямые или косвенные проблемы, связанные с
использованием данного файла. ВЫ ИСПОЛЬЗУЕТЕ ЕГО НА СВОЙ
СТРАХ И РИСК.
Все названия программных продуктов являются зарегистрированными
торовыми марками соответствующих фирм.

Запрещено:
- перепечатывать всю книгу или отдельные главы без письменного разрешения
Автора;
- декомпилировать данный файл и преобразовывать его в любой другой формат.
Операторы позволяют выполнить определенные действия с данными. Например, операторы присваивания служат для сохранения данных в переменной, математические операторы предназначены для арифметических вычислений, а условные операторы позволяют в зависимости от значения логического выражения выполнить отдельный участок программы или наоборот не выполнять его.

Математические операторы

Производить арифметические вычисления позволяют следующие операторы:

- ➔ + — сложение:
  ```
  std::cout << 10 + 15 << std::endl; // 25
  ```

- ➔ — вычитание:
  ```
  std::cout << 35 - 15 << std::endl; // 20
  ```

- ➔ — унарный минус:
  ```
  int x = 10;
  std::cout << -x << std::endl; // -10
  ```

- ➔ * — умножение:
  ```
  std::cout << 25 * 2 << std::endl; // 50
  ```

- ➔ / — деление. Если производится деление целых чисел, то остаток отбрасывается и возвращается целое число. Деление вещественных чисел производится классическим способом. Если в выражении участвуют вещественное и целое числа, то целое число автоматически преобразуется в вещественное. Пример:
  ```
  std::cout << 10 / 3 << std::endl; // 3
  std::cout << 10.0 / 3.0 << std::endl; // 3.33333
  std::cout << 10.0 / 3 << std::endl; // 3.33333
  ```

- ➔ % — остаток от деления. Обратите внимание на то, что этот оператор нельзя применять к вещественным числам. Пример:
  ```
  std::cout << 10 % 2 << std::endl; // 0 (10 - 10 / 2 * 2)
  std::cout << 10 % 3 << std::endl; // 1 (10 - 10 / 3 * 3)
  std::cout << 10 % 4 << std::endl; // 2 (10 - 10 / 4 * 4)
  std::cout << 10 % 6 << std::endl; // 4 (10 - 10 / 6 * 6)
  ```

Листинги на странице http://unicross.narod.ru/cpp/
++ — оператор инкремента. Увеличивает значение переменной на 1:
\[
\begin{aligned}
int x = 10; \\
++x; \\
std::cout << x << std::endl; // Эквивалентно x = x + 1;
\end{aligned}
\]

-- — оператор декремента. Уменьшает значение переменной на 1:
\[
\begin{aligned}
int x = 10; \\
--x; \\
std::cout << x << std::endl; // x = x - 1;
\end{aligned}
\]

Операторы инкремента и декремента могут использоваться в постфиксной или префиксной формах:

++x; x--; // Постфиксная форма
++x; --x; // Префиксная форма

При постфиксной форме (x++) возвращается значение переменной перед операцией, а при префиксной форме (++x) — вначале производится операция и только потом возвращается значение. Продемонстрируем это на примере (листинг 3.1).

### Листинг 3.1. Постфиксная и префиксная форма

```cpp
#include <iostream>
#include <locale>

int main() {
    int x = 0, y = 0;
    std::setlocale(LC_ALL, "Russian_Russia.1251");
    x = 5;
    y = x++; // y = 5, x = 6
    std::cout << "Постфиксная форма (y = x++):" << std::endl
               << "y = " << y << std::endl
               << "x = " << x << std::endl;
    x = 5;
    y = ++x; // y = 6, x = 6
    std::cout << "Предфиксная форма (y = ++x):" << std::endl
               << "y = " << y << std::endl
               << "x = " << x << std::endl;
}
```

std::cin.get();
return 0;
}

В итоге получим следующий результат:
Постфиксная форма (y = x++;):
y = 5
x = 6
Предфиксная форма (y = ++x;):
y = 6
x = 6

Если операторы инкремента и декремента используются в сложных выражениях, то
понять, каким будет результат выполнения выражения становится сложно. Например,
каким будет значение переменной y после выполнения этих инструкций?
int x = 5, y = 0;
y = ++x + ++x;
Думаете, что число 13 (y = 6 + 7)? После выполнения этих инструкций в VC++ 2010
переменная x будет содержать число 14 (y = 7 + 7;). Вначале вычисляется первое
выражение ++x. Значение переменной x становится равным 6 (5 + 1). Далее
вычисляется второе выражение ++x. Значение переменной x становится равным 7 (6 +
1). В результате инструкция примет следующий вид:
x = 7;
y = x + x;
Так как переменная x содержит значение 7, то вычисляется выражение 7 + 7.
Результатом будет число 14. Следует учитывать, что результат выполнения при
использовании другого компилятора может быть другим, так как поведение
компилятора в этом случае в стандарте языка C++ не определено.
Чтобы облегчить жизнь себе и всем другим программистам, которые будут разбираться в
программе, операторы инкrementa и декрeмена лучше использовать отдельно от
других операторов в префиксной форме. Согласитесь, что после выполнения этих
инструкций, значение переменной y не вызывает никаких сомнений:
int x = 5, y = 0;
++x;
++x;
y = x + x;

Листинги на странице http://unicross.narod.ru/cpp/
Побитовые операторы

Побитовые операторы предназначены для манипуляции отдельными битами. Эти операторы нельзя применять к вещественным числам, логическим значениям и другим более сложным типам. Язык C++ поддерживает следующие побитовые операторы:

- ~ — двоичная инверсия. Значение каждого бита заменяется на противоположное:
  ```cpp
  char x = 100; // 01100100
  x = ~x; // 10011011
  ```

- & — двоичное И:
  ```cpp
  char x = 100; // 01100100
  char y = 75; // 01001011
  char z = x & y; // 01000000
  ```

- | — двоичное ИЛИ:
  ```cpp
  char x = 100; // 01100100
  char y = 75; // 01001011
  char z = x | y; // 01101111
  ```

- ^ — двоичное исключающее ИЛИ:
  ```cpp
  unsigned char x = 100; // 01100100
  unsigned char y = 250; // 11110100
  unsigned char z = x ^ y; // 10011110
  ```

- << — сдвиг влево — сдвигает двоичное представление числа влево на один или более разрядов и заполняет разряды справа нулями:
  ```cpp
  char x = 100; // 01100100
  x = x << 1; // 11001000
  x = x << 1; // 10010000
  x = x << 2; // 01000000
  ```

Примечание
Оператор <<, используемый для вывода данных в окно консоли, не имеет никакого отношения к побитовому оператору <<.

- >> — сдвиг вправо — сдвигает двоичное представление числа вправо на один или более разрядов и заполняет разряды слева нулями, если число положительное:

Листинги на странице http://unicross.narod.ru/cpp/
char x = 100;  // 01100100
x = x >> 1;  // 00110010
x = x >> 1;  // 00011001
x = x >> 2;  // 00001100

Если число отрицательное, то разряды слева заполняются единицами:
char x = -127;  // 10000001
x = x >> 1;  // 11000000
x = x >> 2;  // 11100000
x = x << 1;  // 11100000
x = x >> 1;  // 11110000

Примечание
Оператор >>, используемый для ввода данных, не имеет никакого отношения к побитовому оператору >>.

Операторы присваивания

Операторы присваивания предназначены для сохранения значения в переменной. Перечислим операторы присваивания доступные в языке C++:

= — присваивает переменной значение. Обратите внимание на то, что хотя оператор похож на математический знак равенства, смысл у него в языке C++ совершенно другой. Справа от оператора присваивания может располагаться константа или сложное выражение. Слева от оператора присваивания может располагаться переменная, указатель или объект (у которого перегружен оператор =), но не константа. Пример присваивания значения:

```c
int x;
x = 10;
x = 12 * 10 + 45 / 5;
12 + 45 = 45 + 5; // Так нельзя!!!
```

В одной инструкции можно присвоить значение сразу нескольким переменным:

```c
x = y = z = 2;
```

+= — увеличивает значение переменной на указанную величину:

```c
x += 10;  // Эквивалентно x = x + 10;
```

Листинги на странице http://unicross.narod.ru/cpp/
- = — уменьшает значение переменной на указанную величину:
  x -= 10; // Эквивалентно x = x - 10;

* = — умножает значение переменной на указанную величину:
  x *= 10 + 5; // Эквивалентно x = x * (10 + 5);

/ = — делит значение переменной на указанную величину:
  x /= 2; // Эквивалентно x = x / 2;

% = — делит значение переменной на указанную величину и возвращает остаток:
  x %= 2; // Эквивалентно x = x % 2;

&=, |=, ^=, <<= и >>= — побитовые операторы с присваиванием.

Оператор запятая

Оператор запятая позволяет разместить сразу несколько выражений внутри одной инструкции. Например, в предыдущих примерах мы использовали этот оператор для объявления нескольких переменных в одной инструкции:

```cpp
int x, y;
```

Результат вычисления последнего выражения можно присвоить какой-либо переменной. При этом выражения должны быть расположены внутри круглых скобок, так как приоритет оператора запятая меньше приоритета оператора присваивания. Пример:

```cpp
int x = 0, y = 0, z = 0;
x = (y = 10, z = 20, y + z);
std::cout << x << std::endl; // 30
std::cout << y << std::endl; // 10
std::cout << z << std::endl; // 20
```

В этом примере после объявления и инициализации переменных переменной y присваивается значение 10, затем переменной z присваивается значение 20, далее вычисляется выражение y + z и его результат присваивается переменной x.

Операторы сравнения

Операторы сравнения используются в логических выражениях. Перечислим операторы сравнения, доступные в языке C++:

Листинги на странице http://unicross.narod.ru/cpp/
=> — равно;
!= — не равно;
< — меньше;
> — больше;
<= — меньше или равно;
>= — больше или равно.

Логические выражения возвращают только два значения true (истина, соответствует числу 1) или false (ложь, соответствует числу 0). Пример вывода значения логического выражения:

```cpp
std::cout << (10 == 10) << std::endl; // 1 (true)
std::cout << (10 == 5) << std::endl; // 0 (false)
std::cout << (10 != 5) << std::endl; // 1 (true)
std::cout << (10 < 5) << std::endl; // 0 (false)
std::cout << (10 > 5) << std::endl; // 1 (true)
```

Логическое выражение может не содержать операторов сравнения вообще. В этом случае число 0 автоматически преобразуется в ложное значение (false), а любое ненулевое значение (в том числе и отрицательное) — в истинное значение (true):

```cpp
std::cout << (bool)10 << std::endl; // 1 (true)
std::cout << (bool)-10 << std::endl; // 1 (true)
std::cout << (bool)12.5 << std::endl; // 1 (true)
std::cout << (bool)0.1 << std::endl; // 1 (true)
std::cout << (bool)0.0 << std::endl; // 0 (false)
std::cout << (bool)0 << std::endl; // 0 (false)
```

В этом примере мы воспользовались приведением к типу bool. Внутри логического выражения приведение типов выполнять не нужно, так как оно производится автоматически.

Следует учитывать, что оператор проверки на равенство содержит два символа =. Указание одного символа = является логической ошибкой, так как этот оператор используется для присваивания значение переменной, а не для проверки условия. Использовать оператор присваивания внутри логического выражения допускается, поэтому компилятор не выведет сообщение об ошибке, однако программа может выполняться некорректно. Подобную ошибку часто допускают начинающие

Листинги на странице http://unicross.narod.ru/cpp/
программисты. Например, в следующем примере вместо проверки равенства числу 11, производится операция присваивания:

```cpp
int x = 10;
if (x == 11) {
    std::cout << "x == 11" << std::endl;
}
```

Любое число не равное 0 трактуется как логическое значение true, поэтому производится проверка условия true == true, которое является истинным.

Значение логического выражения можно инвертировать с помощью оператора !. В этом случае если логическое выражение возвращает false, то !false вернет значение true. Пример:

```cpp
std::cout << (10 == 5) << std::endl; // 0 (false)
std::cout << ! (10 == 5) << std::endl; // 1 (true)
```

Несколько логических выражений можно объединить в одно большое с помощью следующих операторов:

- `&&` — логическое И. Логическое выражение вернет true только в случае, если оба подвыражения вернут true. Пример:
  ```cpp
  std::cout << ( (10 == 10) && (5 != 3) )
             << std::endl; // 1 (true)
  std::cout << ( (10 == 10) && (5 == 3) )
             << std::endl; // 0 (false)
  ```

- `||` — логическое ИЛИ. Логическое выражение вернет true, если хотя бы одно из подвыражений вернет true. Следует учитывать, что если подвыражение вернет значение true, то следующие подвыражения вычисляться не будут. Например, в логическом выражении f1() || f2() || f3() функция f2() будет вызвана только если функция f1() вернет false, а функция f3() будет вызвана только если функции f1() и f2() вернут false. Пример использования оператора:
  ```cpp
  std::cout << ( (10 == 10) || (5 != 3) )
             << std::endl; // 1 (true)
  std::cout << ( (10 == 10) || (5 == 3) )
             << std::endl; // 1 (true)
  ```

Результаты выполнения операторов `&&` и `||` показаны в табл. 3.1.
Таблица 3.1. Операторы && и ||

| a | b | a && b | a || b |
|---|---|--------|--------|
| true | true | true | true |
| true | false | false | true |
| false | true | false | true |
| false | false | false | false |

Приоритет выполнения операторов

Все операторы выполняются в порядке приоритета (табл. 3.2). Вначале вычисляется выражение, в котором оператор имеет наивысший приоритет, а затем выражение с меньшим приоритетом. Например, выражение с оператором умножения будет выполнено раньше выражения с оператором сложения, так как приоритет оператора умножения выше. Если приоритет операторов одинаковый, то используется порядок вычислений, определенный для конкретного оператора. Операторы присваивания и унарные операторы выполняются справа налево. Математические, побитовые и операторы сравнения, а также оператор запятая выполняются слева направо. Изменить последовательность вычисления выражения можно с помощью круглых скобок. Пример:

```cpp
int x = 0;
x = 5 + 10 * 3 / 2;  // Умножение -> деление -> сложение
std::cout << x << std::endl;  // 20
x = (5 + 10) * 3 / 2;  // Сложение -> умножение -> деление
std::cout << x << std::endl;  // 22
```

Листинги на странице http://unicross.narod.ru/cpp/
### Таблица 3.2. Приоритеты операторов

<table>
<thead>
<tr>
<th>Приоритет</th>
<th>Операторы</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (высший)</td>
<td>( ) [ ] . ~ &gt;</td>
</tr>
<tr>
<td>2</td>
<td>++ -- ~ ! * (разыменование) &amp; (взятие адреса) - (унарный минус)</td>
</tr>
<tr>
<td>3</td>
<td>.* - &gt;*</td>
</tr>
<tr>
<td>4</td>
<td>* / %</td>
</tr>
<tr>
<td>5</td>
<td>+ - (минус)</td>
</tr>
<tr>
<td>6</td>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>7</td>
<td>&lt; &gt; &lt;= &gt;=</td>
</tr>
<tr>
<td>8</td>
<td>== !=</td>
</tr>
<tr>
<td>9</td>
<td>&amp; (побитовое И)</td>
</tr>
<tr>
<td>10</td>
<td>^</td>
</tr>
<tr>
<td>11</td>
<td>!</td>
</tr>
<tr>
<td>12</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>== /= %= += -= &gt;&gt;= &lt;&lt;= &amp;= ^=</td>
</tr>
<tr>
<td>15 (низший)</td>
<td>, (запятая)</td>
</tr>
</tbody>
</table>

### Оператор ветвления `if`

Оператор ветвления `if` позволяет в зависимости от значения логического выражения выполнить отдельный блок программы или наоборот не выполнять его. Оператор имеет следующий формат:

```cpp
if (<Логическое выражение>) {
    <Блок, выполняемый если условие истинно>
}
else {
    <Блок, выполняемый если условие ложно>
```

Листинги на странице http://unicross.narod.ru/cpp/
Если логическое выражение возвращает значение true (истина), то выполняются инструкции, расположенные внутри фигурных скобок, сразу после оператора if. Если логическое выражение возвращает значение false (ложь), то выполняются инструкции после ключевого слова else. Блок else является необязательным. Допускается не указывать фигурные скобки, если блоки состоят из одной инструкции. В качестве примера проверим число, введенное пользователем, на четность и выведем соответствующее сообщение (листинг 3.2).

```
#include <iostream>
#include <locale>

int main() {
    int x = 0;
    std::setlocale(LC_ALL, "Russian_Russia.1251");
    std::cout << "Введите число: ";
    std::cin >> x;
    if (!std::cin.good()) {
        std::cout << std::endl << "Вы ввели не число" << std::endl;
        std::cin.clear(); // Сбрасываем флаг ошибки
        std::cin.ignore(255, '\n').get();
    }
    else {
        if (x % 2 == 0)
            std::cout << x << " - четное число" << std::endl;
        else
            std::cout << x << " - нечетное число" << std::endl;
        std::cin.ignore(255, '\n').get();
    }
    return 0;
}
```

Как видно из примера, один оператор ветвления можно вложить в другой. Первый оператор if проверяет отсутствие ошибки при вводе числа. Метод good() возвращает логическое значение true, если ошибок не произошло, или значение false — в

Листинги на странице http://unicross.narod.ru/cpp/
противном случае. Обратите внимание на то, что логическое выражение не содержит операторов сравнения (кроме оператора !, который инвертирует значение):

```cpp
if (!std::cin.good()) {
    // Проверка соответствия условия значению true производится по умолчанию. Так как
    // при ошибке метод good() вернет значение false, следовательно условие !false ==
    // true будет истинным. Это логическое выражение можно было записать так:
    if (std::cin.good() == false) {
        // или так:
        if (std::cin.good() != true) {
```

Однако подобные проверки в этом случае являются излишними. Достаточно указать
название функции и при необходимости инвертировать значение.

Если ошибки при вводе числа нет, то выполняется блок else. В этом блоке находится
вложенный оператор ветвления, который проверяет число на четность. В зависимости
от условия `x % 2 == 0` выводится соответствующее сообщение. Если число делится на
2 без остатка, то оператор `&` вернет значение 0, в противном случае — число 1.
Обратите внимание на то, что оператор ветвления не содержит фигурных скобок:

```cpp
if (x % 2 == 0)
    std::cout << x << " - четное число" << std::endl;
else
    std::cout << x << " - нечетное число" << std::endl;
std::cin.ignore(255, '\n').get();
```

В этом случае считается, что внутри блока содержится только одна инструкция.
Поэтому последняя инструкция к блоку else не относится. Она будет выполнена в
любом случае, вне зависимости от условия. Чтобы это сделать наглядным, перед
инструкциями, расположенными внутри блока, добавлено одинаковое количество
пробелов. Если записать следующим образом, то ничего не изменится:

```cpp
if (x % 2 == 0)
    std::cout << x << " - четное число" << std::endl;
else
    std::cout << x << " - нечетное число" << std::endl;
std::cin.ignore(255, '\n').get();
```

Однако в дальнейшем разбираться в таком коде будет неудобно. Поэтому перед
инструкциями внутри блока всегда следует размещать одинаковый отступ. В качестве
отступа можно использовать пробелы или символ табуляции. При использовании
пробелов размер отступа равняется трем или четырем пробелам для блока первого

Листинги на странице http://unicross.narod.ru/cpp/
уровня. Для вложенных блоков количество пробелов умножают на уровень вложенности. Если для блока первого уровня вложенности использовалось три пробела, то для блока второго уровня вложенности должно использоваться шесть пробелов, для третьего уровня — девять пробелов и т. д. В одной программе не следует использовать и пробелы и табуляцию в качестве отступа. Необходимо выбрать что-то одно и пользоваться этим во всей программе.

Если блок состоит из нескольких инструкций, то следует указать фигурные скобки. Существует несколько стилей размещения скобок в операторе `if`:

```c++
// Стиль 1
if (<Логическое выражение>) {
    // Инструкции
}
else {
    // Инструкции
}

// Стиль 2
if (<Логическое выражение>) {
    // Инструкции
} else {
    // Инструкции
}

// Стиль 3
if (<Логическое выражение>)
{
    // Инструкции
}
else
{
    // Инструкции
}
```

Многие программисты считают стиль 3 наиболее приемлемым, так как открывающая и закрывающая скобки расположены друг под другом. На мой же взгляд образуются лишние пустые строки. Так как размеры экрана ограничены, при наличии пустой строки на экран помещается меньше кода и приходится чаще пользоваться полосой прокрутки. Если размещать инструкции с равным отступом, то блок кода выделяется визуально и следить за положением фигурных скобок просто излишне. Тем более, что редактор кода

Листинги на странице http://unicross.narod.ru/cpp/
позволяет подсветить парные скобки. Какой стиль использовать, зависит от личного предпочтения программиста или от правил оформления кода, принятых в определенной фирме. Главное, чтобы стиль оформления внутри одной программы был одинаковым. В этой книге мы будем пользоваться стилем 1.

В листинге 3.2 при вложении операторов мы воспользовались следующей схемой:

```c++
if (<Условие 1>) {
    // Иструкции
}
else {
    if (<Условие 2>) {
        // Иструкции
    }
    else {
        // Иструкции
    }
}
```

Чтобы проверить несколько условий эту схему можно изменить так:

```c++
if (<Условие 1>) {
    // <Блок 1>
}
else if (<Условие 2>) {
    // <Блок 2>
}
// ... Фрагмент опущен ...
else if (<Условие N>) {
    // <Блок N>
}
else {
    // <Блок else>
}
```

Если <Условие 1> истинно, то выполняется <Блок 1>, а все остальные условия пропускаются. Если <Условие 1> ложно, то проверяется <Условие 2>. Если <Условие 2> истинно, то выполняется <Блок 2>, а все остальные условия пропускаются. Если <Условие 2> ложно, то точно также проверяются остальные

Листинги на странице http://unicross.narod.ru/cpp/
условия. Если все условия ложны, то выполняется <Блок else>. В качестве примера определим какое число от 0 до 2 ввел пользователь (листинг 3.3).

Листинг 3.3. Проверка нескольких условий

```cpp
#include <iostream>
#include <locale>

int main() {
    int x = 0;
    std::setlocale(LC_ALL, "Russian_Russia.1251");
    std::cout << "Введите число от 0 до 2: ";
    std::cin >> x;
    if (!std::cin.good()) {
        std::cout << std::endl << "Вы ввели не число" << std::endl;
        std::cin.clear(); // Сбрасываем флаж ошибка
    }
    else if (x == 0)
        std::cout << "Вы ввели число 0" << std::endl;
    else if (x == 1)
        std::cout << "Вы ввели число 1" << std::endl;
    else if (x == 2)
        std::cout << "Вы ввели число 2" << std::endl;
    else {
        std::cout << "Вы ввели другое число" << std::endl;
        std::cout << "x = " << x << std::endl;
    }
    std::cin.ignore(255, '\n').get();
    return 0;
}
```

Оператор ?::

Для проверки условия вместо оператора if можно использовать оператор ?::. Оператор имеет следующий формат:

Листинги на странице http://unicross.narod.ru/cpp/
<Переменная> = <Логическое выражение> ? <Выражение если Истина> :  
<Выражение если Ложь>;

Если логическое выражение возвращает значение true, то выполняется выражение, расположенные после вопросительного знака. Если логическое выражение возвращает значение false, то выполняется выражение, расположенные после двоеточия. Результат выполнения выражений становится результатом выполнения оператора. Пример проверки числа на четность и вывода результата:

```cpp
int x = 10;
std::cout << x << 
    << ((x % 2 == 0) ? " - четное число" : " - нечетное число");
```

Обратите внимание на то, что в качестве операндов указываются именно выражения, а не инструкции. Кроме того, выражения обязательно должны возвращать какое-либо значение. Так как оператор возвращает значение, его можно использовать внутри выражений:

```cpp
int x, y;
x = 0;
y = 30 + 10 / (!x ? 1 : x); // 30 + 10 / 1
std::cout << y << std::endl; // 40
x = 2;
y = 30 + 10 / (!x ? 1 : x); // 30 + 10 / 2
std::cout << y << std::endl; // 35
```

В качестве операнда можно указать функцию, которая возвращает значение:

```cpp
int func1(int x) {
    std::cout << x << " - четное число" << std::endl;
    return 0;
}
int func2(int x) {
    std::cout << x << " - нечетное число" << std::endl;
    return 0;
}
```

// ... Фрагмент опущен ...

```cpp
int x = 10;
(x % 2 == 0) ? func1(x) : func2(x);
```

Как видно из примера, значение, возвращаемое оператором, можно проигнорировать.

Листинги на странице http://unicross.narod.ru/cpp/
Оператор выбора \texttt{switch}

Оператор выбора \texttt{switch} имеет следующий формат:

\begin{verbatim}
switch (\texttt{<Выражение>}) {
    case \texttt{<Константа 1>}:
        \texttt{<Инструкции>}
        break;
    [...
         case \texttt{<Константа N>}:
         \texttt{<Инструкции>}
         break;]
    [ default:
    \texttt{<Инструкции>}]
}
\end{verbatim}

Вместо условия оператор \texttt{switch} принимает выражение. В зависимости от значения выражения выполняется один из блоков case, в котором указано это значение. Значением выражения должно быть целое число или символ. Если ни одно из значений не описано в блоках case, то выполняется блок default (если он указан). Обратите внимание на то, что значения в блоках case не могут иметь одинаковые константы. Пример использования оператора \texttt{switch} приведен в листинге 3.4.

\begin{verbatim}
Листинг 3.4. Использование оператора \texttt{switch}

#include <iostream>
#include <locale>

int main() {
    int os = 0;
    std::setlocale(LC_ALL, "Russian_Russia.1251");
    std::cout << "Какой операционной системой вы пользуетесь?\n"
1 - Windows XP\n2 - Windows Vista\n3 - Windows 7\n4 - Другая\nВведите число, соответствующее ответу: ";
    std::cin >> os;

Листинги на странице http://unicross.narod.ru/cpp/
\end{verbatim}
if (!std::cin.good()) {
    std::cout << std::endl << "Вы ввели не число" << std::endl;
    std::cin.clear(); // Сбрасываем флаг ошибки
    std::cin.ignore(255, 'n').get();
    return 0;
}
std::cout << std::endl;
switch (os) {
    case 1:
        std::cout << "Вы выбрали - Windows XP";
        break;
    case 2:
        std::cout << "Вы выбрали - Windows Vista";
        break;
    case 3:
        std::cout << "Вы выбрали - Windows 7";
        break;
    case 4:
        std::cout << "Вы выбрали - Другая";
        break;
    default:
        std::cout << "Мы не смогли определить вашу операционную "
        << "систему";
}
std::cout << std::endl;
std::cin.ignore(255, 'n').get();
return 0;

Как видно из примера, в конце каждого блока case указан оператор break. Этот оператор позволяет досрочно выйти из оператора выбора switch. Если не указать оператор break, то будет выполняться следующий блок case вне зависимости от указанного значения. В некоторых случаях это может быть полезным. Например, можно выполнить один и те же инструкции при разных значениях, разместив инструкции в конце диапазона значений. Пример:

switch (ch) {
    case 'a':

Листинги на странице http://unicross.narod.ru/cpp/
case 'b':
case 'c':
    std::cout << "a, b или c";
    break;
case 'd':
    std::cout << "Только d";
}

Цикл for

Операторы циклов позволяют выполнить одни и те же инструкции многократно. Предположим, нужно вывести все числа от 1 до 100 по одному на строке. Обычным способом пришлось бы писать 100 строк кода:

```cpp
std::cout << 1 << std::endl;
std::cout << 2 << std::endl;
...
std::cout << 100 << std::endl;
```

С помощью циклов то же действие можно выполнить одной строкой кода:

```cpp
for (int i=1; i<=100; ++i) std::cout << i << std::endl;
```

Цикл for используется для выполнения выражений определенное число раз. Цикл имеет следующий формат:

```cpp
for (<Начальное значение>; <Условие>; <Приращение>) {
    <Инструкции>
}
```

Параметры имеют следующие значения:

- `<Начальное значение>` — присваивает переменной-счетчику начальное значение;
- `<Условие>` — содержит логическое выражение. Пока логическое выражение возвращает значение true, выполняются инструкции внутри цикла;
- `<Приращение>` — задает изменение переменной-счетчика на каждой итерации.

Последовательность работы цикла for:

1. Переменной-счетчику присваивается начальное значение.
2. Проверяется условие, если оно истинно, выполняются выражения внутри цикла, а в противном случае выполнение цикла завершается.

Листинги на странице http://unicross.narod.ru/cpp/
3. Переменная-счетчик изменяется на величину, указанную в <Приращение>.

4. Переход к п. 2.
Переменная-счетчик может быть объявлена как вне цикла for, так и в параметре <Начальное значение>. Если переменная объявлена в параметре, то она будет видна только внутри цикла. Кроме того, допускается объявить переменную вне цикла и сразу присвоить ей начальное значение. В этом случае параметр <Начальное значение> можно оставить пустым. Пример:

```cpp
int i; // Объявление вне цикла
for (i=1; i<=20; ++i) {
    std::cout << i << std::endl;
}
// Переменная i видна вне цикла
std::cout << i << std::endl; // 21
// Объявление внутри цикла
for (int j=1; j<=20; ++j) {
    std::cout << j << std::endl;
}
// Переменная j НЕ видна вне цикла
int k = 1; // Инициализация вне цикла
for (; k<=20; ++k) {
    std::cout << k << std::endl;
}
```

Цикл выполняется до тех пор, пока <Условие> не вернет false. Если это не произойдет, то цикл будет бесконечным. Логическое выражение, указанное в параметре <Условие>, вычисляется на каждой итерации. Поэтому, если внутри логического выражения производятся какие-либо вычисления и значение не изменяется внутри цикла, то вычисление следует вынести в параметр <Начальное значение>. В этом случае вычисление указывается после присваивания значения переменной-счетчику через запятую. Пример:

```cpp
for (int i=1, j=10+30; i<=j; ++i) {
    std::cout << i << std::endl;
}
```

Выражение, указанное в параметре <Приращение>, может не только увеличивать значение переменной-счетчика, но и уменьшать его. Кроме того, значение может изменяться на любую величину. Пример:

Листинги на странице http://unicross.narod.ru/cpp/
// Выводим числа от 100 до 1
for (int i=100; i>0; --i) {
    std::cout << i << std::endl;
}

// Выводим четные числа от 1 до 100
for (int j=2; j<100; j+=2) {
    std::cout << j << std::endl;
}

Если переменная-счетчик изменяется внутри цикла, то выражение в параметре <Приращение> можно вообще не указывать:

for (int i=1; i<=10; ) {
    std::cout << i << std::endl;
    ++i; // Приращение
}

Все параметры цикла for и инструкции внутри цикла являются необязательными. Хотя параметры можно не указывать, точки с запятой обязательно должны быть. Если все параметры не указаны, то цикл будет бесконечным. Чтобы выйти из бесконечного цикла следует использовать оператор break. Пример:

int i = 1; // <Начальное значение>
for (; ; ) { // Бесконечный цикл
    if (i<=10) { // <Условие>
        std::cout << i << std::endl;
        ++i; // <Приращение>
    }
    else {
        break; // Выходим из цикла
    }
}

Цикл while

Выполнение выражений в цикле while продолжается до тех пор, пока логическое выражение истинно. Имеет следующий формат:

<Начальное значение>;
while (<Условие>) {
    <Инструкции>;

Листинги на странице http://unicross.narod.ru/cpp/
<Приращение>;
}

Последовательность работы цикла while:
1. Переменной-счетчику присваивается начальное значение.
2. Проверяется условие, если оно истинно, выполняются инструкции внутри цикла, иначе выполнение цикла завершается.
3. Переменная-счетчик изменяется на величину, указанную в <Приращение>.
4. Переход к пункту 2.

Выведем все числа от 1 до 100, используя цикл while:

```cpp
int i = 1;  // <Начальное значение>
while (i<=100) {  // <Условие>
    std::cout << i << std::endl;  // <Инструкции>
    ++i;  // <Приращение>
}
```

**Внимание!**
Если <Приращение> не указано, то цикл будет бесконечным.

**Цикл do...while**

Выполнение выражений в цикле do...while продолжается до тех пор, пока логическое выражение истинно. В отличие от цикла while условие проверяется не в начале цикла, а в конце. По этой причине инструкции внутри цикла do...while выполняются минимум один раз. Цикл имеет следующий формат:

<Начальное значение>;
do {
    <Инструкции>;
    <Приращение>;
} while (<Условие>);

Последовательность работы цикла do...while:
1. Переменной-счетчику присваивается начальное значение.
2. Выполняются инструкции внутри цикла.
3. Переменная-счетчик изменяется на величину, указанную в <Приращение>.

Листинги на странице http://unicross.narod.ru/cpp/
4. Проверяется условие, если оно истинно, происходит переход к п. 2, а если нет — выполнение цикла завершается.

Выведем все числа от 1 до 100, используя цикл do-while:

```c++
int i = 1; // <Начальное значение>
do {
    std::cout << i << std::endl; // <Инструкция>
    ++i;                      // <Приращение>
} while (i <= 100); // <Условие>
```

**Внимание!**
Если <Приращение> не указано, то цикл будет бесконечным.

**Оператор continue. Переход на следующую итерацию цикла**

Оператор continue позволяет перейти к следующей итерации цикла до завершения выполнения всех выражений внутри цикла. В качестве примера выведем все числа от 1 до 100, кроме чисел от 5 до 10 включительно:

```c++
for (int i = 1; i <= 100; ++i) {
    if (i > 4 && i < 11) continue;
    std::cout << i << std::endl;
}
```

**Оператор break. Прерывание цикла**

Оператор break позволяет прервать выполнение цикла досрочно. Для примера выведем все числа от 1 до 100 еще одним способом:

```c++
int i = 1;
while (1) {
    if (i > 100) break;
    std::cout << i << std::endl;
    ++i;
}
```

Здесь мы в условии указали значение 1. В этом случае инструкции внутри цикла будут выполняться бесконечно. Однако использование оператора break прерывает его выполнение, как только 100 строк уже напечатано.

Листинги на странице http://unicross.narod.ru/cpp/
**Обратите внимание**
Оператор break прерывает выполнение цикла, а не программы, то есть далее будет выполнена инструкция, следующая сразу за циклом.

Бесконечный цикл совместно с оператором break удобно использовать для получения неопределенного заранее количества данных от пользователя. В качестве примера просуммируем неопределенное количество целых чисел (листинг 3.5).

**Листинг 3.5. Суммирование неопределенного количества чисел**

```cpp
#include <iostream>
#include <locale>

int main() {
    int x = 0, summa = 0;
    std::setlocale(LC_ALL, "Russian_Russia.1251");
    std::cout << "Введите число 0 для получения результата" << std::endl;
    for ( ; ; ) {
        std::cout << "Введите число: ";
        std::cin >> x;
        if (!std::cin.good()) {
            std::cout << "Вы ввели не число!" << std::endl;
            std::cin.clear(); // Сбрасываем флаг ошибки
            std::cin.ignore(255, '\n');
            continue;
        }
        if (!x) break;
        summa += x;
    }
    std::cout << "Сумма чисел равна: " << summa << std::endl;
    std::cin.ignore(255, '\n').get();
    return 0;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
Оператор goto

С помощью оператора безусловного перехода goto можно передать управление в любое место программы. Оператор имеет следующий формат:

goto <Метка>;

Значение в параметре <Метка> должно быть допустимым идентификатором. Место в программе, в которое передается управление, помечается одноименной меткой, после которой указывается двоеточие. В качестве примера имитируем цикл и выводим числа от 1 до 100:

```cpp
int i = 1;
BLOCK_START: {
    if (i > 100) goto BLOCK_END;
    std::cout << i << std::endl;
    ++i;
    goto BLOCK_START;
}
BLOCK_END;
```

Как видно из примера, фигурные скобки можно использовать не только применительно к условным операторам, циклам, функциям, но и как отдельную конструкцию. Фрагмент кода, заключенный в фигурные скобки, называется блоком. Переменные, объявленные внутри блока видны только в пределах блока.

**Совет**

Следует избегать использования оператора goto, так как его применение делает программу слишком запутанной и может привести к неожиданным результатам.

Листинги на странице http:// unicross.narod.ru/cpp/
Программирование на C++
в Visual Studio® 2010 Express

Глава 4. Числа

© Прохоренок Н.А., 2010 г., unicross@mail.ru

Этот DjVu-файл предоставляется КАК ЕСТЬ. Автор не несет никакой ответственности за прямые или косвенные проблемы, связанные с использованием данного файла. ВЫ ИСПОЛЬЗУЕТЕ ЕГО НА СВОЙ СТРАХ И РИСК.

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Запрещено:
- перепечатывать всю книгу или отдельные главы без письменного разрешения Автора;
- декомпилировать данный файл и преобразовывать его в любой другой формат.
В языке C++ практически все элементарные типы данных (bool, char, wchar_t, int, float и double) являются числовыми. Тип bool может принимать только значения 1 и 0, которые соответствуют константам true и false. Типы char и wchar_t содержат код символа, а не сам символ. Поэтому значения этих типов можно использовать в одном выражении вместе со значениями, имеющими типы int, float и double.
Пример:

```cpp
bool a = true; // 1
char ch = 'w'; // 119
std::cout << (a + ch + 10) << std::endl; // 130
```

Для хранения целых чисел предназначен тип int. Диапазон значений типа int зависит от разрядности операционной системы. В 16-битной операционной системе диапазон от -32 768 до 32 767. В 32-битной операционной системе диапазон от -2 147 483 648 до 2 147 483 647. С помощью ключевых слов short, long и long long можно указать точный размер типа int. При использовании этих ключевых слов тип int подразумевается по умолчанию, поэтому его можно не указывать. Тип short занимает 2 байта (диапазон от -32 768 до 32 767), тип long — 4 байта (диапазон от -2 147 483 648 до 2 147 483 647), а тип long long занимает 8 байт (диапазон значений от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807). В VC++ вместо типов short, long и long long можно использовать типы __int16, __int32 и __int64 соответственно (перед типом указано два символа подчеркивания).

По умолчанию целочисленные типы являются знаковыми. Знак числа хранится в старшем бите: 0 соответствует положительному числу, а 1 — отрицательному. С помощью ключевого слова unsigned можно указать, что число является только положительным. Тип unsigned short может содержать числа в диапазоне от 0 до 65 535, тип unsigned long — от 0 до 4 294 967 295, а тип unsigned long long — от 0 до 18 446 744 073 709 551 615. При преобразовании значения из типа signed в тип unsigned следует учитывать, что знаковый бит (если число отрицательное, то бит содержит значение 1) может стать причиной очень больших чисел, так как старший бит у типа unsigned не содержит признака знака:

```cpp
int x = -1;
std::cout << (unsigned int)x << std::endl; // 4294967295
```

Целочисленное значение задается в десятичной, восьмеричной или шестнадцатеричной форме. Восьмеричные числа начинаются с нуля и содержат цифры от 0 до 7. Шестнадцатеричные числа начинаются с комбинации символов 0x (или 0X) и могут

Листинги на странице http://unicross.narod.ru/cpp/
содержать числа от 0 до 9 и буквы от A до F (регистр букв не имеет значения). Восьмеричные и шестнадцатеричные значения преобразуются в десятичное значение. Пример:
```c
int x, y, z;
```
```c
x = 119; // Десятичное значение
```
```c
y = 0167; // Восьмеричное значение
```
```c
z = 0x77; // Шестнадцатеричное значение
```
По умолчанию целочисленные константы имеют тип `signed int`. Если необходимо изменить тип на другой, то после числа указываются следующие буквы:

- `l` (или `L`) — тип `long int`. Пример указания значения: `10L;`
- `u` (или `U`) — тип `unsigned int`. Пример указания значения: `10u;`
- если буквы `u` и `l` указаны одновременно, то тип будет `unsigned long int`. Пример указания значения: `10UL;`

Для хранения вещественных чисел предназначены типы `float` и `double`. Вещественное число может содержать точку и (или) экспоненту, начинаящуюся с буквы `e` (регистр не имеет значения):
```c
float x, y;
```
```c
double z, k;
```
```c
x = 20.0;
```
```c
y = 12.1e5;
```
```c
z = .123;
```
```c
k = 47.E-5;
```
По умолчанию вещественные константы имеют тип `double`. Если необходимо изменить тип на другой, то после числа указываются следующие буквы:

- `f` (или `f`) — тип `float`. Пример указания значения: `12.3f;`
- `u` (или `u`) — тип `long double`. Значение должно содержать точку и (или) экспоненту, иначе тип будет `long int`. Пример указания значения: `12.3L;`

При выполнении операций над вещественными числами следует учитывать ограничения точности вычислений. Например, результат следующей инструкции может показаться странным:
```c
std::cout << (0.3-0.1-0.1-0.1) << std::endl; // -2.77556e-017
```
Ожидаемым был бы результат 0.0, но как видно из примера мы получили совсем другой результат (`-2.77556e-017`).

Листинги на странице [http://unicross.narod.ru/cpp/]
Если в выражении используются числа, имеющие разный тип данных, то тип результата выражения будет соответствовать наиболее сложному типу. Например, если производится сложение переменной, имеющей тип int, с переменной имеющей тип double, то целое число будет автоматически преобразовано в вещественное. Результатом этого выражения будет значение, имеющее тип double. Однако, если результат выражения присваивается переменной типа int, то тип double будет преобразован в тип int (при этом компилятор выведет предупреждающее сообщение о возможной потере данных):

```cpp
int x = 10, y = 0;
double z = 12.5;
y = x + z;
std::cout << y << std::endl; // 22 (тип int)
std::cout << (x + z) << std::endl; // 22.5 (тип double)
```

### Математические константы

В VC++ в заголовочном файле math.h определены следующие макросы, содержащие значения математических констант:

- **M_PI** — число ПИ. Определение макроса:
  
  ```cpp
  #define M_PI 3.14159265358979323846
  ```

- **M_PI_2** — значение выражения pi/2. Определение макроса:
  
  ```cpp
  #define M_PI_2 1.57079632679489661923
  ```

- **M_PI_4** — значение выражения pi/4. Определение макроса:
  
  ```cpp
  #define M_PI_4 0.785398163397448309616
  ```

- **M_1_PI** — значение выражения 1/pi. Определение макроса:
  
  ```cpp
  #define M_1_PI 0.318309886183790671538
  ```

- **M_2_PI** — значение выражения 2/pi. Определение макроса:
  
  ```cpp
  #define M_2_PI 0.636619772367581343076
  ```

- **M_E** — значение константы е. Определение макроса:
  
  ```cpp
  #define M_E 2.71828182845904523536
  ```

- **M_LOG2E** — значение выражения log2(e). Определение макроса:
  
  ```cpp
  #define M_LOG2E 1.44269504088896340736
  ```

Листинги на странице http://unicross.narod.ru/cpp/
→ **M\_LOG10E** — значение выражения \( \log_{10}(e) \). Определение макрона:

```cpp
#define M_LOG10E 0.434294481903251827651
```

→ **M\_LN2** — значение выражения \( \ln(2) \). Определение макрона:

```cpp
#define M_LN2 0.693147180559945309417
```

→ **M\_LN10** — значение выражения \( \ln(10) \). Определение макрона:

```cpp
#define M_LN10 2.30258509299404568402
```

→ **M\_2\_SQRTPI** — значение выражения \( 2/\sqrt{\pi} \). Определение макрона:

```cpp
#define M_2_SQRTPI 1.12837916709551257390
```

→ **M\_SQRT2** — значение выражения \( \sqrt{2} \). Определение макрона:

```cpp
#define M_SQRT2 1.41421356237309504880
```

→ **M\_SQRT1\_2** — значение выражения \( 1/\sqrt{2} \). Определение макрона:

```cpp
#define M_SQRT1_2 0.707106781186547524401
```

Прежде чем использовать константы необходимо перед подключением файла `math.h` определить макрос с названием `\_USE\_MATH\_DEFINES`. Пример вывода значения констант приведен в листинге 4.1.

### Листинг 4.1. Вывод значений математических констант

```cpp
#include <iostream>
#define _USE_MATH_DEFINES
#include <math.h>

int main() {
    std::cout << M_PI << std::endl;  // 3.14159
    std::cout << M_E << std::endl;   // 2.71828
    std::cin.get();
    return 0;
}
```

### Основные функции для работы с числами

Перечислим основные функции для работы с числами:

abs() — возвращает абсолютное значение. Прототипы функции:

```cpp
#include <cstdlib>
int abs(int X);
long abs(long X);
long long abs(long long X);
#include <cmath>
float abs(float X);
double abs(double X);
long double abs(long double X);
```

Функция abs() для типа long вызывает функцию labs(), для типа long long — llabs(), для типа float — fabsf(), для типа double — fabs(), а для типа long double — fabsl(). Пример:
```
std::cout << std::abs(-1) << std::endl; // 1
```

pow() — возведит число х в степень y. В случае переполнения возникает ошибка выхода за пределы допустимых значений. Прототипы функции:

```cpp
#include <cmath>
float pow(float X, int Y);
float pow(float X, float Y);
double pow(double X, int Y);
double pow(double X, double Y);
long double pow(long double X, int Y);
long double pow(long double X, long double Y);
```

Пример:
```
std::cout << std::pow(10.0, 2) << std::endl; // 100
std::cout << std::pow(3.0, 3.0) << std::endl; // 27
```

sqrt() — квадратный корень. Прототипы функции:

```cpp
#include <cmath>
float sqrt(float X);
double sqrt(double X);
long double sqrt(long double X);
```

Листинги на странице http://unicross.narod.ru/cpp/
Функция sqrt() для типа float вызывает функцию sqrtf(), а для типа long
double — sqrtl(). Пример использования функции:
std::cout << std::sqrt(100.0) << std::endl; // 10
std::cout << std::sqrt(25.0) << std::endl; // 5

- exp() — экспонента. Прототипы функции:
#include <cmath>
float exp(float X);
double exp(double X);
long double exp(long double X);

Функция exp() для типа float вызывает функцию expf(), а для типа long
double — expl();

- log() — натуральный логарифм. Прототипы функции:
#include <cmath>
float log(float X);
double log(double X);
long double log(long double X);

Функция log() для типа float вызывает функцию logf(), а для типа long
double — log1();

- log10() — десятичный логарифм. Прототипы функции:
#include <cmath>
float log10(float X);
double log10(double X);
long double log10(long double X);

Функция log10() для типа float вызывает функцию log10f(), а для типа long
double — log10l();

- fmod() — остаток от деления. Прототипы функции:
#include <cmath>
float fmod(float X, float Y);
double fmod(double X, double Y);
long double fmod(long double X, long double Y);

Листинги на странице http://unicross.narod.ru/cpp/
Функция fmod() для типа float вызывает функцию fmodf(), а для типа long
double — fmodl(). Пример использования функции:
std::cout << std::fmod(100.0, 9.0) << std::endl; // 1
std::cout << std::fmod(100.0, 10.0) << std::endl; // 0

modf() — разделяет вещественное число X на целую и дробную части. В качестве
значения функция возвращает дробную часть. Целая часть сохраняется в
переменной, адрес которой передан во втором параметре. Прототипы функции:
#include <cmath>
float modf(float X, float *Y);
double modf(double X, double *Y);
long double modf(long double X, long double *Y);

Функция mod() для типа float вызывает функцию modff(), а для типа long
double — modfl(). Пример использования функции:
double x = 0.0;
std::cout << std::modf(12.5, &x) << std::endl; // 0.5
std::cout << x << std::endl; // 12

div() — возвращает структуру из двух полей: quot (результат целочисленного
dеления X / Y), rem (остаток от деления X % Y). Прототипы функции:
#include <cstdlib>
div_t div(int X, int Y);
ldiv_t div(long X, long Y);
lldiv_t div(long long X, long long Y);

Объявления структур:
typedef struct _div_t {
    int quot;
    int rem;
} div_t;
typedef struct _ldiv_t {
    long quot;
    long rem;
} ldiv_t;

Листинги на странице http://unicross.narod.ru/cpp/
typedef struct _lldiv_t {
    long long quot;
    long long rem;
} lldiv_t;

Функция div() для типа long вызывает функцию ldiv(), а для типа long long — lldiv(). Пример использования функции:

div_t d;
d = std::div(13, 2);
std::cout << d.quot << std::endl;  // 6 == 13 / 2
std::cout << d.rem << std::endl;   // 1 == 13 % 2

Округление чисел

Для округления чисел предназначены следующие функции:

- **ceil()** — возвращает значение, округленное до ближайшего большего значения. Прототипы функции:

```
#include <cmath>

float ceil(float X);
double ceil(double X);
long double ceil(long double X);
```

Функция ceil() для типа float вызывает функцию ceill(), а для типа long double — ceil(). Пример использования функции:

```
std::cout << std::ceil(1.49) << std::endl;  // 2
std::cout << std::ceil(1.5) << std::endl;    // 2
std::cout << std::ceil(1.51) << std::endl;   // 2
```

- **floor()** — значение, округленное до ближайшего меньшего значения. Прототипы функции:

```
#include <cmath>

float floor(float X);
double floor(double X);
long double floor(long double X);
```

Листинги на странице http://unicross.narod.ru/cpp/
Функция floor() для типа float вызывает функцию floorf(), а для типа long
double — floorl(). Пример использования функции:
```cpp
std::cout << std::floor(1.49) << std::endl; // 1
std::cout << std::floor(1.5) << std::endl; // 1
std::cout << std::floor(1.51) << std::endl; // 1
```

Тригонометрические функции

В языке С++ доступны следующие тригонометрические функции:

- `sin()`, `cos()`, `tan()` — стандартные тригонометрические функции (синус,
  косинус, тангенс). Значение указывается в радианах. Прототипы функций:
  ```cpp
  #include <cmath>
  
  float sin(float X);
  float cos(float X);
  float tan(float X);
  double sin(double X);
  double cos(double X);
  double tan(double X);
  long double sin(long double X);
  long double cos(long double X);
  long double tan(long double X);
  
  Для типа float вызываются функции sinf(), cosf() и tanf(), а для типа long
double — sinl(), cosl() и tanl();
  ```

- `asin()`, `acos()`, `atan()` — обратные тригонометрические функции (арксинус,
  арккосинус, арктангенс). Значение возвращается в радианах. Прототипы функций:
  ```cpp
  #include <cmath>
  
  float acos(float X);
  float asin(float X);
  float atan(float X);
  double asin(double X);
  double acos(double X);
  double atan(double X);
  ```

Листинги на странице http://unicross.narod.ru/cpp/
long double acos(long double X);
long double asin(long double X);
long double atan(long double X);

Для типа float вызываются функции asinf(), acosf() и atanf(), а для типа long double — asinl(), acosl() и atanl().

Преобразование строки в число

Для преобразования строки в число используются следующие функции:

- atoi() — преобразует C-строку в число, имеющее тип int. Прототип функции:

```
#include <cstdlib>
int atoi(const char *Str);
```

Считывание символов производится пока они соответствуют цифрам. Иными словами, в строке могут содержаться не только цифры. Пробельные символы в начале строки игнорируются. Пример преобразования:

```
std::cout << std::atoi("25") << std::endl;       // 25
std::cout << std::atoi("2.5") << std::endl;      // 2
std::cout << std::atoi("5str") << std::endl;    // 5
std::cout << std::atoi("5s10") << std::endl;    // 5
std::cout << std::atoi("\t\n 25") << std::endl;  // 25
std::cout << std::atoi("-25") << std::endl;      // -25
std::cout << std::atoi("str") << std::endl;      // 0
```

- atol() — преобразует C-строку в число, имеющее тип long. Прототип функции:

```
#include <cstdlib>
long atol(const char *Str);
```

Пример преобразования:

```
std::cout << std::atol("25") << std::endl;       // 25
std::cout << std::atol(" -25") << std::endl;      // -25
std::cout << std::atol("2.5") << std::endl;      // 2
std::cout << std::atol("5str") << std::endl;    // 5
std::cout << std::atol("5s10") << std::endl;    // 5
std::cout << std::atol("str") << std::endl;      // 0
```

Листинги на странице http://unicross.narod.ru/cpp/
strtol() и strtol() — преобразуют C-строку в число, имеющее тип long и unsigned long соответственно. Прототипы функций:

```c
#include <cstdlib>

long strtol(const char *Str, char **EndPtr, int Radix);
unsigned long strtoul(const char *Str, char **EndPtr, int Radix);
```

Пробельные символы в начале строки игнорируются. Считывание символов заканчивается на символе не являющимся записью числа. Указатель на этот символ доступен через параметр EndPtr, если он не равен 0. В параметре Radix можно указать систему счисления (число от 2 до 36). Если в параметре radix задано число 0, то система счисления определяется автоматически. Пример преобразования:

```c
char *p = 0;
std::cout << std::strtol("25", 0, 0) << std::endl; // 25
std::cout << std::strtol("025", 0, 0) << std::endl; // 21
std::cout << std::strtol("0x25", 0, 0) << std::endl; // 37
std::cout << std::strtol("111", 0, 2) << std::endl; // 7
std::cout << std::strtol("025", 0, 8) << std::endl; // 21
std::cout << std::strtol("0x25", 0, 16) << std::endl; // 37
std::cout << std::strtol("5s10", &p, 0) << std::endl; // 5
std::cout << p << std::endl; // s10
```

Если в строке нет числа, то возвращается число 0. Если число выходит за диапазон значений для типа, то значение будет соответствовать минимальному (LONG_MIN или 0) или максимальному (LONG_MAX или ULONG_MAX) значению для типа, а переменной errno присваивается значение ERANGE. Чтобы сбросить флаг ошибки следует присвоить значение 0 переменной errno. Пример:

```c
std::cout << std::strtol("str", 0, 0) << std::endl; // 0
std::cout << std::strtol("99999999999999", 0, 0)
    << std::endl; // 2147483647 (соответствует LONG_MAX)
if (errno == ERANGE) {
    std::cout << "Error" << std::endl; // Error
}
```

errno = 0; // Сбрасываем флаг ошибки
std::cout << std::strtol("25", 0, 0) << std::endl; // 25

Листинги на странице http://unicross.narod.ru/cpp/
if (errno != ERANGE) {
    std::cout << "OK" << std::endl; // OK
}

» atof() — преобразует C-строку в число, имеющее тип double. Прототип функции:
#include <cstdlib>
double atof(const char *String);

Пример преобразования:
std::cout << std::atof("25") << std::endl; // 25
std::cout << std::atof("2.5") << std::endl; // 2.5
std::cout << std::atof("5.1str") << std::endl; // 5.1
std::cout << std::atof("5s10") << std::endl; // 5
std::cout << std::atof("str") << std::endl; // 0

» strtod() — преобразует C-строку в число, имеющее тип double. Прототип функции:
#include <cstdlib>
double strtod(const char *Str, char **EndPtr);

Пробельные символы в начале строки игнорируются. Считывание символов заканчивается на символе не являющемся записью вещественного числа. Указатель на этот символ доступен через параметр EndPtr, если он не равен 0. Пример преобразования:
char *p = 0;
std::cout << std::strtod(" \t\n 25", 0) << std::endl; // 25
std::cout << std::strtod("2.5", 0) << std::endl; // 2.5
std::cout << std::strtod("5.1str", 0) << std::endl; // 5.1
std::cout << std::strtod("14.5e5s10", &p)
    << std::endl; // 1.45e+006
std::cout << p << std::endl; // s10

Если в строке нет числа, то возвращается число 0. Если число выходит за диапазон значений для типа, то возвращается значение -HUGE_VAL или HUGE_VAL, а переменной errno присваивается значение ERANGE. Чтобы сбросить флаг ошибки следует присвоить значение 0 переменной errno.

Листинги на странице http://unicross.narod.ru/cpp/
Все рассмотренные в этом разделе функции позволяют преобразовать всю С-стрику в число. Если необходимо преобразовать отдельный символ (представляющий число) в соответствующее целое число от 0 до 9, то можно воспользоваться следующим кодом:

```cpp
char ch = '8';
int n = ch - '0'; // эквивалентно int n = 56 - 48;
std::cout << n << std::endl; // 8
```

В переменной типа `char` хранится код символа, а не сам символ. Символы от '0' до '9' в кодировке ASCII имеют коды от 48 до 57 соответственно. Следовательно, чтобы получить целое число от 0 до 9 достаточно вычесть из текущего символа код символа '0'. В качестве еще одного примера сохраним все числа, встречающиеся в С-строке, в целочисленном массиве (листинг 4.2).

### Листинг 4.2. Преобразование символов С-строки в отдельные числа

```cpp
#include <iostream>

int main() {
    char str[] = "0123456789";
    const int SIZE = 10;
    int arr[SIZE] = {0}, index = 0;
    for (char *p=str; *p; ++p) { // Перебираем строку
        if (*p >= '0' && *p <= '9') {
            arr[index] = *p - '0';
            ++index;
            if (index >= SIZE) break;
        }
    }
    for (int i=0; i<SIZE; ++i) { // Выводим значения
        std::cout << arr[i] << std::endl;
    }
    std::cin.get();
    return 0;
}
```

Преобразование числа в строку

Преобразовать значения элементарных типов в C-строку позволяет функция `sprintf()`. Прототип функции:

```c
#include <stdio.h>
int sprintf(char *DstBuf, const char *Format, ...);
```

В параметре `Format` указывается строка специального формата. Внутри этой строки можно указать обычные символы и спецификаторы формата, начинающиеся с символа `%`. Спецификаторы формата совпадают со спецификаторами, используемыми в функции `printf()` (см. раздел "Вывод данных в языке C"). Вместо спецификаторов формата подставляются значения, указанные в качестве параметров. Количество спецификаторов должно совпадать с количеством переданных параметров. Результат записывается в буфер, адрес которого передается в первом параметре (`DstBuf`). В качестве значения функция возвращает количество символов, записанных в символьный массив. Пример преобразования целого числа в C-строку:

```c
char buf[50];
int x = 100, count = 0;
count = std::sprintf(buf, "%d", x);
std::cout << buf << std::endl;  // 100
std::cout << count << std::endl;  // 3
```

Функция `sprintf()` не производит никакой проверки размера буфера, поэтому возможно переполнение буфера. В VC++ вместо функции `sprintf()` следует использовать функцию `sprintf_s()`. Прототип функции:

```c
#include <stdio.h>
int sprintf_s(char *DstBuf, size_t SizeInBytes, const char *Format, ...);
```

Параметры `DstBuf` и `Format` аналогичны параметрам функции `sprintf()`. В параметре `SizeInBytes` указывается размер буфера. В качестве значения функция возвращает количество символов, записанных в символьный массив. Пример преобразования вещественного числа в C-строку:

```c
char buf[50];
int count = 0;
double pi = 3.14159265359;
```

```c
count = sprintf_s(buf, 50, "%10.5f", pi);
std::cout << buf << std::endl;  // ' 3.14159'
std::cout << count << std::endl;  // 12
```

Листинги на странице http://unicross.narod.ru/cpp/
Генерация псевдослучайных чисел

Для генерации псевдослучайных чисел используются следующие функции:

→ rand() — генерирует псевдослучайное число от 0 до RAND_MAX. Прототип функции и определение макроса RAND_MAX:

```
#include <cstdlib>
int rand(void);
#define RAND_MAX 0x7fff
```

Пример:

```
std::cout << std::rand() << std::endl; // 41
std::cout << std::rand() << std::endl; // 18467
std::cout << RAND_MAX << std::endl;    // 32767
```

Чтобы получить случайное число от 0 до определенного значения, а не до RAND_MAX, следует использовать оператор % для получения остатка от деления. Пример получения числа от 0 до 10 включительно:

```
std::cout << std::rand() % 11 << std::endl;
```

→ srand() — настраивает генератор случайных чисел на новую последовательность. В качестве параметра обычно используется функция time() с нулевым указателем в качестве параметра, возвращающая количество секунд, прошедших с 1 января 1970 г. Прототипы функций:

```
#include <cstdlib>
void srand(unsigned int Seed);
#include <ctime>

time_t time(time_t *Time);
```

Пример:

```
std::srand(static_cast<unsigned int>( std::time(0) ));
std::cout << std::rand() << std::endl;
```

Если функция srand() вызвана с одним и тем же параметром, то будет генерироваться одна и та же последовательность псевдослучайных чисел:

```
std::srand(100);
std::cout << std::rand() << std::endl; // 365
std::srand(100);
```

Листинги на странице http://unicross.narod.ru/cpp/
В качестве примера создадим генератор паролей произвольной длины (листинг 4.3). Для этого добавляем в массив arr все разрешенные символы, а далее в цикле получаем случайный элемент из массива. Затем записываем символ, который содержит элемент массива, в итоговый символьный массив, адрес первого элемента которого передан в качестве первого параметра. В конец символьного массива вставляем нулевой символ. Следует учитывать, что символьный массив должен быть минимум на один символ больше, чем количество символов в пароле.

**Листинг 4.3. Генератор паролей**

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

void passwd_generator(char *pstr, int count_char);

int main() {
    char str[80];
    std::srand(static_cast<unsigned int> (std::time(0)));
    passwd_generator(str, 8);
    std::cout << str << std::endl; // Выведет примерно bjre7Vvue
    passwd_generator(str, 8);
    std::cout << str << std::endl; // CEvsXxCl
    passwd_generator(str, 10);
    std::cout << str << std::endl; // BTJ6rLPPvy
    std::cin.get();
    return 0;
}

void passwd_generator(char *pstr, int count_char) {
    const short SIZE = 60;
    char arr[SIZE] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
        'k', 'l', 'm', 'n', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z',
        'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
        'X', 'Y', 'Z', '1', '2', '3', '4', '5', '6', '7', '8', '9', '0'};
}
```

Листинги на странице http://unicross.narod.ru/cpp/
for (int i=0; i<count_char; ++i) {
    *pstr = arr[std::rand() % SIZE];
    ++pstr;
}
*pstr = '\0';
Программирование на C++
в Visual Studio® 2010 Express

Глава 5. Массивы

© Прохоренок Н.А., 2010 г., unicross@mail.ru

Этот DjVu-файл предоставляется КАК ЕСТЬ. Автор не несет никакой
ответственности за прямые или косвенные проблемы, связанные с
использованием данного файла. ВЫ ИСПОЛЬЗУЕТЕ ЕГО НА СВОЙ
СТРАХ И РИСК.

Все названия программных продуктов являются зарегистрированными
торговыми марками соответствующих фирм.

Запрещено:
- перепечатывать всю книгу или отдельные главы без письменного разрешения
Автора;
- декомпилировать данный файл и преобразовывать его в любой другой формат.
Массив — это нумерованный набор переменных одного типа. Переменная в массиве называется элементом, а ее позиция в массиве задается индексом. Все элементы массива располагаются в смежных ячейках памяти. Например, если объявлен массив из трех элементов, имеющих тип long int (занимает 4 байта), то адрес первого элемента массива (в 32-битной операционной системе) будет 0x0041040, второго — 0x0041044, а третьего — 0x0041048. Объем памяти (в байтах), занимаемый массивом, определяется так:

<Объем памяти> = sizeof (<Тип>) * <Количество элементов>

Объявление и инициализация массива

Объявление массива выглядит следующим образом:

<Tип> <Переменная>[<Количество элементов>];

Пример объявления массива из трех элементов, имеющих тип long int:

long arr[3];

При объявлении элементам массива можно присвоить начальные значения. Для этого после объявления указывается оператор =, а далее значения через запятую внутри фигурных скобок. После закрывающей фигурной скобки обязательно указывается точка с запятой. Пример инициализации массива:

long arr[3] = {10, 20, 30};

Количество значений внутри фигурных скобок может быть меньше количества элементов массива. В этом случае значения присваиваются соответствующим элементам с начала массива. Пример:

long arr[3] = {10, 20};

В этом примере первому элементу массива присваивается значение 10, второму — значение 20, а третьему элементу будет присвоено значение 0.

Присвоить всем элементам массива значение 0 можно так:

int arr[15] = {0};

Если при объявлении массива указываются начальные значения, то количество элементов внутри квадратных скобок можно не указывать. Размер будет соответствовать количеству значений внутри фигурных скобок. Пример:

long arr[] = {10, 20, 30};

Если при объявлении массива начальные значения не указаны, то:

Листинги на странице http://unicross.narod.ru/cpp/
→ элементам глобальных массивов автоматически присваивается значение 0;
→ элементы локальных массивов будут содержать произвольные значения, так называемый "мусор".

Получение и изменение значения элемента массива

Обращение к элементам массива осуществляется с помощью квадратных скобок, в которых указывается индекс элемента. Обратите внимание на то, что нумерация элементов массива начинается с 0, а не с 1, поэтому первый элемент имеет индекс 0. С помощью индексов можно присвоить начальные значения элементам массива уже после объявления:

```c++
long arr[3];
arr[0] = 10; // Первый элемент имеет индекс 0!!!
arr[1] = 20; // Второй элемент
arr[2] = 30; // Третий элемент
```

Следует учитывать, что проверка выхода указанного индекса за пределы диапазона на этапе компиляции не производится. Таким образом, можно перезаписать значение в смежной ячейке памяти и тем самым нарушить работоспособность программы или даже повредить операционную систему. Помните, что контроль корректности индекса входит в обязанности программиста.

После определения массива выделяется необходимый размер памяти, а в переменной сохраняется адрес первого элемента массива. При указании индекса внутри квадратных скобок производится вычисление адреса соответствующего элемента массива. Зная адрес элемента массива можно получить значение или перезаписать его. Иными словами, с элементами массива можно производить такие же операции, как и с обычными переменными. Пример:

```c++
long arr[3] = {10, 20, 30};
long x = 0;
x = arr[1] + 12;
arr[2] = x - arr[2];
std::cout << x << std::endl; // 32
std::cout << arr[2] << std::endl; // 2
```

Листинги на странице http://unicross.narod.ru/cpp/
Перебор элементов массива

Для перебора элементов массива удобно использовать цикл for. В первом параметре переменной-счетчику присваивается значение 0 (элементы массива нумеруются с нуля), условием продолжения является значение переменной-счетчика меньше количества элементов массива. В третьем параметре указывается приращение на единицу на каждой итерации цикла. Внутри цикла доступ к элементу осуществляется с помощью квадратных скобок, внутри которых указывается переменная-счетчик. Пронумеруем все элементы массива, а затем выводим все значения в прямом и обратном порядке:

```cpp
const short ARR_SIZE = 20;
int arr[ARR_SIZE];
// Нумеруем все элементы массива
for (int i=0, j=1; i<ARR_SIZE; ++i, ++j) {
    arr[i] = j;
}
// Выводим значения в прямом порядке
for (int i=0; i<ARR_SIZE; ++i) {
    std::cout << arr[i] << std::endl;
}
std::cout << "---------------------" << std::endl;
// Выводим значения в обратном порядке
for (int j=ARR_SIZE-1; j>=0; --j) {
    std::cout << arr[j] << std::endl;
}
```

В этом примере мы объявили количество элементов массива как постоянную величину (константа ARR_SIZE). Это очень удобно, так как размер массива приходится указывать при каждом переборе массива. Если количество элементов указывать в виде числа, то при изменении размера массива придется вручную изменять все значения. При использовании константы количество элементов достаточно будет изменить только в одном месте.

Чтобы получить количество элементов массива из программы, необходимо общий размер массива в байтах разделить на размер типа. Получить эти размеры можно с помощью оператора sizeof. Пример определения количества элементов массива:

```cpp
int arr[15] = {0};
std::cout << sizeof arr / sizeof (int) << std::endl; // 15
```

Листинги на странице http://unicross.narod.ru/cpp/
Цикл for всегда можно заменить циклом while. В качестве примера пронумеруем элементы в обратном порядке, а затем выведем все значения:

```c
const short ARR_SIZE = 20;
int arr[ARR_SIZE];
// Нумеруем все элементы
int i = 0, j = ARR_SIZE;
while (i<ARR_SIZE) {
    arr[i] = j;
    ++i;
    --j;
}
// Выводим значения массива
i = 0;
while (i<ARR_SIZE) {
    std::cout << arr[i] << std::endl;
    ++i;
}
```

### Доступ к элементам массива с помощью указателя

После определения массива в переменной сохраняется адрес первого элемента. Иными словами, название переменной является указателем, который ссылается на первый элемент массива. Поэтому доступ к элементу массива может осуществляться как по индексу, указанному внутри квадратных скобок, так и с использованием адресной арифметики. Например, следующие инструкции вывода являются эквивалентными:

```c
int arr[3] = {1, 2, 3};
std::cout << arr[1] << std::endl; // 2
std::cout << *(arr + 1) << std::endl; // 2
std::cout << *(1 + arr) << std::endl; // 2
std::cout << 1[arr] << std::endl; // 2
```

Последняя инструкция может показаться странной. Однако, если учесть, что выражение `1[arr]` воспринимается компилятором как `*(1 + arr)`, то все встанет на свои места.

При указании индекса внутри квадратных скобок, каждый раз производится вычисление адреса соответствующего элемента массива. Чтобы сделать процесс доступа к элементу массива более эффективным объявляют указатель и присваивают ему адрес первого элемента. Далее для доступа к элементу просто перемещают указатель на нужную позицию.

указатель на соответствующий элемент. Пример объявления указателя и присваивания ему адреса первого элемента массива:

```cpp
int *p = 0;
int arr[3] = {1, 2, 3};
p = arr;
```
Обратите внимание на то, что перед названием массива не указывается оператор *, так как название переменной содержит адрес первого элемента. Если использовать оператор &, то необходимо дополнительно указать индекс внутри квадратных скобок:

```cpp
p = &arr[0]; // Эквивалентно: p = arr;
```
Таким же образом можно присвоить указателю адрес произвольного элемента массива, например, третьего:

```cpp
p = &arr[2]; // Указатель на третий элемент массива
```
Чтобы получить значение элемента, на который ссылается указатель, необходимо произвести операцию разыменования. Для этого перед названием указателя добавляется оператор *. Пример:

```cpp
std::cout << *p << std::endl;
```
Указатели часто используются для перебора элементов массива. В качестве примера создадим массив из трех элементов, а затем выводим значения с помощью цикла for:

```cpp
const short SIZE = 3;
int *p = 0, arr[SIZE] = {1, 2, 3};
// Устанавливаем указатель на первый элемент массива
p = arr; // Оператор & не указывается!!!
for (int i=0; i<SIZE; ++i) {
    std::cout << *p << std::endl;
    ++p; // Перемещаем указатель на следующий элемент
}
p = arr; // Восстанавливаем положение указателя
```
В первой строке объявляется константа SIZE, в которой сохраняется количество элементов в массиве. Если массив используется часто, то лучше сохранить его размер как константу, так как количество элементов нужно будет указывать при каждом переборе массива. Если в каждом цикле указывать конкретное число, то при изменении размера массива придется вручную вносить изменения во всех циклах. При объявлении константы достаточно будет изменить ее значение один раз при инициализации.

Листинги на странице http://unicross.narod.ru/cpp/
В следующей строке объявляется указатель на тип `int` и массив. Количество элементов массива задается константой `SIZE`. При объявлении массив инициализируется начальными значениями.

Внутри цикла `for` выводится значение элемента на который ссылается указатель, а затем значение указателя увеличивается на единицу (`++p;`). Обратите внимание на то, что изменяется адрес, а не значение элемента массива. При увеличении значения указателя используются правила адресной арифметики, а не правила обычной арифметики. Увеличение значения указателя на единицу означает, что значение будет увеличено на размер типа. Например, если тип `int` занимает 4 байта, то при увеличении значения на единицу указатель вместо адреса `0x0012ff30` будет содержать адрес `0x0012ff34`. Значение увеличилось на 4, а не на 1. В нашем примере вместо двух инструкций внутри цикла можно использовать одну:

```cpp
std::cout << *(p++) << std::endl;
```

Выражение `p++` возвращает текущий адрес, а затем увеличивает его на единицу. Символ `*` позволяет получить доступ к значению элемента по указанному адресу. Последовательность выполнения соответствует следующей расстановке скобок:

```cpp
std::cout << *(p++) << std::endl;
```

Если скобки расставить так:

```cpp
std::cout << (*p)++ << std::endl;
```

tо, вначале будет получен доступ к элементу массива и выведено его текущее значение, а затем произведено увеличение значения элемента массива. Перемещение указателя на следующий элемент не производится.

Указатель можно использовать в качестве переменной-счетчика в цикле `for`. В этом случае начальным значением будет адрес первого элемента массива, а условием продолжения — адрес меньше адреса первого элемента плюс количество элементов. Приращение осуществляется аналогично обычному, только вместо обычной арифметики применяется адресная арифметика. Пример:

```cpp
const short SIZE = 3;
int arr[SIZE] = {1, 2, 3};
for (int *p=arr; p<arr+SIZE; ++p) {
    std::cout << *p << std::endl;
}
```

Вывести все значения массива с помощью цикла `while` и указателя можно так:

```cpp
const short SIZE = 3;
```

int *p = 0, i = SIZE, arr[SIZE] = {1, 2, 3};
p = arr;
while (i-- > 0) {
    std::cout << *p++ << std::endl;
}
p = arr;

Массивы указателей

Указатели можно сохранять в массиве. При объявлении массива указателей
используется следующий синтаксис:
<Tип> *<Название массива>[<Количество элементов>];

Пример использования массива указателей:
int *p[3]; // Массив указателей из трех элементов
int x = 10, y = 20, z = 30;
p[0] = &x;
p[1] = &y;
p[2] = &z;
std::cout << *p[0] << std::endl; // 10
std::cout << *p[1] << std::endl; // 20
std::cout << *p[2] << std::endl; // 30

Динамические массивы

При объявлении массива необходимо задать точное количество элементов. На основе
этой информации при запуске программы автоматически выделяется необходимый
объем памяти. Иными словами, размер массива необходимо знать до выполнения
программы. Во время выполнения программы увеличить размер существующего
массива нельзя.

Чтобы произвести увеличение массива во время выполнения программы необходимо
выделить достаточный объем памяти с помощью оператора new, перенести
существующие элементы, а лишь затем добавить новые элементы. Управление
dинамической памятью полностью лежит на плечах программиста, поэтому после
завершения работы с памятью необходимо самим возвратить память операционной
системе с помощью оператора delete. Если память не возвратить операционной
системе, то участок памяти станет недоступным для дальнейшего использования.

Листинги на странице http://unicross.narod.ru/cpp/
Подобные ситуации приводят к утечке памяти. Сделать эти участи опять доступными можно только после перезагрузки компьютера.

Выделение памяти под массив производится следующим образом:

```cpp
<Указатель> = new <Тип данных>[<Количество элементов>];
```

Освободить выделенную память можно так:

```cpp
delete [] <Указатель>;
```

Обратите внимание на то, что при освобождении памяти количество элементов не указывается.

При выделении памяти может возникнуть ситуация нехватки памяти. В ранних версиях C++ в этом случае возвращался нулевой указатель. Согласно новой версии стандарта в случае ошибки оператор `new` должен возбуждать исключение `bad_alloc` (объект исклюения определён в файле `new`). Обработать это исключение можно с помощью конструкции `try...catch`. Пример выделения памяти с обработкой исключения:

```cpp
#include <new>

// ... Фрагмент опущен ...

const short SIZE = 3;       // Размер массива
int *p = 0;                  // Создаем указатель

try {
    p = new int[SIZE];       // Выделяем память
}

catch (std::bad_alloc err) {
    std::cout << "Error" << std::endl;
    std::exit (1);           // Выходим при ошибке
}

p[0] = 1;                     // Пользуемся памятью
p[1] = 2;
p[2] = 3;
for (int i=0; i<SIZE; ++i) {
    std::cout << p[i] << std::endl;
}

delete [] p;                 // Возвращаем память
p = 0;                        // Обнуляем указатель
```

Выделение памяти производится внутри блока `try`. Если при этом возникнет исключение `bad_alloc`, то управление будет передано в блок `catch`. После выполнения инструкций в блоке `catch` управление передается инструкции,

Листинги на странице http://unicross.narod.ru/cpp/
расположенной сразу после блока. Иными словами, компилятор считает, что вы обработали исключение и можно продолжить выполнение программы. Следует учитывать, что пользоваться указателем после обработки нельзя, поэтому внутри блока catch обычно выводят сообщение об ошибке и завершают выполнение программы. Если исключение не обработано, то программа аварийно завершится. Если исключение не возникло, то инструкции внутри блока catch не выполняются.

Обратите внимание на то, что определение указателя производится вне блока try. Если определение разместить внутри блока, то область видимости переменной будет ограничена этим блоком. После выхода из блока переменная автоматически уничтожается, а выделенная память операционной системе не возвращается. Поэтому, определение указателя должно находиться перед блоком, а не внутри него.

После использования оператора delete указатель по-прежнему будет содержать прежний адрес, поэтому указатель принято обнулять.

Многомерные массивы

Массивы в языке C++ могут быть многомерными. Объявление многомерного массива имеет следующий формат:

```cpp
<Tип> <Переменная>[(<Количество элементов1>)]...[(<Количество элементовN>)];
```

На практике наиболее часто используются двухмерные массивы, позволяющие хранить значения ячеек таблицы, содержащей определенное количество строк и столбцов. Объявление двухмерного массива выглядит так:

```cpp
<Tип> <Переменная>[(<Количество строк>)][(<Количество столбцов>)];
```

Пример объявления двухмерного массива, содержащего две строки и четыре столбца:

```cpp
int arr[2][4]; // Две строки из 4-х элементов каждая
```

Все элементы двухмерного массива располагаются в памяти друг за другом. Вначале элементы первой строки, затем второй и т. д. При инициализации двухмерного массива элементы указываются внутри фигурных скобок через запятую. Пример:

```cpp
int arr[2][4] = {
    1, 2, 3, 4,
    5, 6, 7, 8
};
```

Чтобы сделать процесс инициализации наглядным мы расположили элементы на отдельных строках. Количество элементов на строке совпадает с количеством столбцов.

в массиве. Если элементы одной строки разместить внутри фигурных скобок, то процесс станет еще нагляднее:

```cpp
int arr[2][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8}
};
```
Если при объявлении производится инициализация, то количество строк можно не указывать, оно будет определено автоматически:

```cpp
int arr[][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8}
};
```

Получить или задать значение элемента можно указав два индекса (не забывайте, что нумерация начинается с нуля):

```cpp
std::cout << arr[0][0] << std::endl; // 1
std::cout << arr[1][0] << std::endl; // 5
std::cout << arr[1][3] << std::endl; // 8
```
Чтобы вывести все значения массива необходимо использовать вложенные циклы. В качестве примера пронумуем все элементы массива, а затем выведем все значения:

```cpp
const short ARR_ROWS = 4;
const short ARR_COLS = 5;
int i, j, n = 1, arr[ARR_ROWS][ARR_COLS];
```
```
Нумеруем все элементы массива
for (i=0; i<ARR_ROWS; ++i) {
    for (j=0; j<ARR_COLS; ++j) {
        arr[i][j] = n;
        ++n;
    }
}
```
```
// Выводим значения
for (i=0; i<ARR_ROWS; ++i) {
    for (j=0; j<ARR_COLS; ++j) {
        std::cout.width(3); // Ширина поля
        std::cout << arr[i][j];
    }
}
```

Листинги на странице http://unicross.narod.ru/cpp/
std::cout << std::endl;
}

Все элементы многомерного массива располагаются в памяти друг за другом. Поэтому для перебора массива можно использовать указатель. В качестве примера выведем все элементы двухмерного массива с помощью цикла for и указателя:

```cpp
cost short ROWS = 4, COLS = 5;
int arr[ROWS][COLS] = {
    {1, 2, 3, 4, 5},
    {6, 7, 8, 9, 10},
    {11, 12, 13, 14, 15},
    {16, 17, 18, 19, 20}
};
for (int *p=arr[0]; p<arr[0]+ROWS*COLS; ++p) {
    std::cout << *p << std::endl;
}
```

ПОИСК МИНИМАЛЬНОГО И МАКСИМАЛЬНОГО ЗНАЧЕНИЯ

Чтобы в массиве найти минимальное значение следует присвоить переменной min значение первого элемента массива, а затем произвести сравнение с остальными элементами. Если значение текущего элемента меньше значения переменной min, то присваиваем значение текущего элемента переменной min. Чтобы найти максимальное значение следует присвоить переменной max значение первого элемента массива, а затем произвести сравнение с остальными элементами. Если значение текущего элемента больше значения переменной max, то присваиваем значение текущего элемента переменной max. Пример поиска минимального и максимального значения приведен в листинге 5.1.

Листинг 5.1. Поиск минимального и максимального значения

```cpp
#include <iostream>

int main() {
    const short ARR_SIZE = 5;
    int arr[ARR_SIZE] = {2, 5, 6, 1, 3};
    int min = arr[0], max = arr[0];
    for (int i=1; i<ARR_SIZE; ++i) {
        Листинги на странице http://unicross.narod.ru/cpp/
if (min > arr[i]) min = arr[i];
if (max < arr[i]) max = arr[i];
}
std::cout << "min = " << min << std::endl;
std::cout << "max = " << max << std::endl;
std::cin.get();
return 0;
}

Сортировка массива

Сортировка массива — это упорядочивание элементов по возрастанию или убыванию значений. Сортировка применяется при выводе значений, а также при подготовке массива к частому поиску значений. Поиск по отсортированному массиву производится гораздо быстрее, так как не нужно каждый раз просматривать все значения массива.

Для упорядочивания элементов массива очень часто применяется метод, называемый пузырьковой сортировкой. При этом методе наименьшее значение как бы "всплывает" в начало массива, а наибольшее значение "погружается" в конец массива. Сортировка выполняется в несколько проходов. При каждом проходе последовательно сравниваются значения двух элементов, которые расположены рядом. Если значение первого элемента больше второго, то значения элементов меняются местами. Для сортировки массива из пяти элементов необходимо максимум четыре прохода и десять сравнений. Если после прохода не было ни одной перестановки, то сортировку можно прервать. В этом случае для сортировки ранее уже отсортированного массива нужен всего один проход. Последовательность сравнения элементов массива при пузырьковой сортировке по возрастанию показана в табл. 5.1, а код программы приведен в листинге 5.2.
### Таблица 5.1. Последовательность сравнения элементов при пузырьковой сортировке

<table>
<thead>
<tr>
<th>Проход</th>
<th>Описание</th>
<th>Значения элементов массива</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Начальные значения</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Сравниваются arr[0] и arr[1]</td>
<td>1</td>
</tr>
</tbody>
</table>

### Листинг 5.2. Пузырьковая сортировка по возрастанию

```cpp
#include <iostream>

int main() {
    const short ARR_SIZE = 5;
    int arr[ARR_SIZE] = {10, 5, 6, 1, 3};
    int j = 0, tmp = 0, k = ARR_SIZE - 2;
    bool is_swap = false;
    for (int i = 0; i <= k; ++i) {
        is_swap = false;
        for (j = k; j >= i; --j) {
            if (arr[j] > arr[j+1]) {
                tmp = arr[j+1];
                swap(arr[j], arr[j+1]);
            }
        }
    }
}
```

arr[j+1] = arr[j];
arr[j] = tmp;
is_swap = true;
}
}
// Если перестановок не было, то выходим
if (!is_swap) break;

// Выводим отсортированный массив
for (int i=0; i<ARR_SIZE; ++i) {
    std::cout << arr[i] << std::endl;
}
std::cin.get();
return 0;
}

В качестве еще одного примера произведем сортировку по убыванию (листинг 5.3). Чтобы пример был более полезным изменим направление проходов.

### Листинг 5.3. Пузырьковая сортировка по убыванию

```
#include <iostream>

int main() {
    const short ARR_SIZE = 5;
    int arr[ARR_SIZE] = {10, 5, 6, 1, 3};
    int j = 0, tmp = 0;
    bool is_swap = false;
    for (int i=ARR_SIZE-1; i>=1; --i) {
        is_swap = false;
        for (j=0; j<i; ++j) {
            if (arr[j] < arr[j+1]) {
                tmp = arr[j+1];
                arr[j+1] = arr[j];
                arr[j] = tmp;
                is_swap = true;
            }
        }
    }
    std::cout << "Сортированый массив:"
    for (int i=0; i<ARR_SIZE; ++i) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
}    
    if (!is_swap) break;
}
for (int i=0; i<ARR_SIZE; ++i) {
    std::cout << arr[i] << std::endl;
}
std::cin.get();
return 0;
}

Для сортировки массива можно также воспользоваться стандартной функцией qsort(). Прототип функции:
#include <cstdlib>
void qsort(void *Base, size_t NumOfElements, size_t SizeOfElements, 
           int (*PtFuncCompare)(const void *, const void *));
В параметре Base указывается адрес первого элемента массива, в параметре NumOfElements — количество элементов массива, а в параметре SizeOfElements — размер каждого элемента в байтах. Адрес пользовательской функции (указывается название функции без круглых скобок и параметров), внутри которой производится сравнение двух элементов, передается в последнем параметре. Прототип пользовательской функции сравнения должен выглядеть так:
int <Название функции>(const void *arg1, const void *arg2);
При сортировке по возрастанию функция должна возвращать отрицательное значение, если arg1 меньше arg2, положительное значение, если arg1 больше arg2, или 0, если значения равны. Внутри функции необходимо выполнить приведение указателя void * к определенному типу. Пример использования функции qsort() приведен в листинге 5.4.

Листинг 5.4. Сортировка массива с помощью функции qsort()

#include <iostream>
#include <cstdlib>

int mysort(const void *arg1, const void *arg2);

int main() {
    const short ARR_SIZE = 5;

    Листинги на странице http://unicross.narod.ru/cpp/
```cpp
int arr[ARR_SIZE] = {10, 5, 6, 1, 3};
std::qsort(arr, ARR_SIZE, sizeof(int), mysort);
for (int i=0; i<ARR_SIZE; ++i) {
    std::cout << arr[i] << std::endl;
}
std::cin.get();
return 0;
}
int mysort(const void *arg1, const void *arg2) {
    return *(int *)arg1 - *(int *)arg2;
}
Чтобы произвести сортировку по убыванию, достаточно поменять значения местами:
int mysort(const void *arg1, const void *arg2) {
    return *(int *)arg2 - *(int *)arg1;
}

Проверка наличия значения в массиве
Если массив не отсортирован, то проверка наличия значения в массиве сводится к
перебиранию всех элементов от начала до конца. При наличии первого вхождения
могу прервать поиск. Пример поиска первого вхождения:
const short ARR_SIZE = 5;
int arr[ARR_SIZE] = {10, 5, 6, 1, 3}, index = -1;
int search_key = 6;
for (int i=0; i<ARR_SIZE; ++i) {
    if (arr[i] == search_key) {
        index = i;
        break;
    }
}
if (index != -1) {
    std::cout << "Yes" << std::endl;
    std::cout << "index = " << index << std::endl;
}
else std::cout << "No" << std::endl;
```

Листинги на странице http://unicross.narod.ru/cpp/
Если значение находится в начале массива, то поиск будет произведен достаточно быстро, но если значение расположено в конце массива, то придется просматривать весь массив. Если массив состоит из 100 000 элементов, то нужно будет сделать 100 000 сравнений.

Поиск по отсортированному массиву производится гораздо быстрее, так как не нужно каждый раз просматривать все значения массива. Наиболее часто применяется бинарный поиск (листинг 5.5), при котором массив делится пополам и производится сравнение значения элемента, расположенного в середине массива, с искомым значением. Если искомое значение меньше значения элемента, то пополам делится первая половина массива, а если больше, то пополам делится вторая половина. Далее таким же образом производится деление оставшейся части массива. Поиск заканчивается когда будет найдено первое совпадение или когда начальный индекс станет больше конечного индекса. Таким образом, на каждой итерации отбрасывается половина оставшихся элементов. Поиск значения, которое расположено в конце массива, состоящего из 100 000 элементов, будет произведен всего за 17 шагов. Однако, если поиск производится один раз, то затраты на сортировку сведут на нет все преимущества бинарного поиска. В этом случае прямой перебор элементов может стать эффективнее.

**Листинг 5.5. Бинарный поиск в отсортированном массиве**

```cpp
#include <iostream>
#include <cstdlib>

int mysort(const void *arg1, const void *arg2);
int mysearch(int key, const int *arr, int count);

int main() {
    const short ARR_SIZE = 5;
    int arr[ARR_SIZE] = {10, 5, 6, 1, 3};
    // Сортируем по возрастанию
    std::qsort(arr, ARR_SIZE, sizeof(int), mysort);
    // Производим поиск
    int index = mysearch(6, arr, ARR_SIZE);
    if (index != -1) {
        std::cout << "Yes" << std::endl;
        std::cout << "index = " << index << std::endl;
    }
}
```

Листинги на странице http://unicross.narod.ru/cpp/
}  
else std::cout << "No" << std::endl; 
std::cin.get();  
return 0;  
}  
// Сортировка по возрастанию  
int mysort(const void *arg1, const void *arg2) {  
    return *(int *)arg1 - *(int *)arg2;  
}  
// Бинарный поиск в отсортированном массиве  
int mysearch(int key, const int *arr, int count) {  
    if (count < 1 || !arr) return -1;  
    int start = 0, end = count - 1, i = 0;  
    while (start <= end) {  
        i = (start + end) / 2;  
        if (arr[i] == key) return i;  
        else if (arr[i] < key) start = i + 1;  
        else if (arr[i] > key) end = i - 1;  
    }  
    return -1;  
}  

Для проверки наличия элемента в массиве можно также воспользоваться стандартной функцией bsearch(). Прототип функции:  
#include <cstdlib>  
void *bsearch(const void *Key, const void *Base, 
              size_t NumOfElements, size_t SizeOfElements, 
              int (*PtFuncCompare)(const void *, const void *));  

В параметре Key передается адрес переменной, в которой хранится искомое значение, в параметре Base указывается адрес первого элемента массива, в параметре NumOfElements — количество элементов массива, а в параметре SizeOfElements — размер каждого элемента в байтах. Адрес пользовательской функции (указываетя название функции без круглых скобок и параметров), внутри которой производится сравнение двух элементов, передается в последнем параметре. Прототип пользовательской функции сравнения должен выглядеть так:  
int <Название функции>(const void *arg1, const void *arg2);  

Листинги на странице http://unicross.narod.ru/cpp/
Функция должна возвращать отрицательное значение, если arg1 меньше arg2, положительное значение, если arg1 больше arg2, или 0, если значения равны. Внутри функции необходимо выполнить приведение указателя void * к определенному типу. Кроме того, необходимо выполнить приведение указателя при получении результата. Если значение не найдено, то возвращается нулевой указатель, в противном случае указатель ссылается на найденный элемент. Пример использования функции bsearch() приведен в листинге 5.6.

### Листинг 5.6. Проверка наличия значения в массиве с помощью функции bsearch()

```cpp
#include <iostream>
#include <cstdlib>

int mysort(const void *arg1, const void *arg2);
int mysearch(const void *arg1, const void *arg2);

int main() {
    const short ARR_SIZE = 5;
    int arr[ARR_SIZE] = {10, 5, 6, 1, 3}, search_key = 6;
    // Сортируем по возрастанию
    std::qsort(arr, ARR_SIZE, sizeof(int), mysort);
    // Производим поиск
    int *p = (int *)std::bsearch(&search_key, arr, ARR_SIZE,
                                  sizeof(int), mysearch);
    if (p) std::cout << "Yes" << std::endl;
    else std::cout << "No" << std::endl;
    std::cin.get();
    return 0;
}

// Сортировка по возрастанию
int mysort(const void *arg1, const void *arg2) {
    return *(int *)arg1 - *(int *)arg2;
}

// Сравнение при поиске
int mysearch(const void *arg1, const void *arg2) {
    return *(int *)arg1 - *(int *)arg2;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
Копирование элементов из одного массива в другой

Для копирования элементов из одного массива в другой используются следующие функции:

- `memcpy()` — копирует первые `size` байтов из массива `Src` в массив `Dst`. В качестве значения функция возвращает указатель на массив `Dst`. Если массив `Src` длиннее массива `Dst`, то произойдет переполнение буфера. Если указатели пересекаются, то поведение функции непредсказуемо. Прототип функции:

```
#include <cstring>
void *memcpy(void *Dst, const void *Src, size_t Size);
```

Пример использования функции:

```cpp
class Solution {
public:
    void solution() {
        // Constant values:
        const short ARR1_SIZE = 5, ARR2_SIZE = 3;
        const int arr1[ARR1_SIZE] = {0}, arr2[ARR2_SIZE] = {1, 2, 3};
        int *p = 0, i = 0;

        // Copy all elements of array arr2
        p = (int *)std::memcpy(arr1, arr2, sizeof(arr2));
        for (i = 0; i < ARR1_SIZE; ++i) {
            std::cout << arr1[i] << " ";
        } // 1 2 3 0 0
        std::cout << std::endl;
        if (!p) std::exit(1);
        // Move the pointer to the fourth element of array arr2
        p += 3;
        // Copy only two first elements of array arr2
        std::memcpy(p, arr2, 2 * sizeof(int));
        for (i = 0; i < ARR1_SIZE; ++i) {
            std::cout << arr1[i] << " ";
        } // 1 2 3 1 2
        std::cout << std::endl;
    }
};
```

Листинги на странице http://unicross.narod.ru/cpp/
В VC++ вместо функции `memcpy()` можно использовать функцию `memcpy_s()`.
Прототип функции:
```c
#include <cstring>
errno_t memcpy_s(void *Dst, rsize_t DstSize, const void *Src,       
rsize_t MaxCount);
```
Функция `memcpy_s()` копирует первые `MaxCount` байтов из массива `Src` в массив `Dst`. В параметре `DstSize` указывается размер массива `Dst` в байтах. Если копирование прошло успешно функция возвращает значение 0. Пример:
```c
const short ARR1_SIZE = 5, ARR2_SIZE = 3;
int arr1[ARR1_SIZE] = {0}, arr2[ARR2_SIZE] = {1, 2, 3};
memcpy_s(arr1, sizeof(arr1), arr2, 2 * sizeof(int));
for (int i=0; i<ARR1_SIZE; ++i) {
    std::cout << arr1[i] << " ";
} // 1 2 0 0 0
std::cout << std::endl;
```

Функция `memmove()` — копирует первые `Size` байтов из массива `Src` в массив `Dst`. В качестве значения функция возвращает указатель на массив `Dst`. Если массив `src` длиннее массива `Dst`, то произойдет переполнение буфера. Основное отличие функции `memmove()` от `memcpy()` в выполнении корректных действий при пересечении указателей. Прототип функции:
```c
#include <cstring>
void *memmove(void *Dst, const void *Src, size_t Size);
```
Пример:
```c
const short ARR_SIZE = 5;
int arr[ARR_SIZE] = {1, 2, 3, 4, 5}, *p = 0;
p = arr + 2;
// Указатели пересекаются
std::memmove(p, arr, 3 * sizeof(int));
for (int i=0; i<ARR_SIZE; ++i) {
    std::cout << arr[i] << " ";
} // 1 2 1 2 3 - копирование произведено корректно
```

Листинги на странице http://unicross.narod.ru/cpp/
std::cout << std::endl;

В VC++ вместо функции `memmove()` можно использовать функцию `memmove_s()`. Прототип функции:

```c
#include <cstring>

errno_t memmove_s(void *Dst, size_t DstSize, const void *Src,
                   size_t MaxCount);
```

Функция `memmove_s` копирует первые `MaxCount` байтов из массива `Src` в массив `Dst`. В параметре `DstSize` указывается размер массива `Dst` в байтах. Если копирование прошло успешно функция возвращает значение 0. Пример:

```cpp
const short ARR_SIZE = 5;
int arr[ARR_SIZE] = {1, 2, 3, 4, 5}, *p = 0;
p = arr + 2;
memmove_s(p, sizeof arr, arr, 3 * sizeof(int));
for (int i=0; i<ARR_SIZE; ++i) {
    std::cout << arr[i] << " ";
} // 1 2 1 2 3
std::cout << std::endl;
```

### Сравнение массивов

Для сравнения массивов предназначена функция `memcmp()`. Функция `memcmp()` сравнивает первые `size` байтов массивов `Buf1` и `Buf2`. В качестве значения функция возвращает:

- отрицательное число — если `Buf1` меньше `Buf2`;
- 0 — если массивы равны;
- положительное число — если `Buf1` больше `Buf2`.

Прототип функции:

```c
#include <cstring>

int memcmp(const void *Buf1, const void *Buf2, size_t Size);
```

Пример:

```cpp
int arr1[3] = {1, 2, 3}, arr2[3] = {1, 2, 3}, x = 0;
x = std::memcmp(arr1, arr2, sizeof arr2);
```

Функция `memcmp()` производит сравнение с учетом регистра символов. Если необходимо произвести сравнение без учета символов, то в VC++ можно воспользоваться функцией `_memicmp()`. Для сравнения русских букв следует настроить локаль. Прототип функции:

```
#include <cstring>

int _memicmp(const void *Buf1, const void *Buf2, size_t Size);
```

Предназначение параметров и возвращаемое значение такое же, как у функции `memcmp()`. Пример использования функции:

```
std::setlocale(LC_ALL, "Russian_Russia.1251"); // Настройка локали
char str1[] = "абв", str2[] = "АБВ";
int x = 0;
x = _memicmp(str1, str2, sizeof str2);
std::cout << x << std::endl; // 0
x = std::memcmp(str1, str2, sizeof str2);
std::cout << x << std::endl; // 1
```
Программирование на C++ в Visual Studio® 2010 Express

Глава 6.
Символы и строки

© Прохоренок Н.А., 2010 г., unicross@mail.ru

Этот DjVu-файл предоставляется КАК ЕСТЬ. Автор не несет никакой ответственности за прямые или косвенные проблемы, связанные с использованием данного файла. ВЫ ИСПОЛЬЗУЕТЕ ЕГО НА СВОЙ СТРАХ И РИСК.

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Запрещено:
- перепечатывать всю книгу или отдельные главы без письменного разрешения Автора;
- декомпилировать данный файл и преобразовывать его в любой другой формат.
В языке C++ доступны два типа однобайтовых строк: строка, применяемая в языке C (поэтому часто называемая C-строкой), и класс string. C-строка является массивом символов (тип char), последний элемент которого содержит нулевой символ (\0). Обратите внимание на то, что нулевой символ (нулевой байт) не имеет никакого отношения к числу 0. Коды этих символов разные. Класс string предоставляет более удобный интерфейс доступа, который избавляет программиста от необходимости следить за размером символьного массива и предоставляет множество методов для обработки строки. При необходимости можно преобразовать объект класса string в C-строку.

Объявление и инициализация отдельного символа

Для хранения символа используется тип char. Переменной, имеющей тип char, можно присвоить числовое значение (код символа) или указать символ внутри апострофов. Обратите внимание на то, что использовать кавычки нельзя, так как в этом случае вместо одного символа будет два: собственно сам символ плюс нулевой символ. Пример:

cchar ch1, ch2;
ch1 = 119; // Буква w
ch2 = 'w'; // Буква w

Внутри апострофов можно указать специальные символы. Специальные символы — это комбинации знаков, обозначающих служебные или непечатаемые символы. Перечислим специальные символы, доступные в C++:

- \0 — нулевой символ;
- \n — перевод строки;
- \r — возврат каретки;
- \t — горизонтальная табуляция;
- \v — вертикальная табуляция;
- \a — звуковой сигнал;
- \b — возврат на один символ;
- \f — перевод формата;
- \' — апостроф;

Листинги на странице http://unicross.narod.ru/cpp/
→ \" — кавычка;
→ \? — знак вопроса;
→ \\ — обратная косая черта;
→ \n — восьмеричное значение N;
→ \xN — шестнадцатеричное значение N.

Пример указания восьмеричного и шестнадцатеричного значений:

```c
char ch1, ch2;
ch1 = '\167'; // Буква w (восьмеричное значение)
ch2 = '\x77'; // Буква w (шестнадцатеричное значение)
```

По умолчанию тип char является знаковым и позволяет хранить диапазон значений от -128 до 127. Если перед типом указано ключевое слово unsigned, то диапазон будет от 0 до 255. Когда переменной присваивается символ внутри апострофов, он автоматически преобразуется в соответствующий целочисленный код.

Тип char занимает в памяти один байт (восемь бит). Если тип является знаковым, то старший бит содержит признак знака: 0 соответствует положительному числу, а 1 — отрицательному. Если тип является беззнаковым, то признак знака не используется. Это следует учитывать при преобразовании знакового типа в беззнаковый, так как старший бит станет причиной больших значений:

```c
char ch1 = -1;
unsigned char ch2 = (unsigned char)ch1;
std::cout << (int)ch2 << std::endl; // 255
```

Символы, имеющие код меньше 128 (занимают 7 бит), соответствуют кодировке ASCII. Коды этих символов одинаковы практически во всех однобайтовых кодировках. В состав кодировки ASCII входят цифры, буквы латинского алфавита, знаки препинания и некоторые служебные символы (например, перенос строки, табуляция и т. д.). Коды основных символов в кодировке ASCII в десятичном (dec), восьмеричном (oct) и шестнадцатеричном (hex) виде приведены в табл. 6.1.

Листинги на странице http://unicross.narod.ru/cpp/
Символы ASCII

<table>
<thead>
<tr>
<th>dec</th>
<th>oct</th>
<th>hex</th>
<th>символ</th>
</tr>
</thead>
</table>
| 0   | 0   | 0   | \
| 7   | 7   | 7   | \a   |
| 8   | 10  | 8   | \b   |
| 9   | 11  | 9   | \t   |
| 10  | 12  | a   | \n   |
| 11  | 13  | b   | \v   |
| 12  | 14  | c   | \f   |
| 13  | 15  | d   | \r   |
| 32  | 40  | 20  | пробел |
| 33  | 41  | 21  | !    |
| 34  | 42  | 22  | "    |
| 35  | 43  | 23  | #    |
| 36  | 44  | 24  | $    |
| 37  | 45  | 25  | %    |
| 38  | 46  | 26  | &    |
| 39  | 47  | 27  | '    |
| 40  | 50  | 28  | (    |
| 41  | 51  | 29  | )    |
| 42  | 52  | 2a  | *    |
| 43  | 53  | 2b  | +    |
| 44  | 54  | 2c  | ,    |
| 45  | 55  | 2d  | -    |
| 46  | 56  | 2e  | .    |

<table>
<thead>
<tr>
<th>dec</th>
<th>oct</th>
<th>hex</th>
<th>символ</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>114</td>
<td>4c</td>
<td>L</td>
</tr>
<tr>
<td>77</td>
<td>115</td>
<td>4d</td>
<td>M</td>
</tr>
<tr>
<td>78</td>
<td>116</td>
<td>4e</td>
<td>N</td>
</tr>
<tr>
<td>79</td>
<td>117</td>
<td>4f</td>
<td>O</td>
</tr>
<tr>
<td>80</td>
<td>120</td>
<td>50</td>
<td>P</td>
</tr>
<tr>
<td>81</td>
<td>121</td>
<td>51</td>
<td>Q</td>
</tr>
<tr>
<td>82</td>
<td>122</td>
<td>52</td>
<td>R</td>
</tr>
<tr>
<td>83</td>
<td>123</td>
<td>53</td>
<td>S</td>
</tr>
<tr>
<td>84</td>
<td>124</td>
<td>54</td>
<td>T</td>
</tr>
<tr>
<td>85</td>
<td>125</td>
<td>55</td>
<td>U</td>
</tr>
<tr>
<td>86</td>
<td>126</td>
<td>56</td>
<td>V</td>
</tr>
<tr>
<td>87</td>
<td>127</td>
<td>57</td>
<td>W</td>
</tr>
<tr>
<td>88</td>
<td>130</td>
<td>58</td>
<td>X</td>
</tr>
<tr>
<td>89</td>
<td>131</td>
<td>59</td>
<td>Y</td>
</tr>
<tr>
<td>90</td>
<td>132</td>
<td>5a</td>
<td>Z</td>
</tr>
<tr>
<td>91</td>
<td>133</td>
<td>5b</td>
<td>[</td>
</tr>
<tr>
<td>92</td>
<td>134</td>
<td>5c</td>
<td>\</td>
</tr>
<tr>
<td>93</td>
<td>135</td>
<td>5d</td>
<td>]</td>
</tr>
<tr>
<td>94</td>
<td>136</td>
<td>5e</td>
<td>^</td>
</tr>
<tr>
<td>95</td>
<td>137</td>
<td>5f</td>
<td>_</td>
</tr>
<tr>
<td>96</td>
<td>140</td>
<td>60</td>
<td>\</td>
</tr>
<tr>
<td>97</td>
<td>141</td>
<td>61</td>
<td>a</td>
</tr>
<tr>
<td>98</td>
<td>142</td>
<td>62</td>
<td>b</td>
</tr>
</tbody>
</table>

Листинги на странице http://unicross.narod.ru/cpp/
<table>
<thead>
<tr>
<th>47</th>
<th>57</th>
<th>2f</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>60</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>49</td>
<td>61</td>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>62</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>51</td>
<td>63</td>
<td>33</td>
<td>3</td>
</tr>
<tr>
<td>52</td>
<td>64</td>
<td>34</td>
<td>4</td>
</tr>
<tr>
<td>53</td>
<td>65</td>
<td>35</td>
<td>5</td>
</tr>
<tr>
<td>54</td>
<td>66</td>
<td>36</td>
<td>6</td>
</tr>
<tr>
<td>55</td>
<td>67</td>
<td>37</td>
<td>7</td>
</tr>
<tr>
<td>56</td>
<td>70</td>
<td>38</td>
<td>8</td>
</tr>
<tr>
<td>57</td>
<td>71</td>
<td>39</td>
<td>9</td>
</tr>
<tr>
<td>58</td>
<td>72</td>
<td>3a</td>
<td>:</td>
</tr>
<tr>
<td>59</td>
<td>73</td>
<td>3b</td>
<td>;</td>
</tr>
<tr>
<td>60</td>
<td>74</td>
<td>3c</td>
<td>&lt;</td>
</tr>
<tr>
<td>61</td>
<td>75</td>
<td>3d</td>
<td>=</td>
</tr>
<tr>
<td>62</td>
<td>76</td>
<td>3e</td>
<td>&gt;</td>
</tr>
<tr>
<td>63</td>
<td>77</td>
<td>3f</td>
<td>?</td>
</tr>
<tr>
<td>64</td>
<td>100</td>
<td>40</td>
<td>@</td>
</tr>
<tr>
<td>65</td>
<td>101</td>
<td>41</td>
<td>A</td>
</tr>
<tr>
<td>66</td>
<td>102</td>
<td>42</td>
<td>B</td>
</tr>
<tr>
<td>67</td>
<td>103</td>
<td>43</td>
<td>C</td>
</tr>
<tr>
<td>68</td>
<td>104</td>
<td>44</td>
<td>D</td>
</tr>
<tr>
<td>69</td>
<td>105</td>
<td>45</td>
<td>E</td>
</tr>
<tr>
<td>70</td>
<td>106</td>
<td>46</td>
<td>F</td>
</tr>
<tr>
<td>71</td>
<td>107</td>
<td>47</td>
<td>G</td>
</tr>
<tr>
<td>99</td>
<td>143</td>
<td>63</td>
<td>c</td>
</tr>
<tr>
<td>100</td>
<td>144</td>
<td>64</td>
<td>d</td>
</tr>
<tr>
<td>101</td>
<td>145</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>102</td>
<td>146</td>
<td>66</td>
<td>f</td>
</tr>
<tr>
<td>103</td>
<td>147</td>
<td>67</td>
<td>g</td>
</tr>
<tr>
<td>104</td>
<td>150</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>105</td>
<td>151</td>
<td>69</td>
<td>i</td>
</tr>
<tr>
<td>106</td>
<td>152</td>
<td>6a</td>
<td>j</td>
</tr>
<tr>
<td>107</td>
<td>153</td>
<td>6b</td>
<td>k</td>
</tr>
<tr>
<td>108</td>
<td>154</td>
<td>6c</td>
<td>l</td>
</tr>
<tr>
<td>109</td>
<td>155</td>
<td>6d</td>
<td>m</td>
</tr>
<tr>
<td>110</td>
<td>156</td>
<td>6e</td>
<td>n</td>
</tr>
<tr>
<td>111</td>
<td>157</td>
<td>6f</td>
<td>o</td>
</tr>
<tr>
<td>112</td>
<td>160</td>
<td>70</td>
<td>p</td>
</tr>
<tr>
<td>113</td>
<td>161</td>
<td>71</td>
<td>q</td>
</tr>
<tr>
<td>114</td>
<td>162</td>
<td>72</td>
<td>r</td>
</tr>
<tr>
<td>115</td>
<td>163</td>
<td>73</td>
<td>s</td>
</tr>
<tr>
<td>116</td>
<td>164</td>
<td>74</td>
<td>t</td>
</tr>
<tr>
<td>117</td>
<td>165</td>
<td>75</td>
<td>u</td>
</tr>
<tr>
<td>118</td>
<td>166</td>
<td>76</td>
<td>v</td>
</tr>
<tr>
<td>119</td>
<td>167</td>
<td>77</td>
<td>w</td>
</tr>
<tr>
<td>120</td>
<td>170</td>
<td>78</td>
<td>x</td>
</tr>
<tr>
<td>121</td>
<td>171</td>
<td>79</td>
<td>y</td>
</tr>
<tr>
<td>122</td>
<td>172</td>
<td>7a</td>
<td>z</td>
</tr>
<tr>
<td>123</td>
<td>173</td>
<td>7b</td>
<td>{</td>
</tr>
</tbody>
</table>

Листинги на странице http://unicross.narod.ru/cpp/
Восьмой бит у типа unsigned char предназначен для кодирования символов национальных алфавитов. У типа char эти символы имеют отрицательные значения (старший бит содержит признак знака). Таким образом тип char позволяет закодировать всего 256 символов.

Для кодирования букв русского языка предназначено пять кодировок — Windows-1251 (cp1251), cp866, iso8859-5, koi8-r и mac-cyrillic. Проблема заключается в том, что код одной и той же русской буквы в этих кодировках может быть разным. Из-за этого возникает множество проблем. Например, при выводе русских букв в консоли может отображаться нечитаемый текст. Причина искажения русских букв заключается в том, что по умолчанию в окне консоли используется кодировка cp866, а в программе мы выводим текст в кодировке cp1251. Коды русских букв и некоторых символов в кодировках cp866 и cp1251 в десятичном (unsigned и signed), восьмеричном (oct) и шестнадцатеричном (hex) виде приведены в табл. 6.2. Обратите внимание на то, что коды букв "ё" и "Ё" выпадают из последовательности кодов, поэтому при использовании, например, регулярных выражений буква "ё" не попадает в диапазон [а-я], ее необходимо указывать дополнительно [а-яё].

### Таблица 6.2. Коды русских букв и некоторых символов в кодировках cp866 и cp1251

<table>
<thead>
<tr>
<th>Символ</th>
<th>cp866</th>
<th></th>
<th></th>
<th></th>
<th>cp1251</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>unsigned</td>
<td>signed</td>
<td>oct</td>
<td>hex</td>
<td>unsigned</td>
<td>signed</td>
<td>oct</td>
<td>hex</td>
</tr>
<tr>
<td>А</td>
<td>128</td>
<td>-128</td>
<td>200</td>
<td>80</td>
<td>192</td>
<td>-64</td>
<td>300</td>
<td>c0</td>
</tr>
<tr>
<td>Б</td>
<td>129</td>
<td>-127</td>
<td>201</td>
<td>81</td>
<td>193</td>
<td>-63</td>
<td>301</td>
<td>c1</td>
</tr>
<tr>
<td>В</td>
<td>130</td>
<td>-126</td>
<td>202</td>
<td>82</td>
<td>194</td>
<td>-62</td>
<td>302</td>
<td>c2</td>
</tr>
<tr>
<td>Г</td>
<td>131</td>
<td>-125</td>
<td>203</td>
<td>83</td>
<td>195</td>
<td>-61</td>
<td>303</td>
<td>c3</td>
</tr>
<tr>
<td>Д</td>
<td>132</td>
<td>-124</td>
<td>204</td>
<td>84</td>
<td>196</td>
<td>-60</td>
<td>304</td>
<td>c4</td>
</tr>
<tr>
<td>Е</td>
<td>133</td>
<td>-123</td>
<td>205</td>
<td>85</td>
<td>197</td>
<td>-59</td>
<td>305</td>
<td>c5</td>
</tr>
</tbody>
</table>

Листинги на странице http://unicross.narod.ru/cpp/
<table>
<thead>
<tr>
<th>Б</th>
<th>240</th>
<th>-16</th>
<th>360</th>
<th>60</th>
<th>168</th>
<th>-88</th>
<th>250</th>
<th>a8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ж</td>
<td>134</td>
<td>-122</td>
<td>206</td>
<td>86</td>
<td>198</td>
<td>-58</td>
<td>306</td>
<td>c6</td>
</tr>
<tr>
<td>З</td>
<td>135</td>
<td>-121</td>
<td>207</td>
<td>87</td>
<td>199</td>
<td>-57</td>
<td>307</td>
<td>c7</td>
</tr>
<tr>
<td>И</td>
<td>136</td>
<td>-120</td>
<td>210</td>
<td>88</td>
<td>200</td>
<td>-56</td>
<td>310</td>
<td>c8</td>
</tr>
<tr>
<td>Й</td>
<td>137</td>
<td>-119</td>
<td>211</td>
<td>89</td>
<td>201</td>
<td>-55</td>
<td>311</td>
<td>c9</td>
</tr>
<tr>
<td>К</td>
<td>138</td>
<td>-118</td>
<td>212</td>
<td>8a</td>
<td>202</td>
<td>-54</td>
<td>312</td>
<td>cα</td>
</tr>
<tr>
<td>Л</td>
<td>139</td>
<td>-117</td>
<td>213</td>
<td>8b</td>
<td>203</td>
<td>-53</td>
<td>313</td>
<td>cβ</td>
</tr>
<tr>
<td>М</td>
<td>140</td>
<td>-116</td>
<td>214</td>
<td>8c</td>
<td>204</td>
<td>-52</td>
<td>314</td>
<td>cс</td>
</tr>
<tr>
<td>Н</td>
<td>141</td>
<td>-115</td>
<td>215</td>
<td>8d</td>
<td>205</td>
<td>-51</td>
<td>315</td>
<td>cд</td>
</tr>
<tr>
<td>О</td>
<td>142</td>
<td>-114</td>
<td>216</td>
<td>8e</td>
<td>206</td>
<td>-50</td>
<td>316</td>
<td>cе</td>
</tr>
<tr>
<td>П</td>
<td>143</td>
<td>-113</td>
<td>217</td>
<td>8f</td>
<td>207</td>
<td>-49</td>
<td>317</td>
<td>cф</td>
</tr>
<tr>
<td>Р</td>
<td>144</td>
<td>-112</td>
<td>220</td>
<td>90</td>
<td>208</td>
<td>-48</td>
<td>320</td>
<td>d0</td>
</tr>
<tr>
<td>С</td>
<td>145</td>
<td>-111</td>
<td>221</td>
<td>91</td>
<td>209</td>
<td>-47</td>
<td>321</td>
<td>d1</td>
</tr>
<tr>
<td>Т</td>
<td>146</td>
<td>-110</td>
<td>222</td>
<td>92</td>
<td>210</td>
<td>-46</td>
<td>322</td>
<td>d2</td>
</tr>
<tr>
<td>У</td>
<td>147</td>
<td>-109</td>
<td>223</td>
<td>93</td>
<td>211</td>
<td>-45</td>
<td>323</td>
<td>d3</td>
</tr>
<tr>
<td>Ф</td>
<td>148</td>
<td>-108</td>
<td>224</td>
<td>94</td>
<td>212</td>
<td>-44</td>
<td>324</td>
<td>d4</td>
</tr>
<tr>
<td>Х</td>
<td>149</td>
<td>-107</td>
<td>225</td>
<td>95</td>
<td>213</td>
<td>-43</td>
<td>325</td>
<td>d5</td>
</tr>
<tr>
<td>Ц</td>
<td>150</td>
<td>-106</td>
<td>226</td>
<td>96</td>
<td>214</td>
<td>-42</td>
<td>326</td>
<td>d6</td>
</tr>
<tr>
<td>Ч</td>
<td>151</td>
<td>-105</td>
<td>227</td>
<td>97</td>
<td>215</td>
<td>-41</td>
<td>327</td>
<td>d7</td>
</tr>
<tr>
<td>Щ</td>
<td>152</td>
<td>-104</td>
<td>230</td>
<td>98</td>
<td>216</td>
<td>-40</td>
<td>330</td>
<td>d8</td>
</tr>
<tr>
<td>Ъ</td>
<td>153</td>
<td>-103</td>
<td>231</td>
<td>99</td>
<td>217</td>
<td>-39</td>
<td>331</td>
<td>d9</td>
</tr>
<tr>
<td>Ь</td>
<td>154</td>
<td>-102</td>
<td>232</td>
<td>9a</td>
<td>218</td>
<td>-38</td>
<td>332</td>
<td>da</td>
</tr>
<tr>
<td>Ы</td>
<td>155</td>
<td>-101</td>
<td>233</td>
<td>9b</td>
<td>219</td>
<td>-37</td>
<td>333</td>
<td>db</td>
</tr>
<tr>
<td>Ь</td>
<td>156</td>
<td>-100</td>
<td>234</td>
<td>9c</td>
<td>220</td>
<td>-36</td>
<td>334</td>
<td>dc</td>
</tr>
<tr>
<td>Э</td>
<td>157</td>
<td>-99</td>
<td>235</td>
<td>9d</td>
<td>221</td>
<td>-35</td>
<td>335</td>
<td>dd</td>
</tr>
</tbody>
</table>

Листинги на странице http://unicross.narod.ru/cpp/
<table>
<thead>
<tr>
<th>Ю</th>
<th>158</th>
<th>-98</th>
<th>236</th>
<th>9e</th>
<th>222</th>
<th>-34</th>
<th>336</th>
<th>de</th>
</tr>
</thead>
<tbody>
<tr>
<td>Я</td>
<td>159</td>
<td>-97</td>
<td>237</td>
<td>9f</td>
<td>223</td>
<td>-33</td>
<td>337</td>
<td>df</td>
</tr>
<tr>
<td>А</td>
<td>160</td>
<td>-96</td>
<td>240</td>
<td>a0</td>
<td>224</td>
<td>-32</td>
<td>340</td>
<td>e0</td>
</tr>
<tr>
<td>Б</td>
<td>161</td>
<td>-95</td>
<td>241</td>
<td>a1</td>
<td>225</td>
<td>-31</td>
<td>341</td>
<td>e1</td>
</tr>
<tr>
<td>В</td>
<td>162</td>
<td>-94</td>
<td>242</td>
<td>a2</td>
<td>226</td>
<td>-30</td>
<td>342</td>
<td>e2</td>
</tr>
<tr>
<td>Г</td>
<td>163</td>
<td>-93</td>
<td>243</td>
<td>a3</td>
<td>227</td>
<td>-29</td>
<td>343</td>
<td>e3</td>
</tr>
<tr>
<td>Д</td>
<td>164</td>
<td>-92</td>
<td>244</td>
<td>a4</td>
<td>228</td>
<td>-28</td>
<td>344</td>
<td>e4</td>
</tr>
<tr>
<td>Е</td>
<td>165</td>
<td>-91</td>
<td>245</td>
<td>a5</td>
<td>229</td>
<td>-27</td>
<td>345</td>
<td>e5</td>
</tr>
<tr>
<td>Ё</td>
<td>241</td>
<td>-15</td>
<td>361</td>
<td>f1</td>
<td>184</td>
<td>-72</td>
<td>270</td>
<td>b8</td>
</tr>
<tr>
<td>Ж</td>
<td>166</td>
<td>-90</td>
<td>246</td>
<td>a6</td>
<td>230</td>
<td>-26</td>
<td>346</td>
<td>e6</td>
</tr>
<tr>
<td>З</td>
<td>167</td>
<td>-89</td>
<td>247</td>
<td>a7</td>
<td>231</td>
<td>-25</td>
<td>347</td>
<td>e7</td>
</tr>
<tr>
<td>И</td>
<td>168</td>
<td>-88</td>
<td>250</td>
<td>a8</td>
<td>232</td>
<td>-24</td>
<td>350</td>
<td>e8</td>
</tr>
<tr>
<td>Й</td>
<td>169</td>
<td>-87</td>
<td>251</td>
<td>a9</td>
<td>233</td>
<td>-23</td>
<td>351</td>
<td>e9</td>
</tr>
<tr>
<td>К</td>
<td>170</td>
<td>-86</td>
<td>252</td>
<td>aа</td>
<td>234</td>
<td>-22</td>
<td>352</td>
<td>eа</td>
</tr>
<tr>
<td>Л</td>
<td>171</td>
<td>-85</td>
<td>253</td>
<td>aб</td>
<td>235</td>
<td>-21</td>
<td>353</td>
<td>eб</td>
</tr>
<tr>
<td>М</td>
<td>172</td>
<td>-84</td>
<td>254</td>
<td>ас</td>
<td>236</td>
<td>-20</td>
<td>354</td>
<td>ес</td>
</tr>
<tr>
<td>Н</td>
<td>173</td>
<td>-83</td>
<td>255</td>
<td>ад</td>
<td>237</td>
<td>-19</td>
<td>355</td>
<td>ed</td>
</tr>
<tr>
<td>О</td>
<td>174</td>
<td>-82</td>
<td>256</td>
<td>ae</td>
<td>238</td>
<td>-18</td>
<td>356</td>
<td>ее</td>
</tr>
<tr>
<td>П</td>
<td>175</td>
<td>-81</td>
<td>257</td>
<td>af</td>
<td>239</td>
<td>-17</td>
<td>357</td>
<td>ef</td>
</tr>
<tr>
<td>Р</td>
<td>224</td>
<td>-32</td>
<td>340</td>
<td>e0</td>
<td>240</td>
<td>-16</td>
<td>360</td>
<td>f0</td>
</tr>
<tr>
<td>С</td>
<td>225</td>
<td>-31</td>
<td>341</td>
<td>e1</td>
<td>241</td>
<td>-15</td>
<td>361</td>
<td>f1</td>
</tr>
<tr>
<td>Т</td>
<td>226</td>
<td>-30</td>
<td>342</td>
<td>e2</td>
<td>242</td>
<td>-14</td>
<td>362</td>
<td>f2</td>
</tr>
<tr>
<td>У</td>
<td>227</td>
<td>-29</td>
<td>343</td>
<td>e3</td>
<td>243</td>
<td>-13</td>
<td>363</td>
<td>f3</td>
</tr>
<tr>
<td>Ф</td>
<td>228</td>
<td>-28</td>
<td>344</td>
<td>e4</td>
<td>244</td>
<td>-12</td>
<td>364</td>
<td>f4</td>
</tr>
<tr>
<td>Х</td>
<td>229</td>
<td>-27</td>
<td>345</td>
<td>e5</td>
<td>245</td>
<td>-11</td>
<td>365</td>
<td>f5</td>
</tr>
</tbody>
</table>

Листинги на странице http://unicross.narod.ru/cpp/
<table>
<thead>
<tr>
<th>Ц</th>
<th>230</th>
<th>-26</th>
<th>346</th>
<th>eb</th>
<th>246</th>
<th>-10</th>
<th>366</th>
<th>f6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ч</td>
<td>231</td>
<td>-25</td>
<td>347</td>
<td>e7</td>
<td>247</td>
<td>-9</td>
<td>367</td>
<td>f7</td>
</tr>
<tr>
<td>Ш</td>
<td>232</td>
<td>-24</td>
<td>350</td>
<td>e8</td>
<td>248</td>
<td>-8</td>
<td>370</td>
<td>f8</td>
</tr>
<tr>
<td>Щ</td>
<td>233</td>
<td>-23</td>
<td>351</td>
<td>e9</td>
<td>249</td>
<td>-7</td>
<td>371</td>
<td>f9</td>
</tr>
<tr>
<td>Ъ</td>
<td>234</td>
<td>-22</td>
<td>352</td>
<td>ea</td>
<td>250</td>
<td>-6</td>
<td>372</td>
<td>fa</td>
</tr>
<tr>
<td>Ы</td>
<td>235</td>
<td>-21</td>
<td>353</td>
<td>eb</td>
<td>251</td>
<td>-5</td>
<td>373</td>
<td>fb</td>
</tr>
<tr>
<td>Ь</td>
<td>236</td>
<td>-20</td>
<td>354</td>
<td>ec</td>
<td>252</td>
<td>-4</td>
<td>374</td>
<td>fc</td>
</tr>
<tr>
<td>Э</td>
<td>237</td>
<td>-19</td>
<td>355</td>
<td>ed</td>
<td>253</td>
<td>-3</td>
<td>375</td>
<td>fd</td>
</tr>
<tr>
<td>Ю</td>
<td>238</td>
<td>-18</td>
<td>356</td>
<td>ee</td>
<td>254</td>
<td>-2</td>
<td>376</td>
<td>fe</td>
</tr>
<tr>
<td>Я</td>
<td>239</td>
<td>-17</td>
<td>357</td>
<td>ef</td>
<td>255</td>
<td>-1</td>
<td>377</td>
<td>ff</td>
</tr>
<tr>
<td>Є</td>
<td>242</td>
<td>-14</td>
<td>362</td>
<td>f2</td>
<td>170</td>
<td>-86</td>
<td>252</td>
<td>aa</td>
</tr>
<tr>
<td>Є</td>
<td>243</td>
<td>-13</td>
<td>363</td>
<td>f3</td>
<td>186</td>
<td>-70</td>
<td>272</td>
<td>ba</td>
</tr>
<tr>
<td>°</td>
<td>248</td>
<td>-8</td>
<td>370</td>
<td>f8</td>
<td>176</td>
<td>-80</td>
<td>260</td>
<td>b0</td>
</tr>
<tr>
<td>Ъ</td>
<td>252</td>
<td>-4</td>
<td>374</td>
<td>fc</td>
<td>185</td>
<td>-71</td>
<td>271</td>
<td>b9</td>
</tr>
<tr>
<td>&quot;</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>132</td>
<td>−124</td>
<td>204</td>
<td>84</td>
</tr>
<tr>
<td>...</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>133</td>
<td>−123</td>
<td>205</td>
<td>85</td>
</tr>
<tr>
<td>Ћ</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>136</td>
<td>−120</td>
<td>210</td>
<td>88</td>
</tr>
<tr>
<td>Ћ</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>137</td>
<td>−119</td>
<td>211</td>
<td>89</td>
</tr>
<tr>
<td>'</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>145</td>
<td>−111</td>
<td>221</td>
<td>91</td>
</tr>
<tr>
<td>&quot;</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>146</td>
<td>−110</td>
<td>222</td>
<td>92</td>
</tr>
<tr>
<td>&quot;</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>147</td>
<td>−109</td>
<td>223</td>
<td>93</td>
</tr>
<tr>
<td>&quot;</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>148</td>
<td>−108</td>
<td>224</td>
<td>94</td>
</tr>
<tr>
<td>-</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>150</td>
<td>−106</td>
<td>226</td>
<td>96</td>
</tr>
<tr>
<td>-</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>151</td>
<td>−105</td>
<td>227</td>
<td>97</td>
</tr>
<tr>
<td>&quot;</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>153</td>
<td>−103</td>
<td>231</td>
<td>99</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>166</th>
<th>-90</th>
<th>246</th>
<th>\texttt{ab}</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>167</td>
<td>-89</td>
<td>247</td>
<td>\texttt{a7}</td>
</tr>
<tr>
<td>$\odot$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>169</td>
<td>-87</td>
<td>251</td>
<td>\texttt{a9}</td>
</tr>
<tr>
<td>$\ll$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>171</td>
<td>-85</td>
<td>253</td>
<td>\texttt{ab}</td>
</tr>
<tr>
<td>$\neg$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>172</td>
<td>-84</td>
<td>254</td>
<td>\texttt{ac}</td>
</tr>
<tr>
<td>$\ominus$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>173</td>
<td>-83</td>
<td>255</td>
<td>\texttt{ad}</td>
</tr>
<tr>
<td>$\pm$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>174</td>
<td>-82</td>
<td>256</td>
<td>\texttt{ae}</td>
</tr>
<tr>
<td>$\gg$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>187</td>
<td>-69</td>
<td>273</td>
<td>\texttt{bb}</td>
</tr>
</tbody>
</table>

**Настройка локали в языке C**

При изменении регистра русских букв может возникнуть проблема. Чтобы ее избежать, необходимо правильно настроить локаль (совокупность локальных настроек системы). Для настройки локали используется функция `setlocale()`. Прототип функции:

```c
#include <locale>
char *setlocale(int Category, const char *Locale);
```

В первом параметре указывается категория в виде числа от 0 до 5 или соответствующий числу макрос:

- 0 — `LC_ALL` — устанавливает локаль для всех категорий;
- 1 — `LC_COLLATE` — для сравнения строк;
- 2 — `LC_CTYPE` — для перевода символов в нижний или верхний регистр;
- 3 — `LC_MONETARY` — для отображения денежных единиц;
- 4 — `LC_NUMERIC` — для форматирования дробных чисел;
- 5 — `LC_TIME` — для форматирования вывода даты и времени.

Во втором параметре задается название локали в виде строки (например, "$\text{rus}$", "$\text{Russian_Russia}$", "$\text{Russian_Russia.1251}$" или "$\text{1251}$"). Вместо названия можно указать пустую строку. В этом случае будет использоваться локаль, настроенная в системе. В VC++ после настройки локали автоматически производится преобразование кодировок при выводе в окно консоли. Пример настройки локали для всех категорий:

std::setlocale(LC_ALL, "Russian_Russia.1251");

В качестве значения функция возвращает указатель на строку с названием локали, соответствующей заданной категории, или нулевой указатель в случае ошибки. Если во втором параметре передан нулевой указатель, то функция возвращает указатель на строку с названием текущей локали. Пример настройки локали и вывода текущего названия локали приведен в листинге 6.1.

### Листинг 6.1. Настройка локали

```cpp
#include <iostream>
#include <locale>

int main() {
    char *pch = 0;
    // Настройка локали
    pch = std::setlocale(LC_ALL, "Russian_Russia.1251");
    if (pch) std::cout << pch << std::endl; // Russian_Russia.1251
    // Получение текущей локали
    pch = std::setlocale(LC_ALL, 0);
    if (pch) std::cout << pch << std::endl; // Russian_Russia.1251
    std::cout << "В VC++ этот русский текст в консоли "
              << "отображается нормально" << std::endl;
    std::cin.get();
    return 0;
}
```

Функция `localeconv()` позволяет получить информацию о способе форматирования вещественных чисел и денежных сумм для текущей локали. Прототип функции:

```cpp
#include <locale>
struct lconv *localeconv(void);
```

Функция возвращает указатель на структуру `lconv`. Пример вывода символа десятичного разделителя для локали `Russian_Russia.1251`:

```cpp
lconv *p = 0;
std::setlocale(LC_ALL, "Russian_Russia.1251");
p = std::localeconv();
if (p) {
    std::cout << p->decimal_point << std::endl;
}
```

Объявление структуры `lconv` выглядит следующим образом:

```c
struct lconv {
    char *decimal_point; // Десятичный разделитель (",")
    char *thousands_sep; // Разделитель тысяч (" ")
    char *grouping; // Способ группировки значений
    char *int_curr_symbol; // Название валюты ("RUR")
    char *currency_symbol; // Символ валюты ("р.")
    char *mon_decimal_point; // Десятичный разделитель для денежных сумм (",")
    char *mon_thousands_sep; // Разделитель тысяч для денежных сумм (" ")
    char *mon_grouping; // Способ группировки для денежных сумм
    char *positive_sign; // Положительный знак для денежных сумм
    char *negative_sign; // Отрицательный знак для денежных сумм ("-")
    char int_frac_digits; // Количество цифр в дробной части для денежных сумм в международном формате (2)
    char frac_digits; // Количество цифр в дробной части для денежных сумм в национальном формате (2)
    char p_cs_precedes; // 1 - если символ валюты перед значением
                              // 0 - если символ валюты после значения
    char p_sep_by_space; // 1 - если символ валюты отделяется пробелом
                             // 0 - в противном случае
                              // p_cs_precedes и p_sep_by_space применяются для положительных значений
    char n_cs_precedes; // 1 - если символ валюты перед значением
                          // 0 - если символ валюты после значения
    char n_sep_by_space; // 1 - если символ валюты отделяется пробелом
                          // 0 - в противном случае
                          // n_cs_precedes и n_sep_by_space применяются для отрицательных значений
    char p_sign_posn; // Позиция символа положительного значения
    char n_sign_posn; // Позиция символа отрицательного значения
    wchar_t *W_decimal_point; // Аналогичные значения для расширенных символов. Описание см. выше
    wchar_t *W_thousands_sep; // символов.
    wchar_t *W_int_curr_symbol;
```

Листинги на странице http://unicross.narod.ru/cpp/
```c
wchar_t *_W_currency_symbol;
wchar_t *_W_mon_decimal_point;
wchar_t *_W_mon_thousands_sep;
wchar_t *_W_positive_sign;
wchar_t *_W_negative_sign;
```

**Настройка локали в языке C++**

В языке C++ настройка локали производится с помощью класса `locale`. Прежде чем использовать класс необходимо подключить заголовочный файл `locale`. Наиболее часто используются следующие конструкторы класса:

```c
#include <locale>
locale();
locale(const char *Locname, category Cat = all);
locale(const string &Str, category Cat = all);
locale(const locale &Right);
locale(const locale &Loc, const char *Locname, category Cat);
locale(const locale &Loc, const string &Str, category Cat);
locale(const locale &Loc, const locale &Other, category Cat);
```

В параметрах `Locname` и `Str` задается название локали в виде C-строки или объекта класса `string` соответственно (например, "rus", "Russian_Russia", "Russian_Russia.1251" или "1251"). Вместо названия можно указать пустую строку. В этом случае будет использоваться локаль, настроенная в системе. В параметре `Cat` можно указать следующие флаги (или их комбинацию через побитовый оператор `|`):

- `locale::all` — устанавливает локаль для всех категорий;
- `locale::collate` — для сравнения строк;
- `locale::ctype` — для перевода символов в нижний или верхний регистр;
- `locale::monetary` — для отображения денежных единиц;
- `locale::numeric` — для форматирования дробных чисел;
- `locale::time` — для форматирования вывода даты и времени;
- `locale::messages`;
- `locale::none` — флаги не установлены.

Пример создания объекта класса locale для всех категорий в локали Russian_Russia.1251:

```cpp
std::locale loc("Russian_Russia.1251", std::locale::all);
```

Чтобы установить локаль на глобальном уровне следует передать созданный объект в статический метод `global()`. Прототип метода:

```cpp
static locale global(const locale &Loc);
```

Метод принимает объект класса `locale` и возвращает прежнюю настройку локали. Получить название локали позволяют методы `c_str()` и `name()`. Метод `c_str()` возвращает значение в виде C-строки, а метод `name()` — в виде объекта класса `string`. Прототипы методов:

```cpp
const char *c_str() const;
string name() const;
```

Пример настройки локали и вывода названия текущей локали приведен в листинге 6.2.

**Листинг 6.2. Настройка локали**

```cpp
#include <iostream>
#include <locale>

int main() {
    std::locale loc("Russian_Russia.1251", std::locale::all);
    std::locale loc2 = std::locale::global(loc);
    std::cout << loc.c_str() << std::endl; // Russian_Russia.1251
    std::cout << loc2.c_str() << std::endl; // C
    std::cout << "Этот текст корректно отображается в окне консоли"
             << std::endl;
    std::cin.get();
    return 0;
}
```

В параметре метода `global()` можно указать объект еще одним способом:

```cpp
std::locale::global(std::locale("Russian_Russia.1251"));
```

Как видно из примера, объект сразу передается в метод `global()` и нигде не сохраняется. Чтобы получить название настроенной локали следует использовать конструктор по умолчанию. Пример получения названия:

```cpp
std::cout << std::locale().c_str() << std::endl; // C
```

std::locale::global(std::locale("Russian_Russia.1251"));
std::cout << std::locale().c_str() << std::endl; // Russian_Russia.1251

Создать объект класса locale для локали с позволяет статический метод classic(). С помощью этого метода обычно производят сравнение локалей. Прототип метода:
static const locale &classic();

Пример:
if (std::locale() == std::locale::classic()) {
    std::cout << "C" << std::endl;
} else {
    std::cout << std::locale().c_str() << std::endl;
}

Создать пустой объект класса locale позволяет метод empty(). Прототип метода:
static locale empty();

Пример:
std::locale loc = std::locale::empty();
std::cout << loc.c_str() << std::endl; // *

### Изменение регистра символа

Для изменения регистра символа предназначены следующие функции:

- **toupper**() — возвращает код символа в верхнем регистре. Если преобразования регистра не было, то код символа возвращается без изменений. Прототип функции:

```c++
#include <cctype>
int toupper(int C);
```

Пример:

```c++
std::setlocale(LC_ALL, "Russian_Russia.1251");
std::cout << (char)std::toupper('в') << std::endl; // В
std::cout << (char)std::toupper('В') << std::endl; // В
std::cout << (char)std::toupper((unsigned char)'ё')
         << std::endl; // ё
```

- **tolower**() — возвращает код символа в нижнем регистре. Если преобразования регистра не было, то код символа возвращается без изменений. Прототип функции:

Листинги на странице http://unicross.narod.ru/cpp/

#include <ctype>
int tolower(int C);

Пример:
std::setlocale(LC_ALL, "Russian_Russia.1251");
std::cout << (char)std::tolower('w') << std::endl; // w
std::cout << (char)std::tolower('W') << std::endl; // W
std::cout << (char)std::tolower((unsigned char)'E')
         << std::endl; // Е

Проверка типа содержимого символа

Для проверки типа содержимого символа предназначены следующие функции:

→ isdigit() — возвращает ненулевое значение, если символ является десятичной цифрой, в противном случае — 0. Прототип функции:
#include <cctype>
int isdigit(int C);

Пример:
std::cout << std::isdigit('w') << std::endl; // 0
std::cout << std::isdigit('2') << std::endl; // 4
std::cout << std::isdigit((unsigned char)'6')
         << std::endl; // 0

→ isxdigit() — возвращает ненулевое значение, если символ является шестнадцатеричной цифрой (число от 0 до 9 или буква от A до F (регистр не имеет значения)), в противном случае — 0. Прототип функции:
#include <cctype>
int isxdigit(int C);

Пример:
std::cout << std::isxdigit('8') << std::endl; // 128
std::cout << std::isxdigit('a') << std::endl; // 128
std::cout << std::isxdigit('F') << std::endl; // 128
std::cout << std::isxdigit('g') << std::endl; // 0

Листинги на странице http://unicross.narod.ru/cpp/
isalpha() — возвращает ненулевое значение, если символ является буквой, в противном случае — 0. Прототип функции:

```c++
#include <ctype>

int isalpha(int C);
```

Для русских букв необходимо настроить локаль. Пример:

```c++
std::setlocale(LC_ALL, "Russian_Russia.1251");
std::cout << std::isalpha('ω') << std::endl; // 258
std::cout << std::isalpha('2') << std::endl; // 0
std::cout << std::isalpha((unsigned char)'ё')
    << std::endl; // 258
std::cout << std::isalpha((unsigned char)'Б')
    << std::endl; // 257
```

isspace() — возвращает ненулевое значение, если символ является пробельным символом (пробелом, табуляцией, переводом строки или возвратом каретки), в противном случае — 0. Прототип функции:

```c++
#include <ctype>

int isspace(int C);
```

Пример:

```c++
std::cout << std::isspace('ω') << std::endl; // 0
std::cout << std::isspace('\n') << std::endl; // 8
std::cout << std::isspace('\t') << std::endl; // 8
std::cout << std::isspace((unsigned char)'ё')
    << std::endl; // 0
```

isalnum() — возвращает ненулевое значение, если символ является буквой или цифрой, в противном случае — 0. Прототип функции:

```c++
#include <ctype>

int isalnum(int C);
```

Для русских букв необходимо настроить локаль. Пример:

```c++
std::setlocale(LC_ALL, "Russian_Russia.1251");
std::cout << std::isalnum('ω') << std::endl; // 258
std::cout << std::isalnum('8') << std::endl; // 4
```

Листинги на странице http://unicross.narod.ru/cpp/
std::cout << std::isalnum('	') << std::endl; // 0
std::cout << std::isalnum((unsigned char)'б')
       << std::endl; // 258

islower() — возвращает ненулевое значение, если символ является буквой в нижнем регистре, в противном случае — 0. Прототип функции:
#include <cctype>
int islower(int C);

Для русских букв необходимо настроить локаль. Пример:
std::setlocale(LC_ALL, "Russian_Russia.1251");
std::cout << std::islower('в') << std::endl; // 2
std::cout << std::islower('8') << std::endl; // 0
std::cout << std::islower('	') << std::endl; // 0
std::cout << std::islower((unsigned char)'б')
       << std::endl; // 2
std::cout << std::islower((unsigned char)'Б')
       << std::endl; // 0

isupper() — возвращает ненулевое значение, если символ является буквой в верхнем регистре, в противном случае — 0. Прототип функции:
#include <cctype>
int isupper(int C);

Для русских букв необходимо настроить локаль. Пример:
std::setlocale(LC_ALL, "Russian_Russia.1251");
std::cout << std::isupper('в') << std::endl; // 1
std::cout << std::isupper('8') << std::endl; // 0
std::cout << std::isupper('	') << std::endl; // 0
std::cout << std::isupper((unsigned char)'б')
       << std::endl; // 0
std::cout << std::isupper((unsigned char)'Б')
       << std::endl; // 1

ispunct() — возвращает ненулевое значение, если символ является символом пунктуации, в противном случае — 0. Прототип функции:

Листинги на странице http://unicross.narod.ru/cpp/

#include <cctype>

int ispunct(int C);

Пример:
std::cout << std::ispunct('8') << std::endl; // 0
std::cout << std::ispunct('f') << std::endl; // 0
std::cout << std::ispunct(',') << std::endl; // 16
std::cout << std::ispunct('.') << std::endl; // 16
std::cout << std::ispunct(' ') << std::endl; // 0

→ isprint() — возвращает ненулевое значение, если символ является печатаемым (включая пробел), в противном случае — 0. Прототип функции:
#include <cctype>

int isprint(int C);

Для русских букв необходимо настроить локаль. Пример:
std::setlocale(LC_ALL, "Russian_Russia.1251");
std::cout << std::isprint('8') << std::endl; // 4
std::cout << std::isprint('\x5') << std::endl; // 0
std::cout << std::isprint(' ') << std::endl; // 64
std::cout << std::isprint((unsigned char)'6')
       << std::endl; // 258

→ isgraph() — возвращает ненулевое значение, если символ является печатаемым (пробел печатаемым не считается), в противном случае — 0. Прототип функции:
#include <cctype>

int isgraph(int C);

Для русских букв необходимо настроить локаль. Пример:
std::setlocale(LC_ALL, "Russian_Russia.1251");
std::cout << std::isgraph('8') << std::endl; // 4
std::cout << std::isgraph('\x5') << std::endl; // 0
std::cout << std::isgraph(' ') << std::endl; // 0
std::cout << std::isgraph((unsigned char)'6')
       << std::endl; // 258

Листинги на странице http://unicross.narod.ru/cpp/
исcntrl() — возвращает ненулевое значение, если символ является
непечатаемым, в противном случае — 0. Прототип функции:
#include <cctype>
int iscntrl(int C);

Пример:
std::cout << std::iscntrl('8') << std::endl;  // 0
std::cout << std::iscntrl('\x5') << std::endl;  // 32
std::cout << std::iscntrl(' ') << std::endl;  // 0

Обратите внимание на то, что перед русскими буквами указывается операция
приведения к типу unsigned char. Если приведение типов не указать, то производится
попытка преобразования к типу unsigned int. Так как коды русских букв по
умолчанию имеют отрицательные значения, знаковый бит станет причиной большого
значения, которое выходит за рамки диапазона значений для типа char. Например, для
буквы "б" значение будет равно 4294967265. Пример:
std::cout << (unsigned int)(unsigned char)'б'
    << std::endl;  // 225
std::cout << (unsigned int)'б'
    << std::endl;  // 4294967265

C-строки

C-строка является массивом символов (тип char), последний элемент которого
содержит нулевой символ (\0). Обратите внимание на то, что нулевой символ (нулевой
байт) не имеет никакого отношения к числу 0. Коды этих символов разные.

Объявление и инициализация C-строки

Объявляется C-строка также как и массив элементов типа char:
char str[7];
При инициализации C-строки можно перечислить символы внутри фигурных скобок:
char str[7] = {'S', 't', 'r', 'i', 'n', 'g', '0'};
или указать строку внутри двойных кавычек:
char str[7] = "String";
При использовании двойных кавычек следует учитывать, что длина строки на один
символ больше, так как в конец будет автоматически вставлен нулевой символ. Если

Листинги на странице http://unicross.narod.ru/cpp/
это не предусмотреть и объявить массив из шести элементов, вместо семи, то при компиляции будет выведено сообщение об ошибке.

Если размер массива при объявлении не указать, то он будет определен автоматически в соответствии с длиной строки:
```c
char str[] = "String";
```
Обратите внимание на то, что присваивать строку в двойных кавычках можно только при инициализации. Попытка присвоить строку позже приведет к ошибке:
```c
char str[7];
str = "String"; // Ошибка!!!
```
Внутри строки в двойных кавычках можно указывать специальные символы (например, \n, \t и dr.), которые мы уже рассматривали в начале этой главы. Если внутри строки встречается кавычка, то ее необходимо экранировать с помощью обратного слеша:
```c
char str[] = "Труппа "Кино"\n;";
```
Если внутри строки встречается обратный слеш, то его необходимо экранировать. Это обязательно следует учитывать при указании пути к файлу:
```c
char str1[] = "C:\temp\new\file.txt"; // Правильно
char str2[] = "C:\temp\new\file.txt"; // Неправильно!!!
```
Обратите внимание на последнюю строку. В этом пути присутствуют сразу три специальных символов: \t, \n и \f. После преобразования специальных символов путь будет выглядеть следующим образом:
```c
C:\<Табуляция>emp<Перевод строки>ew<Перевод формата>ile.txt
```
Разместить C-столько при инициализации на нескольких строках нельзя. Переход на новую строку вызовет синтаксическую ошибку ("error C2001: newline в константе"):
```c
char str[] = "string1
string2"; // Ошибка: error C2001: newline в константе
```
Чтобы разместить C-столько на нескольких строках, следует перед символом перевода строки указать символ \
```c
char str[] = "string1 \nstring2 \nstring3"; // После символа \ не должно быть никаких символов
std::cout << str << std::endl; // string1 string2 string3
```
Кроме того, можно воспользоваться неявной конкатенацией. Если строки расположены подряд внутри одной инструкции, то они объединяются в одну большую строку. Пример:
```c
Листинги на странице http://unicross.narod.ru/cpp/
```
char str[] = "string1"
"string2" "string3";
std::cout << str << std::endl; // string1string2string3

Объявление массива строк выглядит так:
std::cout << str[0] << std::endl; // String1
std::cout << str[1] << std::endl; // String2
std::cout << str[2] << std::endl; // String3

или так:
char *str[] = {"String1", "String2", "String3"};
std::cout << str[0] << std::endl; // String1
std::cout << str[1] << std::endl; // String2
std::cout << str[2] << std::endl; // String3

Доступ к символам внутри C-строки

После определения C-строки в переменной сохраняется адрес первого символа. Иными словами, название переменной является указателем, который ссылается на первый символ строки. Поэтому доступ к элементу массива может осуществляться как по индексу (нумерация начинается с нуля), указанному внутри квадратных скобок, так и с использованием адресной арифметики. Например, следующие две инструкции вывода являются эквивалентными:
char str[] = "String";
std::cout << str[1] << std::endl; // t
std::cout << *(str + 1) << std::endl; // t

Символ можно не только получить таким образом, но и изменить:
char str[] = "String";
str[0] = 's';    // Изменение с помощью индекса
*(str + 1) = 'T'; // Изменение с помощью указателя
std::cout << str << std::endl; // sTing

Обратите внимание на то, что отдельный символ указывается внутри апострофов, а не внутри кавычек. Если указать символ внутри кавычек, то вместо одного символа будет два: собственно сам символ плюс нулевой символ.

Объявить указатель и присвоить ему адрес строки можно следующим образом:
char *p = 0;
char str[] = "String";

Листинги на странице http://unicross.narod.ru/cpp/
p = str;
*p = 's';
++p;    // Перемещаем указатель на второй символ
*p = 'T';
std::cout << str << std::endl; // String

Обратите внимание на то, что перед названием строки не указывается оператор &, так как название переменной содержит адрес первого символа. Если использовать оператор &, то необходимо дополнительно указать индекс внутри квадратных скобок:
p = &str[0]; // Эквивалентно: p = str;

При инициализации указателя ему можно присвоить строку. Такие строки не рекомендуется изменять, поэтому обычно перед типом указывают ключевое слово const. Пример:

const char *str = "String";
std::cout << str << std::endl; // String

Перебор символов C-строки

Для перебора символов удобно использовать цикл for. В первом параметре переменной-счетчику присваивается значение 0 (символы в строке нумеруются с нуля), условием продолжения является значение переменной-счетчика меньше количества символов в строке. Получить количество символов можно с помощью функции strlen(). Прототип функции:

#include <cstring>
size_t strlen(const char *Str);

Функция strlen() возвращает количество символов без учета нулевого символа. Тип size_t определен как беззнаковое целое. В третьем параметре цикла for указывается приращение на единицу на каждой итерации цикла. Выведем все символы строки в прямом и обратном порядке:

char str[] = "String";
int len = std::strlen(str);
// Выводим символы в прямом порядке
for (int i=0; i<len; ++i) {
    std::cout << str[i] << std::endl;
}
std::cout << "---------------------" << std::endl;
// Выводим символы в обратном порядке

Листинг на странице http://unicross.narod.ru/cpp/
for (int j=len-1; j>=0; --j) {
    std::cout << str[j] << std::endl;
}
В этом примере количество символов сохраняется в переменной len вне цикла. Если
функцию strlen() указать внутри условия, то вычисление количества символов будет
вычисляться на каждой итерации цикла. Поэтому количество символов лучше получить
вне цикла или присваивать значение переменной в первом параметре цикла for.
Пример:
for (int i=0, len = std::strlen(str); i<len; ++i) {
    std::cout << str[i] << std::endl;
}
Перебор символов C-строки с помощью указателя и цикла for выполняется так:
char str[] = "String";
for (char *p=str; *p; ++p) {
    std::cout << *p << std::endl;
}
В этом примере начальным значением является адрес первого символа. Условием
продолжения цикла является значение на которое ссылаются указатель. Любой символ
трактуется как true, кроме нулевого символа, который имеет код 0. Так как C-строка
завершается нулевым символом, то этот символ вернет значение false и цикл
завершится. В третьем параметре цикла for указывается приращение указателя на
единицу на каждой итерации цикла В этом случае используются правила адресной
арифметики.
Вместо цикла for всегда можно использовать цикл while:
char *p = 0;
char str[] = "String";
p = str;
while (*p) {
    std::cout << *p++ << std::endl;
}
p = str; // Восстанавливаем положение указателя
Вначале объявляются указатель и строка. Далее указателю присваивается адрес первого
символа. Цикл while выполняется до тех пор, пока значение, на которое ссылается
указатель, не равно нулевому символу. Внутри цикла while выводится символ, на
который ссылается указатель, а затем указатель перемещается на следующий символ

Листинги на странице http://unicross.narod.ru/cpp/

Основные функции для работы с C-строками

Перечислим основные функции для работы с C-строками:

→ strlen() — возвращает количество символов в C-строке без учета нулевого символа. Прототип функции:

```c
#include <cstring>
size_t strlen(const char *Str);
```

Тип size_t определен как беззнаковое целое число. Пример:

```cpp
char str[7] = "String";
std::cout << std::strlen(str) << std::endl; // 6, а не 7
```

Определить общий размер массива можно с помощью оператора sizeof. Оператор возвращает размер в байтах. Так как тип char занимает 1 байт, следовательно размер в байтах будет равен размеру массива. Пример:

```cpp
char str[20] = "String";
std::cout << sizeof str << std::endl; // 20
```

→ strcpy() — копирует символы из C-строки Source в C-строку Dest и вставляет нулевой символ. В качестве значения функция возвращает указатель на строку Dest. Если строка Source длиннее строки Dest, то произойдет переполнение буфера. Прототип функции:

```c
#include <cstring>
char *strcpy(char *Dest, const char *Source);
```

Пример использования функции:

```cpp
char str[7];
std::strcpy(str, "String");
std::cout << str << std::endl; // String
```

В VC++ при использовании функции strcpy() выводится предупреждающее сообщение ("warning C4996"), поэтому вместо функции strcpy() следует использовать функцию strcpy_s(). Прототип функции:

Листинги на странице http://unicross.narod.ru/cpp/
#include <cstring>

errno_t strcpy_s(char *Dst, rsize_t SizeInBytes,
    const char *Src);

Функция strcpy_s() копирует символы из строки Src в строку Dst и вставляет нулевой символ. В параметре SizeInBytes указывается максимальное количество элементов массива Dst. Пример:
const short SIZE = 7;
char str[SIZE];
strcpy_s(str, SIZE, "String");
std::cout << str << std::endl; // String

-> strncpy() — копирует первые Count символов из C-строки Source в C-строку Dest. В качестве значения функция возвращает указатель на строку Dest. Если строка Source длиннее строки Dest, то произойдет переполнение буфера. Прототип функции:
#include <cstring>

char *strncpy(char *Dest, const char *Source, size_t Count);

Если количество символов в строке Source меньше числа Count, то строка Dest будет дополнена нулевыми символами, а если больше, то копируются только Count символов, при этом нулевой символ автоматически не вставляется. Пример копирования шести символов:
char str[7];
std::strncpy(str, "String", 6);
str[6] = '\0'; // Нулевой символ автоматически не вставляется
std::cout << str << std::endl; // String

В VC++ при использовании функции strncpy() выводится предупреждающее сообщение ("warning C4996"), поэтому вместо функции strncpy() следует использовать функцию strcpy_s(). Прототип функции:
#include <cstring>

errno_t strcpy_s(char *Dst, rsize_t SizeInBytes,
    const char *Src, rsize_t MaxCount);

Листинги на странице http://unicross.narod.ru/cpp/
Функция `strncpy_s()` копирует первые `MaxCount` символов из строки `Src` в строку `Dst` и вставляет нулевой символ. В параметре `SizeInBytes` указывается максимальное количество элементов массива `Dst`. Пример:

```c
const short SIZE = 7;
char str[SIZE];
strncpy_s(str, SIZE, "String", 5);
std::cout << str << std::endl; // String
```

Функция `strcat()` — копирует символы из C-столбца `Source` в конец C-столбца `Dest` и вставляет нулевой символ. В качестве значения функция возвращает указатель на строку `Dest`. Если строка `Dest` имеет недостаточный размер, то произойдет переполнение буфера. Прототип функции:

```c
#include <cstring>

char *strcat(char *Dest, const char *Source);
```

Пример использования функции:

```c
char str[20] = "One";
std::strcat(str, "Two");
std::strcat(str, "Three");
std::cout << str << std::endl; // OneTwoThree
```

Обратите внимание на то, что перед копированием в строке `Dest` обязательно должен быть нулевой символ. Локальные переменные автоматически не инициализируются, поэтому этот код приведет к ошибке:

```c
char str[20]; // Локальная переменная
std::strcat(str, "One"); // Ошибка, нет нулевого символа!
```

Чтобы избавиться от ошибки можно произвести инициализацию строки нулями следующим образом:

```c
char str[20] = {0}; // Инициализация нулями
std::strcat(str, "One"); // Все нормально
std::cout << str << std::endl; // One
```

В VC++ при использовании функции `strcat()` выводится предупреждающее сообщение ("warning C4996"), поэтому вместо функции `strcat()` следует использовать функцию `strcat_s()`. Прототип функции:

```c
#include <cstring>
```

Листинги на странице http://unicross.narod.ru/cpp/
errno_t strcat_s(char *Dst, rsize_t SizeInBytes, 
    const char *Src);

Функция strcat_s() копирует символы из строки Src в конец строки Dst. В параметре SizeInBytes указывается максимальное количество элементов массива Dst. Пример:
const short SIZE = 20;
char str[SIZE] = {0};
strcat_s(str, SIZE, "One");
strcat_s(str, SIZE, "Two");
strcat_s(str, SIZE, "Three");
std::cout << str << std::endl; // OneTwoThree

strncat() — копирует первые Count символов из C-строки Source в конец строки Dest и вставляет нулевой символ. В качестве значения функция возвращает указатель на строку Dest. Обратите внимание на то, что перед копированием в строке Dest обязательно должен быть нулевой символ. Если строка Dest имеет недостаточный размер, то произойдет переполнение буфера. Прототип функции:
#include <cstring>
char *strncat(char *Dest, const char *Source, size_t Count);

Пример использования функции:
char str[20] = "One";
std::strncat(str, "Two", 3);
std::strncat(str, "Three", 3);
std::cout << str << std::endl; // OneTwoThr

В VC++ при использовании функции strncat() выводится предупреждающее сообщение ("warning C4996"), поэтому вместо функции strncat() следует использовать функцию strcat_s(). Прототип функции:
#include <cstring>
errno_t strncat_s(char *Dst, rsize_t SizeInBytes, 
    const char *Src, rsize_t MaxCount);

Функция strncat_s() копирует первые MaxCount символов из строки Src в конец строки Dst. В параметре SizeInBytes указывается максимальное количество элементов массива Dst. Пример:

Листинги на странице http://unicross.narod.ru/cpp/
const short SIZE = 20;
char str[SIZE] = "One";
strncat_s(str, SIZE, "Two", 3);
strncat_s(str, SIZE, "Three", 3);
std::cout << str << std::endl; // OneTwoThr

strtok() — разделяет C-строку str на подстроки, используя в качестве разделителей символы из C-строки Delim. При первом вызове указываются оба параметра. В качестве значения возвращается указатель на первую подстроку или нулевой указатель, если символы-разделители не найдены в C-строке str. Чтобы получить последующие подстроки необходимо каждый раз вызывать функцию strtok(), указывая в первом параметре нулевой указатель, а во втором параметре те же самые символы-разделители. Прототип функции:
#include <cstring>
char *strtok(char *Str, const char *Delim);

Пример использования функции:
char str[] = "a,b.c=d", *p = 0;
p = std::strtok(str, ",.,="");
while (p) {
    std::cout << p << "-";
p = std::strtok(0, ",.,="");
} // Результат: a-b-c-d-
std::cout << std::endl;

В VC++ при использовании функции strtok() выводится предупреждающее сообщение ("warning C4996"), поэтому вместо функции strtok() следует использовать функцию strncat_s(). Прототип функции:
#include <cstring>
char *strncat_s(char *Str, const char *Delim, char **Context);

Первые два параметра аналогичны параметрам функции strtok(). В параметре Context передается адрес указателя. Пример:
char str[] = "a b c,d", *p = 0, *context = 0;
p = strncat_s(str, ",.,", &context);
while (p) {

Листинг на странице http://unicross.narod.ru/cpp/
std::cout << p << "-";
p = strtok_s(0, " ", &context);
} // Результат: a-b-c-d-
std::cout << std::endl;

Практически все стандартные функции, которые мы рассмотрели в этом разделе выводят предупреждающее сообщение "warning C4996", так как по умолчанию функции не производят никакой проверки корректности данных. В этом случае возможно переполнение буфера. Забота о корректности данных полностью лежит на плечах программиста. Если вы уверены в своих действиях, то можно подавить вывод предупреждающих сообщений. Сделать это можно двумя способами:

➤ определить макрос с названием _CRT_SECURE_NO_WARNINGS в самом начале программы, перед подключением заголовочных файлов. Пример:

```cpp
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
```

➤ вставить прамгу warning. Пример:

```cpp
#pragma warning( disable : 4996 )
```

**Обратите внимание**

После директив `#define` и `#pragma` точка с запятой не указывается.

Вместо отключения вывода сообщения можно использовать функции, у которых название заканчивается на "_s" (например, `strncpy_s()` вместо `strncpy()`) или перегрузить функции, чтобы они автоматически преобразовывались в "_s"-аналоги. Для этого необходимо в начале программы перед подключением заголовочных файлов определить макрос `_CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES` со значением 1. Этот макрос не производит преобразование функций, в которых указывается конкретное количество символов (например, функции `strncpy()`). Чтобы перегрузить эти функции необходимо дополнительно определить макрос с названием `_CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES_COUNT` и значением 1. Пример перегрузки приведен в листинге 6.3.

Листинги на странице http://unicross.narod.ru/cpp/
Листинг 6.3. Перегрузка строковых функций

```cpp
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES 1
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES_COUNT 1
#include <iostream>
#include <cstring>

int main() {
    std::strcpy(str1, "String");
    std::cout << str1 << std::endl;  // String
    std::strncpy(str2, "String", 5);
    std::cout << str2 << std::endl;  // String
    std::cin.get();
    return 0;
}
```

Поиск и замена в C-строке

Для поиска и замены в C-строке предназначены следующие функции:

- `strchr()` — ищет в C-строке `Str` первое вхождение символа `ch`. В качестве значения функция возвращает указатель на первый найденный символ в C-строке `Str` или нулевой указатель, если символ не найден. Прототипы функции:

  ```cpp
  #include <cstring>
  char *strchr(char *Str, int Ch);
  const char *strchr(const char *Str, int Ch);
  ```

  Пример использования функции:

  ```cpp
  char str[] = "string string", *p = 0;
  p = std::strchr(str, 'n');
  if (p) {
    std::cout << "Index: " << p - str << std::endl;
  }  // Результат: Index: 4
  ```

- `strrchr()` — ищет в C-строке `Str` последнее вхождение символа `ch`. В качестве значения функция возвращает указатель на символ или нулевой указатель, если символ не найден. Прототипы функции:

  Листинги на странице http://unicross.narod.ru/cpp/
#include <cstring>
char *strrchr(char *Str, int Ch);
const char *strrchr(const char *Str, int Ch);

Пример:
char str[] = "string string", *p = 0;
p = std::strrchr(str, 'n');
if (p) {
    std::cout << "Index: " << p - str << std::endl;
} // Результат: Index: 11

t memchr() — ищет в строке pv первое вхождение символа с. Максимально
просматривается n символов. В качестве значения функция возвращает указатель на
первый найденный символ в строке pv или нулевой указатель, если символ не
найден. Перед присваиванием результата указателю необходимо выполнить
приведение к типу char *. Прототип функции:
#include <cstring>
void *memchr(void *Pv, int C, size_t N);

В качестве примера найдем в строке символ "n" и выведем его индекс, используя
адресную арифметику (вычитание одного указателя из другого):
char str[] = "String", *p = 0;
p = (char *)std::memchr(str, 'n', std::strlen(str));
if (p) {
    std::cout << "Index: " << p - str << std::endl;
} else std::cout << "Not found" << std::endl;

t strpbrk() — ищет в C-строке str первое вхождение одного из символов (нулевой
символ не учитывается), входящих в C-строку control. В качестве значения
функция возвращает указатель на первый найденный символ или нулевой указатель,
если символы не найдены. Прототипы функции:
#include <cstring>
char *strpbrk(char *Str, const char *Control);
const char *strpbrk(const char *Str, const char *Control);

Пример:

Листинги на странице http://unicross.narod.ru/cpp/
char str[] = "string string", *p = 0;
p = std::strpbrk(str, "nr");
if (p) {
    std::cout << "Index: " << p - str << std::endl;
} // Результат: Index: 2 (индекс первой буквы "r")

strcspn() — возвращает индекс первого символа в С-строке str, который
совпадает с одним из символов, входящих в С-строку control. Прототип функции:

#include <cstring>
size_t strcspn(const char *Str, const char *Control);

Пример:
char str[] = "string1 string2";
size_t index;
index = std::strcspn(str, "ingr");
std::cout << "Index: " << index << std::endl;
// Результат: Index: 2 (буква "r")

strspn() — возвращает индекс первого символа в С-строке str, который не
совпадает ни с одним из символов, входящих в С-строку control. Прототип функции:

#include <cstring>
size_t strspn(const char *Str, const char *Control);

Пример:
char str[] = "string1 string2";
size_t index;
index = std::strspn(str, "singtr");
std::cout << "Index: " << index << std::endl;
// Результат: Index: 6 ("1" не входит в "singtr")

strstr() — ищет в С-строке str первое вхождение целого фрагмента из С-строки
SubStr. В качестве значения функция возвращает указатель на первое вхождение
фрагмента в строку или нулевой указатель — в противном случае. Прототипы
функции:

#include <cstring>

Листинги на странице http://unicross.narod.ru/cpp/
char *strstr(char *Str, const char *SubStr);
const char *strstr(const char *Str, const char *SubStr);
Пример:
char str[] = "string1 string2", *p = 0;
p = std::strstr(str, "ing2");
if (p) {
    std::cout << "Index: " << p - str << std::endl;
} // Результат: Index: 11

memset() — заменяет первые Size символов строки Dest символом Val. В качестве значения функция возвращает указатель на строку Dest. Перед присваиванием результату указателю необходимо выполнить приведение к типу char *. Прототип функции:
#include <cstring>
void *memset(void *Dst, int Val, size_t Size);
Пример:
char str[] = "String", *p = 0;
p = (char *)std::memset(str, ' ', 4);
if (!p) std::exit(1);
std::cout << str << std::endl; // ****ng
std::cout << p << std::endl; // ****ng

Сравнение C-строк
Для сравнения C-строк предназначены следующие функции:

strcmp() — сравнивает C-строку Str1 с C-строкой Str2 и возвращает одно из значений:
• отрицательное число — если Str1 меньше Str2;
• 0 — если строки равны;
• положительное число — если Str1 больше Str2.
Сравнение производится с учетом регистра символов. Прототип функции:
#include <cstring>
int strcmp(const char *Str1, const char *Str2);

Листинг на странице http://unicross.narod.ru/cpp/
Пример:
char str1[] = "абв", str2[] = "абв";
int x = 0;
x = std::strcmp(str1, str2);
std::cout << x << std::endl;    // 0
str1[2] = 'ö'; // str1[] = "абв", str2[] = "абв";
x = std::strcmp(str1, str2);
std::cout << x << std::endl;    // -1
str1[2] = 'р'; // str1[] = "абр", str2[] = "абв";
x = std::strcmp(str1, str2);
std::cout << x << std::endl;    // 1

strcmp() — сравнивает Count первых символов в С-строках Str1 и Str2. Если нулевой байт встретится раньше, то значение параметра Count игнорируется. Функция возвращает одно из значений:
- отрицательное число — если Str1 меньше Str2;
- 0 — если строки равны;
- положительное число — если Str1 больше Str2.

Сравнение производится с учетом регистра символов. Прототип функции:
#include <cstring>
int strcmp(const char *Str1, const char *Str2, size_t Count);

Пример:
char str1[] = "абв", str2[] = "абр";
int x = 0;
x = std::strcmp(str1, str2, 2);
std::cout << x << std::endl;    // 0
x = std::strcmp(str1, str2, 3);
std::cout << x << std::endl;    // -1

strcoll() — функция аналогична функции strcmp(), но сравнение производится с учетом значения LC_COLLATE в текущей локали. Например, буква "ё" в диапазон между буквами "е" и "ж" не попадает, так как буква "ё" в кодировке cp1251 имеет

Листинги на странице http://unicross.narod.ru/cpp/
код 184, буква "е" — 229, а буква "ж" — 230. Если сравнение производится с помощью функции strcmp(), то буква "е" будет больше буквы "ё" (229 > 184). При использовании функции strcoll() с настройкой локали "Russian_Russia.1251" буква "е" попадет в диапазон между буквами "е" и "ж", так как используются локальные настройки по алфавиту. Если локаль не настроена, то эта функция эквивалентна функции strcmp(). Сравнение производится с учетом регистра символов. Прототип функции:

```c
#include <cstring>

int strcoll(const char *Str1, const char *Str2);
```

Пример:

```c
std::setlocale(LC_COLLATE, "Russian_Russia.1251");
char str1[] = "е", str2[] = "ё";
int x = std::strcoll(str1, str2);
std::cout << x << std::endl;
// 1 (без настройки локали: е (код 229) больше ё (код 184))
// -1 (после настройки локали: е меньше ё)
```

функция `strxfrm()` — преобразует C-строку `src` в строку специального формата и записывает ее в `dst`. В конец вставляется нулевой символ. Записывается не более `maxcount` символов. Если количество символов `maxcount` меньше необходимого количества символов после преобразования, то содержимое `dst` не определено. В качестве значения функция возвращает число необходимых символов. Чтобы просто получить количество необходимых символов (без учета нулевого символа), то в параметре `maxcount` укажывается число 0, а в параметре `dst` передается нулевой указатель. Прототип функции:

```c
#include <cstring>

size_t strxfrm(char *Dst, const char *Src, size_t MaxCount);
```

Функция strcoll() прежде чем произвести сравнение явно выполняет преобразование переданных строк в строки специального формата. Функция strxfrm() позволяет произвести такое преобразование явным образом. Преобразование строки производится с учетом значения LC_COLLATE в текущей локали. Если локаль не настроена, то просто выполняется копирование. В дальнейшем строки специального формата можно сравнивать с помощью функции strcmp(). Результат сравнения будет соответствовать результату сравнения с

Листинги на странице http://unicross.narod.ru/cpp/
помощью функции `strcoll()`. Функцию `strxfrm()` следует использовать, если сравнение строк производится многократно. Пример:

```cpp
std::setlocale(LC_ALL, "Russian_Russia.1251");
int x = 0;
char str1[] = "е", str2[] = "ё";
char buf1[10] = {0}, buf2[10] = {0};
x = std::strxfrm(0, str1, 0);
std::cout << x << std::endl; // 6 (нужен буфер 6 + 1)
std::strxfrm(buf1, str1, 10);
std::strxfrm(buf2, str2, 10);
x = std::strcmp(str1, str2);
std::cout << x << std::endl; // 1 ("е" больше "ё")
x = std::strcmp(buf1, buf2);
std::cout << x << std::endl; // -1 ("е" меньше "ё")
```

- `strcmp()` — сравнивает C-строки без учета регистра символов. Для русских букв необходимо настроить локаль. Функция доступна только в VC++ и не входит в стандарт. Прототип функции:

```cpp
#include <cstring>
int _strcmp(const char *Str1, const char *Str2);
```

Пример:

```cpp
std::setlocale(LC_CTYPE, "Russian_Russia.1251");
char s1[] = "абв", s2[] = "АБВ";
std::cout << _strcmp(s1, s2) << std::endl; // 0
```

Для сравнения строк можно также использовать функцию `memcmp()`. Функция `memcmp()` сравнивает первые `size` байтов массивов `buf1` и `buf2`. В качестве значения функция возвращает:

- отрицательное число — если `buf1` меньше `buf2`;
- 0 — если массивы равны;
- положительное число — если `buf1` больше `buf2`.

Прототип функции:

```cpp
#include <cstring>
```

Листинги на странице http://unicross.narod.ru/cpp/
int memcmp(const void *Buf1, const void *Buf2, size_t Size);

Пример:
char str1[] = "abc", str2[] = "abc";
int x = 0;
x = std::memcmp(str1, str2, sizeof str2);
std::cout << x << std::endl;  // 0
str1[2] = 'b';
x = std::memcmp(str1, str2, sizeof str2);
std::cout << x << std::endl;  // -1
str1[2] = 'd';
x = std::memcmp(str1, str2, sizeof str2);
std::cout << x << std::endl;  // 1

Функция memcmp() производит сравнение с учетом регистра символов. Если необходимо произвести сравнение без учета символов, то в VC++ можно воспользоваться функцией _memicmp(). Для сравнения русских букв следует настроить локаль. Прототип функции:

#include <cstring>

int _memicmp(const void *Buf1, const void *Buf2, size_t Size);

Предназначение параметров и возвращаемое значение такое же, как у функции memcmp(). Пример использования функции:

std::setlocale(LC_ALL, "Russian_Russia.1251");  // Настройка локаль
char str1[] = "абв", str2[] = "АБВ";
int x = 0;
x = _memicmp(str1, str2, sizeof str2);
std::cout << x << std::endl;  // 0
x = std::memcmp(str1, str2, sizeof str2);
std::cout << x << std::endl;  // 1

Для сравнения строк без учета регистра символов можно использовать функции _stricmp() и _memicmp(). Кроме того, можно воспользоваться функцией stricmp(), но эта функция объявлена устаревшей. Однако, все эти функции не входят в стандарт, следовательно в других компиляторах они могут быть недоступны. По этой причине напишем свою функцию (casestrcmp()) для сравнения без учета регистра символов (листиング 6.4).

Листинги на странице http://unicross.narod.ru/cpp/
Листинг 6.4. Сравнение строк без учета регистра символов

```cpp
#include <iostream>
#include <locale>
#include <cstring>
#include <ctype>

int casncmp(const char *str1, const char *str2);

int main() {
    char s1[] = "абв", s2[] = "АБВ";
    std::cout << std::strcmp(s1, s2) << std::endl; // 1
    std::cout << casncmp(s1, s2) << std::endl;    // 0
    std::cout << casncmp("аё", "аё") << std::endl; // 0
    std::cout << casncmp(s1, "АВ") << std::endl;   // 226
    std::cout << casncmp("аёа", s2) << std::endl;  // -2
    std::cin.get();
    return 0;
}

int casncmp(const char *str1, const char *str2) {
    std::setlocale(LC_CTYPE, "Russian_Russia.1251");
    int ch1 = 0, ch2 = 0, result = 0;
    while (1) {
        ch1 = std::tolower((unsigned char)*str1);
        ch2 = std::tolower((unsigned char)*str2);
        result = ch1 - ch2;
        // Если значения не равны, то возвращаем результат
        if (result) return result;
        // Если достигнут конец строк, то строки равны
        if (!ch1 && !ch2) return 0;
        ++str1;    // Перемещаем указатели
        ++str2;
    }
    return 0;
}
```

Листинги на странице http://unicros.s.narod.ru/cpp/
Функция `casecmp()` принимает в качестве параметров адреса двух строк (`str1` и `str2`) и возвращает 0 — если строки равны, отрицательное число — если `str1` меньше `str2`, и положительное число — если `str1` больше `str2`. Чтобы в качестве параметра можно было передать строковую константу, перед параметрами указываем ключевое слово `const` (например, `const char *str1`). Внутри функции производим настройку локали и перебор всех символов в строках в бесконечном цикле. Внутри цикла оба символа приводим к одному регистру с помощью функции `tolower()`, а затем символ первой строки вычитаем из символа второй строки. Если результат вычитания не равен нулю, то возвращаем это значение из функции. Если результат равен нулю, то проверяем не достигнут ли конец двух строк. В этом случае строки равны и можно вернуть значение 0. Если конец строк не достигнут, то перемещаем указатели на следующий символ и опять производим сравнение.

**Класс string**

C-строки являются массивами, поэтому работать с такими строками не очень удобно. Например, нельзя присвоить строку в двойных кавычках после объявления строки:

```c
char str1[] = "abc"; // Нормально
char str2[7];
str2 = "abc"; // Ошибка
```

Кроме того, нельзя увеличить размер строки после объявления или соединить две строки в одну с помощью операторов:

```c
char str1[] = "abc", str2[7] = "def";
char str3[] = str1 + str2; // Ошибка
```

Класс `std::string` предоставляет более удобный интерфейс доступа, который избавляет программиста от необходимости следить за размером символьного массива и предоставляет множество методов для обработки строки. При необходимости можно преобразовать объект класса `std::string` в C-строку. Название класса `std::string` является лишь псевдонимом шаблонного класса `basic_string`:

```c
typedef basic_string<char, char_traits<char>, allocator<char> > string;
```

Для использования класса `std::string` необходимо подключить файл `string`:

```c
#include <string>
```

**Объявление и инициализация строки**

Объявить объект класса `std::string` можно следующими способами:

объявить экземпляр класса без инициализации. Для этого перед названием переменной указывается название класса. Размер строки указывать не нужно. Пример объявления без инициализации:

```cpp
std::string str;
```

указать C-стрику внутри круглых скобок или после оператора =, тем самым производя инициализацию объекта. Пример:

```cpp
std::string str1("String1");
std::string str2 = "String2";
```

Если во втором параметре передать число, то можно присвоить только часть строки, а не всю строку:

```cpp
std::string str("String", 3); // Str
```

указать объект класса string внутри круглых скобок или после оператора =:

```cpp
std::string str1("String");
std::string str2(str1);
std::string str3 = str1;
```

Можно присвоить не все содержимое объекта, а его часть. Для этого во втором параметре следует передать начальный индекс, а в третьем参数е конечный индекс. Если конечный индекс не указан, то копируется фрагмент до конца строки. Пример:

```cpp
std::string str1("String");
std::string str2(str1, 3); // ing
std::string str3(str1, 0, 3); // Str
```

указать в первом параметре количество символов, а во втором параметре символ-заполнитель:

```cpp
std::string str(5, 's'); // sssss
```

Присвоить значение переменной уже после объявления можно с помощью оператора =. Следить за размером строки не нужно. Если производится присваивание объекта класса string, то создается копия исходного объекта, а не присваивается ссылка на него. Пример:

```cpp
std::string str1, str2;
str1 = "string"; // Присваивание C-стрики
str1 = "new string"; // Следить за размером строки не нужно
```

Листинги на странице http://unicross.narod.ru/cpp/
str2 = str1;    // Присваивание объекта класса string
Вместо оператора = для присваивания значения можно воспользоваться методом assign(). Название метода указывается после имени переменной через точку.
Прототипы метода:

```cpp
string &assign(size_type Count, char Ch);
string &assign(const char *Ptr);
string &assign(const char *Ptr, size_type Count);
string &assign(const string &Right);
string &assign(const string &Right, size_type Roff,
               size_type Count);
template<class _It> string &assign(_It First, _It Last);
string &assign(const_pointer First, const_pointer Last);
string &assign(const_iterator First, const_iterator Last);
```

Первый прототип присваивает Count символов ch. Второй прототип позволяет присвоить C-строку целиком, а третий прототип — только Count первых символов из C-строки ptr. Четвертый прототип позволяет присвоить объект класса string целиком, а пяthy прототип — только Count символов, начиная с индекса Roff. Остальные прототипы позволяют указать начальную First и конечную Last позиции с помощью _iterators. Пример:

```cpp
std::string str1, str2("BC");
char str3[] = "DEF";
str1.assign(2, '*');    // Прототип 1
std::cout << str1 << std::endl;  // **
str1.assign(str3);    // Прототип 2
std::cout << str1 << std::endl;  // DEF
str1.assign(str3, 2);    // Прототип 3
std::cout << str1 << std::endl;  // DEF
str1.assign(str2);    // Прототип 4
std::cout << str1 << std::endl;  // BC
str1.assign(str2, 0, 1);    // Прототип 5
std::cout << str1 << std::endl;  // B
str1.assign(str2.begin(), str2.end());    // Прототип 6
std::cout << str1 << std::endl;  // BC
```

Листинги на странице http://unicross.narod.ru/cpp/
Ввод и вывод строк

Объекты класса string можно вводить и выводить с помощью операторов >> и << соответственно. Следует учитывать, что с помощью оператора >> можно ввести только фрагмент до первого пробельного символа. Пример ввода и вывода значений:

```cpp
std::string str;
std::cout << "str = ";
std::cin >> str; // Получаем строку до первого пробела
std::cout << str << std::endl; // Выводим строку
```

Чтобы ввести строку необходимо воспользоваться функцией getline(). В первом параметре указывается объект std::cin, во втором параметре — объект класса string, а в третьем параметре — символ, до которого производится считывание. Если третий параметр не указан, то считывание производится до символа перевода строки. Пример ввода строки:

```cpp
std::string str;
std::cout << "str = ";
std::getline(std::cin, str, '
'); // Получаем строку
std::cout << str << std::endl; // Выводим строку
```

Преобразование объекта в C-строку или в массив символов

Для преобразования объекта класса string в другой тип предназначены следующие методы:

- `c_str()` — возвращает указатель типа const char *. Его можно использовать в тех случаях, когда необходимо передать C-строку (например, в функцию), вместо объекта класса string. Изменять такую строку нельзя. Кроме того, следует учитывать, что после изменения объекта указатель может стать некорректным, поэтому после изменения объекта необходимо обновить значение указателя. Прототип метода:

```cpp
#include <string>
const char *c_str() const;
```

Пример преобразования объекта класса string в C-строку:

```cpp
std::string str1("String");
char str2[80];
strcpy_s(str2, 80, str1.c_str());
```

Листинги на странице http://unicross.narod.ru/cpp/
data() — возвращает указатель типа const char *. В отличие от метода c_str() метод data() возвращает указатель на массив символов (нулевой символ не добавляется), а не на C-строку. Если необходимо получить строку, заканчивающуюся нулевым символом, то следует использовать метод c_str(). Следует учитывать, что после изменения объекта указатель может стать некорректным, поэтому после изменения объекта необходимо обновить значение указателя. Прототип метода:
#include <string>
const char *data() const;

Пример:
std::string str("String");
const char *p = 0;
p = str.data();
std::cout << p[0] << std::endl; // S
std::cout << p[5] << std::endl; // g

copy() — копирует Count символов, начиная с индекса Off, в символьный массив Ptr. Если индекс не указан, то символы копируются с начала строки. Нулевой символ автоматически не добавляется. В качестве значения метод возвращает количество скопированных символов. Прототип метода:
#include <string>
size_type copy(char *Ptr, size_type Count,
               size_type Off=0) const;

Пример использования метода copy():
std::string str1("String");
char str2[20] = {0}, str3[20] = {0};
std::basic_string< char >::size_type count;
count = str1.copy(char::size_type count;
count = str1.copy(str2, 5);
std::cout << count << std::endl; // 5
std::cout << str2 << std::endl; // Strin
str1.copy(str3, 3, 1);
std::cout << str3 << std::endl; // tri

Листинги на странице http://unicross.narod.ru/cpp/
В VC++ при использовании метода `copy()` выводится предупреждающее сообщение ("warning C4996"), поэтому вместо метода `copy()` следует использовать метод `_copy_s()`. Прототип метода:

```cpp
#include <cstring>

size_type _Copy_s(char *Dest, size_type Dest_size,
                  size_type Count, size_type Off=0) const;
```

Метод `_Copy_s()` копирует `Count` символов, начиная с индекса `Off`, в символьный массив `Dest`. Если индекс не указан, то символы копируются с начала строки. В параметре `Dest_size` указывается размер символьного массива `Dest`. Пример:
```
std::string str1(“String”);
char str2[20] = {0}, str3[20] = {0};
std::basic_string<
char>::size_type count;

count = str1._Copy_s(str2, sizeof str2, 5);
std::cout << count << std::endl; // 5
std::cout << str2 << std::endl; // Str

str1._Copy_s(str3, sizeof str3, 3, 1);
std::cout << str3 << std::endl; // tri
```

### Получение и изменение размера строки

Для получения и изменения размера строки предназначены следующие методы:

- `size()` и `length()` — возвращают текущее количество символов в строке. Прототипы методов:
  ```cpp
  #include <string>

  size_type size() const;
  size_type length() const;
  ```
  Пример:
  ```cpp
  std::string str(“String”);
  std::cout << str.size() << std::endl; // 6
  std::cout << str.length() << std::endl; // 6
  ```

- `capacity()` — возвращает количество символов, которое можно записать в строку без перераспределения памяти. Прототип метода:

  ```cpp
  ```

#include <string>
size_type capacity() const;

Пример:
std::string str("String");
std::cout << str.size() << std::endl; // 6
std::cout << str.capacity() << std::endl; // 15
str += " string2 string3"
std::cout << str.size() << std::endl; // 22
std::cout << str.capacity() << std::endl; // 31

reserve() — позволяет задать минимальное количество символов, которое можно записать в строку без перераспределения памяти. Как видно из предыдущего примера, выделение дополнительной памяти производится автоматически с некоторым запасом. Если дозапись в строку производится часто, то это может снизить эффективность программы, так как перераспределение памяти будет выполнено несколько раз. Поэтому, если количество символов заранее известно, то следует указать его с помощью метода reserve(). Прототип метода:
#include <string>
void reserve(size_type Newcap = 0);

Пример указания минимального размера строки:
std::string str("String");
str.reserve(50);
std::cout << str.size() << std::endl; // 6
std::cout << str.capacity() << std::endl; // 63
str += " string2 string3"
std::cout << str.size() << std::endl; // 22
std::cout << str.capacity() << std::endl; // 63

shrink_to_fit() — уменьшает размер строки до минимального значения. Прототип метода:
#include <string>
void shrink_to_fit();

Пример:
std::string str("String");
str.reserve(50);
std::cout << str.capacity() << std::endl; // 63
str.shrink_to_fit();
std::cout << str.capacity() << std::endl; // 15

resize() — задает количество символов в строке, равное числу Newsize. Если указанное количество символов меньше текущего количества, то лишние символы будут удалены. Если количество символов необходимо увеличить, то в параметре ch можно указать символ, который заполнит новое пространство. Прототипы метода:
#include <string>
void resize(size_type Newsize);
void resize(size_type Newsize, char Ch);

Пример:
std::string str("String");
str.resize(4);
std::cout << str << std::endl; // Str
str.resize(8, '*');
std::cout << str << std::endl; // Str***

clear() — удаляет все символы. Прототип метода:
#include <string>
void clear();

Пример:
std::string str("String");
str.clear();
std::cout << str.size() << std::endl; // 0

empty() — возвращает значение true, если строка не содержит символов, и false — в противном случае. Прототип метода:
#include <string>
bool empty() const;

Пример:
std::string str("String");
if (str.empty()) {
    std::cout << "Строка пустая" << std::endl;
} else {
    std::cout << "Строка содержит символы" << std::endl;
}

max_size() — возвращает максимальное количество символов, которое может содержаться в строке. Прототип метода:

```cpp
#include <string>
size_type max_size() const;
```

Пример:

```cpp
std::string str("String");
std::cout << str.max_size() << std::endl; // 4294967294
```

### Получение и изменение содержимого строки

К любому символу объекта класса string можно обратиться как к элементу массива. Достаточно указать его индекс в квадратных скобках. Нумерация начинается с нуля. Можно как получить символ, так и изменить его. Если индекс выходит за границы диапазона, то возвращаемое значение не определено. Пример доступа к символу по индексу:

```cpp
std::string str("C");
std::cout << str[0] << std::endl; // C
str[0] = 'D';
std::cout << str[0] << std::endl; // D
```

Получить количество символов в строке позволяет метод size(). В качестве примера выведем все символы по одному на строке:

```cpp
std::string str("string");
for (int i=0, c = str.size(); i<c; ++i) {
    std::cout << str[i] << std::endl;
}
```

Для объектов класса string определена операция конкатенации (объединения строк). Оператор + позволяет объединить:

→ два объекта класса string:

std::string str1("A"), str2("B"), str3;
str3 = str1 + str2; // AB

» объект класса string с символом:
std::string str1("A"), str2, str3;
char ch = 'B';
str2 = str1 + ch; // AB
str3 = ch + str1; // BA

» объект класса string с C-строкой:
std::string str1("A"), str2, str3;
str2 = str1 + "B"; // AB
str3 = "B" + str1; // BA

Обратите внимание на то, что оператор + для объединения двух C-строк применять нельзя. Чтобы объединить две C-строки они указываются без операторов между ними.
Пример:
std::string str1("A"), str2, str3;
str2 = "C" + "B"; // Ошибка
str2 = "C" + str1 + "B"; // Нормально
str3 = "C" "B"; // Нормально

Помимо оператора + доступен оператор +=, который производит конкатенацию с присваиванием:
std::string str1("A"), str2("B");
str1 += "C"; // AC
str2 += str1; // BAC
str2 += str1 + "D"; // BACACD

Для получения и изменения содержимого строки предназначены следующие методы:

» at() — возвращает ссылку на символ, расположенный по индексу off. Метод позволяет как получить символ, так и изменить его. Если индекс выходит за границы диапазона, то метод генерирует исключение. Прототипы метода:
#include <string>
reference at(size_type Off);
const_reference at(size_type Off) const;

Пример:

Листинги на странице http://unicross.narod.ru/cpp/
std::string str1("String1");
const std::string str2("String2");
std::basic_string<char>::reference rstr1 = str1.at(0);
std::basic_string<char>::const_reference rstr2 = str2.at(6);
rstr1 = 's';
str1.at(6) = '3';
std::cout << str1 << std::endl; // string3
std::cout << str1.at(1) << std::endl; // t
std::cout << rstr2 << std::endl; // 2 (метод at())
std::cout << str2[6] << std::endl; // 2 (оператор [])

➤ front() — возвращает ссылку на первый символ в строке. Метод позволяет как получить символ, так и изменить его. Прототипы метода:
#include <string>
reference front();
const_reference front() const;

Пример:
std::string str("String");
str.front() = 's';
std::cout << str.front() << std::endl; // s

➤ back() — возвращает ссылку на последний символ в строке. Метод позволяет как получить символ, так и изменить его. Прототипы метода:
#include <string>
reference back();
const_reference back() const;

Пример:
std::string str("String");
str.back() = 'G';
std::cout << str.back() << std::endl; // G

➤ substr() — возвращает фрагмент строки, состоящий из Count символов, начиная с индекса Off. Если начальный индекс не задан, то предполагается, что индекс равен

http://unicross.narod.ru/cpp/
0. Если длина не задана, то возвращается фрагмент, начиная с индекса Off до конца строки. Прототип метода:

```cpp
#include <string>

string substr(size_type Off=0, size_type Count=npos) const;
```

Пример:

```cpp
std::string str1("12345"), str2;
str2 = str1.substr();
std::cout << str2 << std::endl; // 12345
str2 = str1.substr(2);
std::cout << str2 << std::endl; // 345
str2 = str1.substr(2, 2);
std::cout << str2 << std::endl; // 34
```

- **push_back()** — добавляет символ Ch в конец строки. Прототип метода:

```cpp
#include <string>

void push_back(char Ch);
```

Пример:

```cpp
std::string str("String");
str.push_back('1');
std::cout << str << std::endl; // String1
```

- **append()** — добавляет символы и строки в конец исходной строки. Метод можно использовать для конкатенации вместо операторов + и +=. Прототипы метода:

```cpp
#include <string>

string &append(size_type Count, char Ch);
string &append(const char *Ptr);
string &append(const char *Ptr, size_type Count);
string &append(const string &Right);
string &append(const string &Right, size_type Roff,
size_type Count);

template<class _It> string &append(_It First, _It Last);
string &append(const_pointer First, const_pointer Last);
```

Листинги на странице http://unicross.narod.ru/cpp/
string &append(const_iterator First, const_iterator Last);

Первый прототип добавляет count символов ch в конец строки. Второй прототип позволяет добавить C-строку целиком, а третий прототип — только count первых символов из C-строки ptr. Четвертый прототип позволяет добавить объект класса string целиком, а пятый прототип — только count символов, начиная с индекса roff. Остальные прототипы позволяют указать начальную First и конечную Last позиции с помощью итераторов (указателей). Пример:

```c++
std::string str1("A"), str2("BC");
char str3[] = "DEF";
str1.append(2, '+'); // Прототип 1
std::cout << str1 << std::endl; // A**
str1.append(str3); // Прототип 2
std::cout << str1 << std::endl; // A**DEF
str1.append(str3, 2); // Прототип 3
std::cout << str1 << std::endl; // A**DEFDE
str1 = "A";
str1.append(str2); // Прототип 4
std::cout << str1 << std::endl; // ABC
str1.append(str2, 0, 1); // Прототип 5
std::cout << str1 << std::endl; // ABCB
str1.append(str2.begin(), str2.end()); // Прототип 6
std::cout << str1 << std::endl; // ABCBVC
```

→ insert() — вставляет символы и строки в позицию указанную индексом или итератором (указателем). Остальные элементы сдвигаются к концу строки. Прототипы метода:

```c++
#include <string>
string &insert(size_type Off, size_type Count, char Ch); // 1
iterator insert(const_iterator Where, char Ch); // 2
void insert(const_iterator Where, size_type Count,
            char Ch); // 3
string &insert(size_type Off, const char *Ptr); // 4
string &insert(size_type Off, const char *Ptr,
```
size_type Count); // 5
string &insert(size_type Off, const string &Right); // 6
string &insert(size_type Off, const string &Right,
               size_type Roff, size_type Count); // 7

template<class _It>
void insert(const_iterator Where, _It First, _It Last); // 8
void insert(const_iterator Where, const_pointer First,
            const_pointer Last); // 9
void insert(const_iterator Where, const_iterator First,
            const_iterator Last); // 10
iterator insert(const_iterator Where); // 11

Прототип 1 вставляет Count символов Ch в позицию с индексом Off. Прототип 2
вставляет символ Ch в позицию на которую указывает константный итератор Where. Прототип 3 вставляет Count символов Ch в позицию на которую указывает константный итератор Where. Прототип 4 позволяет вставить C-строку Ptr целиком в позицию с индексом Off, а прототип 5 — только count первых символов из C-строки Ptr. Прототип 6 позволяет вставить объект класса string Right целиком в позицию с индексом Off, а прототип 7 — только Count символов, начиная с индекса Roff. Прототипы 8, 9 и 10 позволяют указать начальную First и конечную Last позиции с помощью итераторов (указателей). Символы будут вставлены в позицию на которую указывает константный итератор Where. Прототип 11 вставляет нулевой символ в позицию на которую указывает константный итератор Where. Пример:

std::string str1("12345"), str2("###");
char str3[] = "+++";
str1.insert(2, 5, '*'); // Прототип 1
std::cout << str1 << std::endl; // 12*****345
str1 = "12345";
str1.insert(str1.begin() + 2, '*'); // Прототип 2
std::cout << str1 << std::endl; // 12*345
str1 = "12345";
str1.insert(str1.begin() + 2, 5, '*'); // Прототип 3
std::cout << str1 << std::endl; // 12*****345

Листинги на странице http://unicross.narod.ru/cpp/
str1 = "12345";
str1.insert(2, str3);         // Прототип 4
std::cout << str1 << std::endl; // 12+++345
str1 = "12345";
str1.insert(2, str3, 1);      // Прототип 5
std::cout << str1 << std::endl; // 12+345
str1 = "12345";
str1.insert(2, str2);         // Прототип 6
std::cout << str1 << std::endl; // 12###345
str1 = "12345";
str1.insert(2, str2, 0, 1);    // Прототип 7
std::cout << str1 << std::endl; // 12#345
str1 = "12345";
str1.insert(str1.begin() + 2,
            str2.begin(), str2.end()); // Прототип 8
std::cout << str1 << std::endl; // 12###345
str1 = "12345";
str1.insert(str1.begin() + 2); // Прототип 11
std::cout << str1.c_str() << std::endl; // 12

pop_back() — удаляет последний символ в строке. Прототип метода:
#include <string>
void pop_back();

Пример:
std::string str("12345");
str.pop_back();
std::cout << str << std::endl;     // 1234
str.pop_back();
std::cout << str << std::endl;     // 123

erase() — удаляет один символ или фрагмент. Остальные элементы сдвигаются к началу строки. Прототипы метода:
#include <string>

Листинги на странице http://unicross.narod.ru/cpp/
iterator erase(const_iterator Where);
iterator erase(const_iterator First, const_iterator Last);
string &erase(size_type Off=0, size_type Count=npos);

Первый прототип удаляет символ, на который указывает итератор (указатель) Where. Второй прототип позволяет удалить фрагмент строки между итераторами First и Last. Третий прототип удаляет фрагмент длиной Count, начинающийся с индекса Off. Если начальный индекс не указан, то предполагается, что индекс равен 0. Если длина не указана, то удаляется фрагмент от индекса Off до конца строки.
Пример:

```cpp
std::string str("12345");
str.erase(str.begin() + 2); // Прототип 1
std::cout << str << std::endl; // 1245
str = "12345";
str.erase(str.begin(), str.begin() + 2); // Прототип 2
std::cout << str << std::endl; // 345
str = "12345";
str.erase(3); // Прототип 3
std::cout << str << std::endl; // 123
str = "12345";
str.erase(0, 2); // Прототип 3
std::cout << str << std::endl; // 345
```

→ swap() — меняет содержимое двух строк местами. Прототип метода:
```cpp
#include <string>

void swap(string &Right);
```
Пример:
```cpp
std::string str1("12345"), str2("67890");
str1.swap(str2);
std::cout << str1 << std::endl; // 67890
std::cout << str2 << std::endl; // 12345
```

→ replace() — заменяет фрагмент строки отдельным символом или подстрокой. Если вставляемая подстрока меньше фрагмента, то остальные символы сдвигаются к

началу строки, а если больше, то раздвигаются таким образом, чтобы вместить всю вставляемую подстроку. Прототипы метода:

```cpp
#include <string>

string &replace(size_type Off, size_type NO,
                 size_type Count, char Ch); // 1
string &replace(const_iterator First, const_iterator Last,
                 size_type Count, char Ch); // 2
string &replace(size_type Off, size_type NO,
                 const char *Ptr); // 3
string &replace(size_type Off, size_type NO,
                 const char *Ptr, size_type Count); // 4
string &replace(const_iterator First, const_iterator Last,
                 const char *Ptr); // 5
string &replace(const_iterator First, const_iterator Last,
                 const char *Ptr, size_type Count); // 6
string &replace(size_type Off, size_type NO,
                 const string &Right); // 7
string &replace(size_type Off, size_type NO,
                 const string &Right, size_type Roff,
                 size_type Count); // 8
string &replace(const_iterator First, const_iterator Last,
                 const string &Right); // 9

template<class _It>
string &replace(const_iterator First, const_iterator Last,
                 _It First2, _It Last2); // 10
string &replace(const_iterator First, const_iterator Last,
                 const_pointer First2, const_pointer Last2); // 11
string &replace(const_iterator First, const_iterator Last,
                 const_iterator First2, const_iterator Last2); // 12
```

Прототип 1 вставляет Count символов Ch вместо фрагмента длиной NO, начинаяющего с индекса Off. Прототип 2 вставляет Count символов Ch вместо

Листинги на странице http://unicross.narod.ru/cpp/
фрагмента, ограниченного элементами на которые указывают итераторы First и Last. Прототип 3 позволяет вставить C-строку ptr целиком вместо фрагмента длиной no, начинаяющегося с индекса off, а прототип 4 — только count первых символов из C-строки ptr. Прототип 5 вставляет C-строку ptr целиком вместо фрагмента, ограниченного элементами на которые указывают итераторы First и Last, а прототип 6 — только count первых символов из C-строки ptr. Прототип 7 позволяет вставить объект класса string Right целиком вместо фрагмента длиной no, начинаяющего с индекса off, а прототип 8 — только count символов, начиная с индекса roff. Прототип 9 вставляет объект класса string Right целиком вместо фрагмента, ограниченного элементами на которые указывают итераторы First и Last. Прототипы 10, 11 и 12 позволяют указать начальную First и конечную Last позиции заменяемого фрагмента с помощью итераторов. Вставляемая подстрока задается с помощью итераторов First2 и Last2. Пример:

```cpp
std::string str1("12345"), str2("67890");
char str3[] = "abcd";
str1.replace(0, 3, 4, '+' );                   // Прототип 1
std::cout << str1 << std::endl;               // ****45
str1.replace(str1.begin(), str1.begin()+3,
            3, '+');                            // Прототип 2
std::cout << str1 << std::endl;               // +++*45
str1.replace(0, 4, str3 );                   // Прототип 3
std::cout << str1 << std::endl;               // abcd45
str1.replace(0, 4, str3, 3 );                // Прототип 4
std::cout << str1 << std::endl;               // abc45
str1.replace(str1.begin(), str1.begin()+3,
            "0123");                            // Прототип 5
std::cout << str1 << std::endl;               // 012345
str1.replace(str1.begin(), str1.begin()+1,
            str3, 2);                            // Прототип 6
std::cout << str1 << std::endl;               // ab12345
str1.replace(0, 5, str2);                   // Прототип 7
std::cout << str1 << std::endl;               // 6789045
str1.replace(0, 5, str2, 1, 2 );             // Прототип 8
```

Листинги на странице http://unicross.narod.ru/cpp/
Поиск в строке
Перечислим методы, предназначенные для поиска в строке:

*find*() — производит поиск символа или фрагмента с начала строки. Возвращает индекс первого совпадения, если фрагмент найден, или значение константы `std::string::npos` — в противном случае. Поиск зависит от регистра символов.

Прототипы метода:

```cpp
#include <string>

size_type find(char Ch, size_type Off=0) const;
size_type find(const char *Ptr, size_type Off=0) const;
size_type find(const char *Ptr, size_type Off,
               size_type Count) const;
size_type find(const string &Right, size_type Off=0) const;
```

Первый прототип ищет в строке символ Ch, начиная с индекса off. Второй прототип позволяет произвести поиск С-строки целиком, начиная с индекса off, а третий прототип — только count первых символов С-строки. Четвертый прототип ищет объект класса string, начиная с индекса off. Если индекс off не задан, то предполагается, что его значение равно 0. Пример:

```cpp
std::string str("123454321"), str2("2");
std::cout << str.find('2') << std::endl; // 1
std::cout << (int)str.find('6') << std::endl; // -1
std::cout << str.find('2', 3) << std::endl; // 7
std::cout << str.find("2") << std::endl; // 1
std::cout << (int)str.find("6") << std::endl; // -1
std::cout << str.find("2", 3) << std::endl; // 7
```

Листинги на странице http://unicross.narod.ru/cpp/
std::cout << str.find("200", 3, 1) << std::endl; // 7
std::cout << str.find(str2) << std::endl; // 1
std::cout << str.find(str2, 3) << std::endl; // 7

rfind() — производит поиск символа или фрагмента с конца строки. Возвращает индекс первого с конца совпадения, если фрагмент найден, или значение константы std::string::npos — в противном случае. Поиск зависит от регистра символов.

Прототипы метода:
#include <string>
size_type rfind(char Ch, size_type Off=npos) const;
size_type rfind(const char *Ptr, size_type Off=npos) const;
size_type rfind(const char *Ptr, size_type Off,
    size_type Count) const;
size_type rfind(const string &Right, size_type Off=npos) const;

Первый прототип ищет в строке символ Ch, начиная с индекса Off. Второй прототип позволяет произвести поиск С-строки целиком, начиная с индекса Off, а третий прототип — только count первых символов C-строки. Четвертый прототип ищет объект класса string, начиная с индекса Off. Поиск производится от индекса Off к началу строки. Если индекс Off не задан, то предполагается, что его значение равно индексу последнего элемента. Пример:
std::string str("123454321"), str2("2");
std::cout << str.rfind('2') << std::endl; // 7
std::cout << (int)str.rfind('6') << std::endl; // -1
std::cout << str.rfind('2', 3) << std::endl; // 1
std::cout << str.rfind("2") << std::endl; // 7
std::cout << (int)str.rfind("6") << std::endl; // -1
std::cout << str.rfind("2", 3) << std::endl; // 1
std::cout << str.rfind("200", 3, 1) << std::endl; // 1
std::cout << str.rfind(str2) << std::endl; // 7
std::cout << str.rfind(str2, 3) << std::endl; // 1

find_first_of() — производит поиск символа или какого-либо символа, входящего в указанную строку. Поиск производится с начала строки. Метод возвращает индекс первого совпадения, если символ найден, или значение

Листинги на странице http://unicross.narod.ru/cpp/
константы std::string::npos — в противном случае. Поиск зависит от регистра символов. Прототипы метода:

```cpp
#include <string>

size_type find_first_of(char Ch, size_type Off=0) const;
size_type find_first_of(const char *Ptr,
                        size_type Off=0) const;
size_type find_first_of(const char *Ptr, size_type Off,
                        size_type Count) const;
size_type find_first_of(const string &Right,
                        size_type Off=0) const;
```

Первый прототип ищет в строке символ Ch, начиная с индекса Off. Второй прототип позволяет найти совпадение с каким-либо символом из C-строки, начиная с индекса Off, а третий прототип — учитывает только count первых символов C-строки. Четвертый прототип ищет совпадение с каким-либо символом из объекта класса string, начиная с индекса Off. Если индекс Off не задан, то предполагается, что его значение равно 0. Пример:

```cpp
std::string str("123454321");
std::string str2("862");
std::cout << str.find_first_of('2') << std::endl; // 1
std::cout << (int)str.find_first_of('6') << std::endl; // -1
std::cout << str.find_first_of('2', 3) << std::endl; // 7
std::cout << str.find_first_of("862") << std::endl; // 1
std::cout << str.find_first_of("862", 3) << std::endl; // 7
std::cout << str.find_first_of("250", 3, 2) << std::endl; // 4
std::cout << str.find_first_of(str2) << std::endl; // 1
std::cout << str.find_first_of(str2, 3) << std::endl; // 7
```

После включения заголовочного файла `#include <string>`, можно использовать функции `find_first_of` для поиска первого символа, который совпадает с указанным параметром. Примеры:

```cpp
find_first_not_of() — возвращает индекс первого символа, который не совпадает ни с одним из символов, входящих в указанную строку. Если ничего не найдено, то возвращается значение константы std::string::npos. Поиск зависит от регистра символов. Прототипы метода:

```cpp
#include <string>

size_type find_first_not_of(char Ch, size_type Off=0) const;
size_type find_first_not_of(const char *Ptr,
                            size_type Off=0) const;
```

Листинги на странице http://unicross.narod.ru/cpp/
size_type Off=0) const;
size_type find_first_not_of(const char *Ptr, size_type Off,
    size_type Count) const;
size_type find_first_not_of(const string &Right,
    size_type Off=0) const;

Первый прототип ищет в строке символ, который не совпадает с символом Ch.
Второй прототип возвращает индекс первого символа, который не совпадает ни с
одним из символов, входящих в C-строку Ptr, а третий прототип — учитывает
только Count первых символов C-строки Ptr. Четвертый прототип возвращает
индекс первого символа, который не совпадает ни с одним из символов, входящих в
объект класса string. Поиск производится начиная с индекса Off. Если индекс Off
не задан, то предполагается, что его значение равно 0. Пример:
std::string str("123454321"), str2("8621");
std::cout << str.find_first_not_of('2') << std::endl; // 0
std::cout << str.find_first_not_of('2', 3) << std::endl; // 3
std::cout << str.find_first_not_of("8621") << std::endl; // 2
std::cout << str.find_first_not_of("8621", 3)
    << std::endl; // 3
std::cout << str.find_first_not_of("250", 3, 2)
    << std::endl; // 3
std::cout << str.find_first_not_of(str2) << std::endl; // 2
std::cout << str.find_first_not_of(str2, 3) << std::endl; // 3

find_last_of() — метод аналогичен методу find_first_of(), но поиск
производится с конца строки, а не с начала. Прототипы метода:
#include <string>
size_type find_last_of(char Ch, size_type Off=npos) const;
size_type find_last_of(const char *Ptr,
    size_type Off=npos) const;
size_type find_last_of(const char *Ptr, size_type Off,
    size_type Count) const;
size_type find_last_of(const string &Right,
    size_type Off=npos) const;

Листинги на странице http://unicross.narod.ru/cpp/
find_last_not_of() — метод аналогичен методу find_first_not_of(), но поиск производится с конца строки, а не с начала. Прототипы метода:

```cpp
#include <string>

size_type find_last_not_of(char Ch, size_type Off=npos) const;
size_type find_last_not_of(const char *Ptr,
                             size_type Off=npos) const;
size_type find_last_not_of(const char *Ptr, size_type Off,
                             size_type Count) const;
size_type find_last_not_of(const string &Right,
                             size_type Off=npos) const;
```

В качестве примера проверим состоит ли строка только из цифр или содержит также буквы:

```cpp
std::string str("5487");
int index = 0;
index = str.find_last_not_of("0123456789");
if (index == std::string::npos) {
    std::cout << "Строка содержит только цифры" << std::endl;
}
else std::cout << "Нет" << std::endl;
```

### Сравнение строк

Для сравнения двух объектов класса string или объекта класса string и C-строки предназначены операторы == (равно), != (не равно), < (меньше), > (больше), <= (меньше или равно) и >= (больше или равно). Пример сравнения строк:

```cpp
std::string str1("abc"), str2("abcd");
if (str1 == str2) {
    std::cout << "str1 == str2" << std::endl;
}
else if (str1 < str2) {
    std::cout << "str1 < str2" << std::endl;
}
else if (str1 > str2) {
    std::cout << "str1 > str2" << std::endl;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
} // Результат: str1 < str2

Обратите внимание на то, что сравнивать с помощью этих операторов С-строки нельзя, так как будут сравниваться адреса, а не символы в строках. Для сравнения С-строк необходимо использовать специальные функции, например, strcmp(). Чтобы операторы работали корректно, необходимо, чтобы справа или слева от оператора находился объект класса string. Пример:

```
char a[] = "abc", b[] = "abc";
std::string c("abc");
std::cout << (a == b) << std::endl; // Сравниваются адреса!!!
std::cout << std::strcmp(a, b) << std::endl; // OK (равны)
std::cout << (a == c) << std::endl; // OK (равны)
std::cout << (c == b) << std::endl; // OK (равны)
```

Для сравнения объекта класса string с другим объектом класса string или С-строкой можно использовать метод compare(). Прототипы метода:

```
#include <string>

int compare(const char *Ptr) const; // 1
int compare(size_type Off, size_type NO, const char *Ptr) const; // 2
int compare(size_type Off, size_type NO, const char *Ptr,
            size_type Count) const; // 3
int compare(const string &Right) const; // 4
int compare(size_type Off, size_type NO,
            const string &Right) const; // 5
int compare(size_type Off, size_type NO, const string &Right,
            size_type Roff, size_type Count) const; // 6
```

Прототип 1 сравнивает объект класса string с С-строкой. Прототип 2 сравнивает фрагмент объекта класса string длиной NO начиная с индекса Off с С-строкой, а прототип 3 позволяет дополнительно указать длину фрагмента в С-строке с начала строки. Прототип 4 сравнивает два объекта класса string. Прототип 5 сравнивает фрагмент объекта класса string длиной NO начиная с индекса Off с другим объектом класса string, а прототип 6 позволяет дополнительно указать длину Count и начальный индекс Roff в строке Right. Сравнение производится с учетом регистра символов. В качестве значения метод возвращает:

- отрицательное число — если объект класса string меньше строки, переданной в качестве параметра;
- 0 — если строки равны;

Листинги на странице http://unicross.narod.ru/cpp/
положительное число — если объект класса string больше строки, переданной в качестве параметра.

Пример сравнения строк:
```cpp
std::string str1("abc"), str2("abd");
std::cout << str1.compare("abd") << std::endl; // Прототип 1
std::cout << str1.compare(0, 2, "ab") << std::endl; // Прототип 2
std::cout << str1.compare(0, 2, "abc", 2)
    << std::endl; // Прототип 3
std::cout << str1.compare(str2) << std::endl; // Прототип 4
std::cout << str1.compare(0, 2, str2) << std::endl; // Прототип 5
std::cout << str1.compare(0, 2, str2, 0, 2)
    << std::endl; // Прототип 6
```

Итераторы

Итератор — это объект, выполняющий в контейнере роль указателя. С помощью итератора можно перемещаться внутри контейнера и получать доступ к отдельным элементам. В классе string определены следующие типы итераторов:

- `iterator` — итератор. При увеличении значения итератор перемещается к концу строки. Пример объявления переменной:
  ```cpp
  std::basic_string<char>::iterator it;
  ```

- `const_iterator` — константный итератор. Изменить значение, на которое ссыльается итератор, нельзя. Пример объявления переменной:
  ```cpp
  std::basic_string<char>::const_iterator it;
  ```

- `reverse_iterator` — обратный итератор. При увеличении значения итератор перемещается к началу строки. Пример объявления переменной:
  ```cpp
  std::basic_string<char>::reverse_iterator it;
  ```

- `const_reverse_iterator` — константный обратный итератор. Изменить значение, на которое ссыльается итератор, нельзя. Пример объявления переменной:
  ```cpp
  std::basic_string<char>::const_reverse_iterator it;
  ```

Присвоить значения переменным позволяют следующие методы:
- `begin()` — возвращает итератор, установленный на первый символ строки. Прототипы метода:

Листинги на странице http://unicross.narod.ru/cpp/
iterator begin();
const_iterator begin() const;

Выведем первый символ строки:
std::string str("String");
std::basic_string<char>::iterator it;
it = str.begin();
std::cout << *it << std::endl; // S

cbegin() — возвращает константный итератор, установленный на первый символ строки. Прототип метода:
const_iterator cbegin() const;

rbegin() — возвращает обратный итератор, установленный на последний символ строки. Прототипы метода:
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;

Выведем последний символ строки:
std::string str("String");
std::basic_string<char>::reverse_iterator it;
it = str.rbegin();
std::cout << *it << std::endl; // g

crbegin() — возвращает константный обратный итератор, установленный на последний символ строки. Прототип метода:
const_reverse_iterator crbegin() const;
end() — возвращает итератор, установленный на позицию после последнего символа строки. Прототипы метода:
iterator end();
const_iterator end() const;

Выведем последний символ строки:
std::string str("String");
std::basic_string<char>::iterator it;
it = str.end();
std::cout << *(--it) << std::endl; // g

Листинги на странице http://unicross.narod.ru/cpp/
→ cend() — возвращает константный итератор, установленный на позицию после последнего символа строки. Прототип метода:
    const_iterator cend() const;

→ rend() — возвращает обратный итератор, установленный на позицию перед первым символом строки. Прототипы метода:
    reverse_iterator rend();
    const_reverse_iterator rend() const;

Выведем первый символ строки:
    std::string str("String");
    std::basic_string<char>::reverse_iterator it;
    it = str.rend();
    std::cout << *(--it) << std::endl; // S

→ crend() — возвращает константный обратный итератор, установленный на позицию перед первым символом строки. Прототип метода:
    const_reverse_iterator crend() const;

Над итераторами можно производить такие же операции, как и с указателями. Чтобы получить или изменить значение, на которое ссылается итератор, перед названием переменной указывается оператор * (*it). Перемещение итератора осуществляется с помощью операторов ++ и --. Кроме того, итераторы можно сравнивать с помощью операторов сравнения. В качестве примера изменяем значение первого символа, а затем выведем все символы строки в прямом и обратном порядке с помощью цикла for (листинг 6.5).

<table>
<thead>
<tr>
<th>Листинг 6.5. Перебор символов в строке с помощью итераторов</th>
</tr>
</thead>
<tbody>
<tr>
<td>#include &lt;iostream&gt;</td>
</tr>
<tr>
<td>#include &lt;string&gt;</td>
</tr>
</tbody>
</table>

    int main() {
        std::string str("String");
        std::basic_string<char>::iterator it1, it2;
        std::basic_string<char>::reverse_iterator it3, it4;
        it1 = str.begin();
        *it1 = 's'; // Изменение значения
    }

Листинги на странице http://unicross.narod.ru/cpp/
// Перебор символов в прямом порядке
for (it1=str.begin(), it2=str.end(); it1!=it2; ++it1) {
    std::cout << *it1 << std::endl;
}
std::cout << "---------------------" << std::endl;
// Перебор символов в обратном порядке
for (it3=str.rbegin(), it4=str.rend(); it3!=it4; ++it3) {
    std::cout << *it3 << std::endl;
}
std::cin.get();
return 0;

Расширенные символы и строки

Помимо обычных символов, имеющих тип char, язык C++ поддерживает расширенные символы, имеющие тип wchar_t. Тип wchar_t занимает в памяти два байта. Диапазон значений от 0 до 65 535. Строки, состоящие из расширенных символов, также как и обычные C-строки, завершаются нулевым символом. Для более удобной работы с расширенными строками можно воспользоваться шаблонным классом wstring.

Для вывода расширенных символов и строк вместо объекта cout следует использовать объект wcout, а для ввода данных объект wcin. Предварительно необходимо настроить локаль. Пример вывода расширенного символа и строки:
std::setlocale(LC_ALL, "Russian_Russia.1251");
std::wcout << L'п' << std::endl;
std::wcout << L"Строка" << std::endl;

Объявление и инициализация расширенного символа

Переменной, имеющей тип wchar_t, можно присвоить числовое значение (код символа) или указать символ внутри апострофов, перед которыми добавлена буква L. Внутри апострофов можно указать специальные символы, например, \n, \t и др.
Пример объявления и инициализации расширенного символа:
wchar_t ch1 = 1087, ch2 = L'п';
std::wcout << ch1 << std::endl; // п
std::wcout << ch2 << std::endl; // п

Листинг на странице http://unicross.narod.ru/cpp/
Коды первых 128 символов совпадают с кодами символов в кодировке ASCII (табл. 6.1). Коды русских букв приведены в табл. 6.3. Обратите внимание на то, что коды букв "ё" и "ъ" выпадают из последовательности кодов.

**Таблица 6.3. Коды русских букв при использовании расширенных символов**

<table>
<thead>
<tr>
<th>буква</th>
<th>код</th>
<th>буква</th>
<th>код</th>
<th>буква</th>
<th>код</th>
<th>буква</th>
<th>код</th>
</tr>
</thead>
<tbody>
<tr>
<td>А</td>
<td>1040</td>
<td>Р</td>
<td>1056</td>
<td>а</td>
<td>1072</td>
<td>р</td>
<td>1088</td>
</tr>
<tr>
<td>Б</td>
<td>1041</td>
<td>С</td>
<td>1057</td>
<td>б</td>
<td>1073</td>
<td>с</td>
<td>1089</td>
</tr>
<tr>
<td>В</td>
<td>1042</td>
<td>Т</td>
<td>1058</td>
<td>в</td>
<td>1074</td>
<td>т</td>
<td>1090</td>
</tr>
<tr>
<td>Г</td>
<td>1043</td>
<td>У</td>
<td>1059</td>
<td>г</td>
<td>1075</td>
<td>у</td>
<td>1091</td>
</tr>
<tr>
<td>Д</td>
<td>1044</td>
<td>Ф</td>
<td>1060</td>
<td>д</td>
<td>1076</td>
<td>ф</td>
<td>1092</td>
</tr>
<tr>
<td>Е</td>
<td>1045</td>
<td>Х</td>
<td>1061</td>
<td>е</td>
<td>1077</td>
<td>х</td>
<td>1093</td>
</tr>
<tr>
<td>Ё</td>
<td>1025</td>
<td>Ц</td>
<td>1062</td>
<td>ё</td>
<td>1105</td>
<td>ц</td>
<td>1094</td>
</tr>
<tr>
<td>Ж</td>
<td>1046</td>
<td>Ч</td>
<td>1063</td>
<td>ж</td>
<td>1078</td>
<td>ч</td>
<td>1095</td>
</tr>
<tr>
<td>З</td>
<td>1047</td>
<td>Ш</td>
<td>1064</td>
<td>з</td>
<td>1079</td>
<td>ш</td>
<td>1096</td>
</tr>
<tr>
<td>И</td>
<td>1048</td>
<td>Щ</td>
<td>1065</td>
<td>и</td>
<td>1080</td>
<td>щ</td>
<td>1097</td>
</tr>
<tr>
<td>Й</td>
<td>1049</td>
<td>Ъ</td>
<td>1066</td>
<td>й</td>
<td>1081</td>
<td>й</td>
<td>1098</td>
</tr>
<tr>
<td>К</td>
<td>1050</td>
<td>Ы</td>
<td>1067</td>
<td>к</td>
<td>1082</td>
<td>ы</td>
<td>1099</td>
</tr>
<tr>
<td>Л</td>
<td>1051</td>
<td>Ь</td>
<td>1068</td>
<td>л</td>
<td>1083</td>
<td>ь</td>
<td>1100</td>
</tr>
<tr>
<td>М</td>
<td>1052</td>
<td>Э</td>
<td>1069</td>
<td>м</td>
<td>1084</td>
<td>э</td>
<td>1101</td>
</tr>
<tr>
<td>Н</td>
<td>1053</td>
<td>Ю</td>
<td>1070</td>
<td>н</td>
<td>1085</td>
<td>ю</td>
<td>1102</td>
</tr>
<tr>
<td>О</td>
<td>1054</td>
<td>Я</td>
<td>1071</td>
<td>о</td>
<td>1086</td>
<td>я</td>
<td>1103</td>
</tr>
<tr>
<td>П</td>
<td>1055</td>
<td>П</td>
<td>1087</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Функции для работы с расширенными символами**

Функции для работы с расширенными символами являются аналогами функций, предназначенных для работы с обычными символами. В названии таких функций добавляется буква "w", например, расширенная функция iswalpha() является...

Листинги на странице http://unicross.narod.ru/cpp/
аналогом функции `isalpha()`. Названия обычных функций и прототипы соответствующих им расширенных функций приведены в табл. 6.4.

<table>
<thead>
<tr>
<th>Обычная функция</th>
<th>Прототип расширенной функции</th>
</tr>
</thead>
<tbody>
<tr>
<td>isalnum()</td>
<td>int iswalnum(wint_t C);</td>
</tr>
<tr>
<td>isalpha()</td>
<td>int iswalpha(wint_t C);</td>
</tr>
<tr>
<td>iscntrl()</td>
<td>int iswcntrl(wint_t C);</td>
</tr>
<tr>
<td>isdigit()</td>
<td>int iswdigit(wint_t C);</td>
</tr>
<tr>
<td>isgraph()</td>
<td>int iswgraph(wint_t C);</td>
</tr>
<tr>
<td>islower()</td>
<td>int iswlower(wint_t C);</td>
</tr>
<tr>
<td>isprint()</td>
<td>int iswprint(wint_t C);</td>
</tr>
<tr>
<td>ispunct()</td>
<td>int iswpunct(wint_t C);</td>
</tr>
<tr>
<td>isspace()</td>
<td>int iswspace(wint_t C);</td>
</tr>
<tr>
<td>isupper()</td>
<td>int iswupper(wint_t C);</td>
</tr>
<tr>
<td>isxdigit()</td>
<td>int iswdxdigit(wint_t C);</td>
</tr>
<tr>
<td>tolower()</td>
<td>wint_t towlower(wint_t C);</td>
</tr>
<tr>
<td>toupper()</td>
<td>wint_t towupper(wint_t C);</td>
</tr>
</tbody>
</table>

Прежде чем использовать расширенные функции необходимо подключить заголовочный файл `cwctype`. Тип `wint_t` определен следующим образом:

typedef unsigned short wint_t;

В качестве примера проверим регистр символа и выведем соответствующее сообщение, а затем преобразуем символ к верхнему регистру и выведем его в окно консоли (листиング 6.6).

Листинг 6.6. Пример работы с расширенными символами

```cpp
#include <iostream>
#include <locale>
#include <cwctype>
```

Листинги на странице http://unicross.narod.ru/cpp/
int main() {
    std::setlocale(LC_ALL, "Russian_Russia.1251");
    wchar_t ch = L'н';
    if (std::iswlower(ch)) {
        std::wcout << L"Строчная" << std::endl;
    }
    else {
        std::wcout << L"Прописная" << std::endl;
    } // Результат: Строчная
    ch = std::towupper(ch);
    std::wcout << ch << std::endl; // Ч
    std::cin.get();
    return 0;
}

Для проверки типа содержимого символа вместо функций, перечисленных в табл. 6.4, можно воспользоваться функцией iswctype(). Прототип функции:

#include <cwctype>
int iswctype(wint_t C, wctype_t Type);

В первом параметре указывается расширенный символ, а во втором параметре результат выполнения функции wctype(). Прототип функции:

#include <cwctype>
wctype_t wctype(const char *Attr);

Функция wctype() принимает в качестве параметра С-строку, содержащую название проверки: "alnum", "alpha", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper" или "xdigit".

Для изменения регистра символа можно воспользоваться функцией towctrans(). Прототип функции:

#include <cwctype>
wint_t towctrans(wint_t C, wctrans_t Type);

В первом параметре указывается расширенный символ, а во втором параметре результат выполнения функции wctrans(). Прототип функции:

#include <cwctype>
wctrans_t wctrans(const char *Attr);

Листинги на странице http://unicross.narod.ru/cpp/
Функция wctrans() принимает в качестве параметра C-строку, содержащую название трансформации: "tolower" (преобразование в нижний регистр) или "toupper" (преобразование в верхний регистр). В качестве примера переделаем предыдущий пример (листинг 6.6) и используем функции iswctype() и towctrans() вместо функций iswlower() и toupper() (листинг 6.7).

Листинг 6.7. Пример использования функций iswctype() и towctrans()

```
#include <iostream>
#include <locale>
#include <cwctype>

int main() {
    std::setlocale(LC_ALL, "Russian_Russia.1251");
    wchar_t ch = L'н';
    if (std::iswctype(ch, std::wctype("lower"))) {
        std::wcout << L"Строчная" << std::endl;
    } else {
        std::wcout << L"Прописная" << std::endl;
    } // Результат: Строчная
    ch = std::towctrans(ch, std::wctrans("toupper"));
    std::wcout << ch << std::endl; // П
    std::cin.get();
    return 0;
}
```

Преобразование расширенных символов в обычные и наоборот

Для преобразования расширенных символов (тип wchar_t) в обычные (тип char) и наоборот предназначены две функции:

→ btowc() — преобразует обычный символ в расширенный. В случае ошибки функция возвращает значение макроопределения WEOF (65 535). Прототип функции:
```
#include <cwchar>

wint_t btowc(int Ch);
```

Листинги на странице http://unicross.narod.ru/cpp/
\textbf{wctob()} — преобразует расширенный символ в обычный. В случае ошибки функция возвращает значение \texttt{-1} и устанавливает значение макроопределения \texttt{errno} соответствующим значением \texttt{ENILSEQ} (равно 42). Прототип функции:

```c
#include <cwchar>

int wctob(wint_t wch);
```

Прежде чем производить преобразования необходимо настроить локаль. Пример преобразования приведен в листинге 6.8.

**Листинг 6.8. Преобразование символов**

```c
#include <iostream>
#include <clocale>
#include <cwchar>

int main() {
    std::setlocale(LC_ALL, "Russian_Russia.1251");
    wchar_t wch = 0;
    char ch = 'п';
    // Преобразование обычного символа в расширенный
    wch = std::towc(ch);
    if (wch == WEOF) {
        std::cout << "Ошибка" << std::endl;
    }
    else {
        std::wcout << wch << std::endl; // п
    }
    // Преобразование расширенного символа в обычный
    ch = std::wctob(wch);
    if (ch == -1) {
        std::cout << "Ошибка" << std::endl;
    }
    else {
        std::cout << ch << std::endl; // п
    }
    std::cin.get();
    return 0;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
Функции для работы с расширенными строками

Расширенные строки, являются массивами, содержащими символы типа wchar_t. Последним символом расширенной строки является нулевой символ. Объявляется расширенная строка также как и массив элементов типа wchar_t:

wchar_t str[7];

При инициализации расширенной строки можно перечислить расширенные символы внутри фигурных скобок или указать строку внутри двойных кавычек, перед которыми добавлена буква L:

wchar_t str1[7] = {L'S', L't', L'r', L'i', L'n', L'g', L'0'};
wchar_t str2[] = L"String";

Функции для работы с расширенными строками являются аналогами функций, предназначенных для работы с обычными строками. В названии таких функций фрагмент "str" заменяется комбинацией символов "wcs", например, расширенная функция wcslen() является аналогом функции strlen(). Прежде чем использовать расширенные функции необходимо подключить заголовочный файл wchar. Названия обычных функций и прототипы соответствующих им расширенных функций приведены в табл. 6.5.

Таблица 6.5. Функции для работы с расширенными строками

<table>
<thead>
<tr>
<th>Обычная функция</th>
<th>Прототип расширенной функции</th>
</tr>
</thead>
<tbody>
<tr>
<td>strcat()</td>
<td>wchar_t *wcscat(wchar_t *Dest, const wchar_t *Source);</td>
</tr>
<tr>
<td>strcat_s()</td>
<td>erro_t wcsat_s(wchar_t *Dst, rsize_t SizeInWords, const wchar_t *Src);</td>
</tr>
<tr>
<td>strcpy()</td>
<td>wchar_t *wcsncpy(wchar_t *Dest, const wchar_t *Source);</td>
</tr>
<tr>
<td>strcpy_s()</td>
<td>erro_t wcsncpy_s(wchar_t *Dst, rsize_t SizeInWords, const wchar_t *Src);</td>
</tr>
<tr>
<td>strlen()</td>
<td>size_t wcslen(const wchar_t *Str);</td>
</tr>
<tr>
<td>strncat()</td>
<td>wchar_t *wcsncat(wchar_t *Dest, const wchar_t *Source, size_t Count);</td>
</tr>
</tbody>
</table>

Листинги на странице http://unicross.narod.ru/cpp/
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strncat_s()</code></td>
<td><code>errno_t wscncat_s(wchar_t *Dst, rsize_t SizeInWords, const wchar_t *Src, rsize_t MaxCount);</code></td>
</tr>
<tr>
<td><code>strncpy()</code></td>
<td><code>wchar_t *wcsncpy(wchar_t *Dest, const wchar_t *Source, size_t Count);</code></td>
</tr>
<tr>
<td><code>strncpy_s()</code></td>
<td><code>errno_t wcsncpy_s(wchar_t *Dst, rsize_t SizeInWords, const wchar_t *Src, rsize_t MaxCount);</code></td>
</tr>
<tr>
<td><code>strtok()</code></td>
<td><code>wchar_t *wcstok(wchar_t *Str, const wchar_t *Delim);</code></td>
</tr>
<tr>
<td><code>strtok_s()</code></td>
<td><code>wchar_t *wcstok_s(wchar_t *Str, const wchar_t *Delim, wchar_t **Context);</code></td>
</tr>
</tbody>
</table>

**Поиск и замена в строке**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strchr()</code></td>
<td><code>wchar_t *wcschr(wchar_t *Str, wchar_t Ch);</code></td>
</tr>
<tr>
<td><code>strcspn()</code></td>
<td><code>size_t wcscspn(const wchar_t *Str, const wchar_t *Control);</code></td>
</tr>
<tr>
<td><code>strpbrk()</code></td>
<td><code>wchar_t *wcpbrk(wchar_t *Str, const wchar_t *Control);</code></td>
</tr>
<tr>
<td><code>strrchr()</code></td>
<td><code>wchar_t *wcsrchr(wchar_t *Str, wchar_t Ch);</code></td>
</tr>
<tr>
<td><code>strspn()</code></td>
<td><code>size_t wcspn(const wchar_t *Str, const wchar_t *Control);</code></td>
</tr>
<tr>
<td><code>strstr()</code></td>
<td><code>wchar_t *wcsstr(const wchar_t *Str, const wchar_t *SubStr);</code></td>
</tr>
</tbody>
</table>

**Сравнение строк**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strcmp()</code></td>
<td><code>int wcscmp(const wchar_t *Str1, const wchar_t *Str2);</code></td>
</tr>
<tr>
<td><code>strcoll()</code></td>
<td><code>int wcscoll(const wchar_t *Str1, const wchar_t *Str2);</code></td>
</tr>
<tr>
<td><code>strncmp()</code></td>
<td><code>int wcsncmp(const wchar_t *Str1, const wchar_t *Str2, size_t MaxCount);</code></td>
</tr>
<tr>
<td><code>strxfrm()</code></td>
<td><code>size_t wcstxfrm(wchar_t *Dst, const wchar_t *Src, size_t MaxCount);</code></td>
</tr>
</tbody>
</table>

Листинги на странице http://unicross.narod.ru/cpp/
## Работа с массивами расширенных символов

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>memchr()</td>
<td>wchar_t *wmemchr(const wchar_t *S, wchar_t c, size_t N);</td>
</tr>
<tr>
<td>memcmp()</td>
<td>int wmempcmp(const wchar_t *S1, const wchar_t *S2, size_t N);</td>
</tr>
<tr>
<td>memcpy()</td>
<td>wchar_t *wmemcpy(const wchar_t *S1, const wchar_t *S2, size_t N);</td>
</tr>
<tr>
<td>memcpy_s()</td>
<td>errno_t wmempcy_s(const wchar_t *S1, rsize_t N1, const wchar_t *S2, rsize_t N);</td>
</tr>
<tr>
<td>memmove()</td>
<td>wchar_t *wmemmove(const wchar_t *S1, const wchar_t *S2, size_t N);</td>
</tr>
<tr>
<td>memmove_s()</td>
<td>errno_t wmempmove_s(const wchar_t *S1, rsize_t N1, const wchar_t *S2, rsize_t N);</td>
</tr>
<tr>
<td>memset()</td>
<td>wchar_t *wmemset(const wchar_t *S, wchar_t c, size_t N);</td>
</tr>
</tbody>
</table>

### Преобразование строки в число

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>strtod()</td>
<td>double wcstod(const wchar_t *Str, wchar_t **EndPtr);</td>
</tr>
<tr>
<td>strtol()</td>
<td>long wcstol(const wchar_t *Str, wchar_t **EndPtr, int Radix);</td>
</tr>
<tr>
<td>strtoul()</td>
<td>unsigned long wcstoul(const wchar_t *Str, wchar_t **EndPtr, int Radix);</td>
</tr>
</tbody>
</table>

Основные операции с расширенными строками показаны в листинге 6.9.

### Листинг 6.9. Основные операции с расширенными строками

```cpp
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES 1
#include <iostream>
#include <locale>
#include <cwchar>

int main() {
    std::locale(LC_ALL, "Russian_Russia.1251");
    wchar_t str1[] = L"Строка", str2[100] = {0};
    std::wcout << std::wcslen(str1) << std::endl; // Длина
    std::wcscpy(str2, L"Один");                // Копирование
    std::wcscat(str2, L"Два");                 // Конкатенация
    std::wcout << str2 << std::endl;            // Вывод
}
```

Класс wstring

Класс wstring предоставляет удобный интерфейс для работы со строками, состоящими из расширенных символов. Название класса wstring является лишь псевдонимом шаблонного класса basic_string:

typedef basic_string<wchar_t, char_traits<wchar_t>, allocator<wchar_t>> wstring;

Для использования класса wstring необходимо подключить файл string:

#include <string>

Классы string и wstring являются псевдонимами шаблонного класса basic_string. Поэтому с экземплярами класса wstring можно производить такие же операции, как и с экземплярами класса string, а также использовать те же самые методы. Различие заключается в типах данных. Тип char во всех прототипах методов заменяется типом wchar_t. Так как других различий практически нет мы не будем повторно рассматривать эти операции и методы. За подробной информацией обращайтесь к описанию класса string. Основные операции с экземплярами класса wstring показаны в листинге 6.10.

Листинг 6.10. Основные операции с экземплярами класса wstring

#include <iostream>
#include <locale>
#include <string>

int main() {
    std::setlocale(LC_ALL, "Russian_Russia.1251");
    std::wstring str1(L"String1"), str2(L"String2"), str3;
    str3 = str1 + str2; // Конкатенация
    std::wcout << str3 << std::endl; // String1String2
    str1[0] = L's'; // Доступ по индексу
    std::wcout << str1 << std::endl; // string1
    std::wcout << str1.size() << std::endl; // Длина
    str2 = str1.substr(2, 3); // Получение фрагмента
}

Листинги на странице http://unicross.narod.ru/cpp/
std::wcout << str2 << std::endl;       // rin
str1.replace(0, 4, L"****");        // Замена
std::wcout << str1 << std::endl;      // ****ng1
std::wcin.get();
return 0;
}
Глава 7.
Работа с датой и временем

© Прохоренок Н.А., 2010 г., unicross@mail.ru

Этот DjVu-файл предоставляется КАК ЕСТЬ. Автор не несет никакой ответственности за прямые или косвенные проблемы, связанные с использованием данного файла. ВЫ ИСПОЛЬЗУЕТЕ ЕГО НА СВОЙ СТРАХ И РИСК.

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Запрещено:
- перепечатывать всю книгу или отдельные главы без письменного разрешения Автора;
- декомпилировать данный файл и преобразовывать его в любой другой формат.
Основные функции для работы с датой и временем в языке C++ объявлены в заголовочном файле `<ctime>` (файл `time.h` в языке C). В этом файле определены также следующие типы данных:

- `size_t` — используется в функции `strftime()`. Размер зависит от настроек. По умолчанию 4 байта. Определение типа:
  ```c
  typedef unsigned int size_t;
  typedef unsigned __int64 size_t;
  ```

- `clock_t` — возвращается функцией `clock()`. Определение типа:
  ```c
  typedef long clock_t;
  ```

- `time_t` — используется для представления времени в виде целочисленного значения. Размер типа зависит от настроек. По умолчанию занимает 8 байт. Определение типа:
  ```c
  typedef __time32_t time_t;
  typedef __time64_t time_t;
  ```

Определение типов `__time32_t` и `__time64_t` выглядит так:

  ```c
  typedef long __time32_t;
  typedef __int64 __time64_t;
  ```

Некоторые функции в качестве значения возвращают указатель на структуру `tm`. Определение структуры `tm` выглядит следующим образом:

```c
struct tm {
    int tm_sec; // Секунды (число от 0 до 59, изредка до 61)
    int tm_min; // Минуты (число от 0 до 59)
    int tm_hour; // Час (число от 0 до 23)
    int tm_mday; // День месяца (число от 1 до 31)
    int tm_mon; // Месяц (число от 0 (январь) до 11 (декабрь))
    int tm_year; // Год, начиная с 1900 года
    int tm_wday; // День недели (число от 0 до 6 (0 - это воскресенье,
                  // 1 - это понедельник, ..., 6 - это суббота))
    int tm_yday; // День в году (число от 0 до 365)
    int tm_isdst; // Если больше 0, значит действует летнее время
                   // Если 0, значит летнее время не установлено
                   // Если меньше 0, то информации нет
};
```

Получение текущей даты и времени

Получить текущую дату и время позволяют следующие функции:

\[ \text{time()} \] — возвращает количество секунд, прошедших с начала эпохи (с 1 января 1970 г.). Прототип функции:

```c
#include <ctime>

time_t time(time_t *Time);
```

Функцию можно вызвать двумя способами, передавая в качестве параметра нулевой указатель или адрес переменной, в которую будет записано возвращаемое значение.

Пример:

```c
std::time_t t1 = std::time(0); // Передаем нулевой указатель
std::cout << t1 << std::endl; // 1285559288
std::time_t t2;
std::time(&t2); // Передаем адрес переменной
std::cout << t2 << std::endl; // 1285559288
```

В VC++ вместо функции `time()` можно использовать функции `__time32()` и `__time64()`. Число перед открывающей круглой скобкой означает количество бит.

Прототипы функций:

```c
__time32_t __time32(__time32_t *Time);
__time64_t __time64(__time64_t *Time);
```

Пример использования:

```c
__time32_t t1;
__time32(&t1);
std::cout << t1 << std::endl; // 1285559288

__time64_t t2;
__time64(&t2);
std::cout << t2 << std::endl; // 1285559288
```

\[ \text{gmtime()} \] — возвращает указатель на структуру `tm` с универсальным временем (UTC) или нулевой указатель в случае ошибки. В качестве параметра указывается адрес переменной, которая содержит количество секунд, прошедших с начала эпохи.

Чтобы получить текущую дату в качестве параметра следует передать результат выполнения функции `time()`. Прототип функции:

Листинги на странице http://unicross.narod.ru/cpp/
#include <ctime>

struct tm *gmtime(const time_t *Time);

Пример использования функции:
std::tm *ptm = 0;
std::time_t t = std::time(0);
ptm = std::gmtime(&t);
if (!ptm) {
    std::cout << "Error" << std::endl;
    std::exit(1);
}
std::cout << ptm->tm_sec << std::endl; // 56
std::cout << ptm->tm_min << std::endl; // 54
std::cout << ptm->tm_hour << std::endl; // 23
std::cout << ptm->tm_mday << std::endl; // 27
std::cout << ptm->tm_mon << std::endl; // 8 (сентябрь)
std::cout << ptm->tm_year << std::endl; // 110 (2010 г.)
std::cout << ptm->tm_wday << std::endl; // 1 (понедельник)
std::cout << ptm->tm_yday << std::endl; // 269
std::cout << ptm->tm_isdst << std::endl; // 0

В VC++ вместо функции gmtime() можно использовать функции _gmtime32() и _gmtime64(). Число перед открывающей круглой скобкой означает количество бит.
Прототипы функций:
struct tm *__gmtime32(const __time32_t *Time);
struct tm *__gmtime64(const __time64_t *Time);

В VC++ при использовании функции gmtime() (а также при использовании функций _gmtime32() и _gmtime64()) выводится предупреждающее сообщение ("warning C4996"), поэтому вместо функции gmtime() следует использовать функцию gmtime_s() (_gmtime32_s() и _gmtime64_s() вместо _gmtime32() и _gmtime64() соответственно). Прототипы функций:
#include <ctime>
erro_t gmtime_s(struct tm *Tm, const time_t *Time);

Листинги на странице http://unicross.narod.ru/cpp/
errno_t _gmtime32_s(struct tm *Tm, const __time32_t *Time);  
errno_t _gmtime64_s(struct tm *Tm, const __time64_t *Time);  
Если ошибок нет, то функции возвращают значение 0. При наличии ошибки возвращается значение макроса EINV (значение равно 22) и переменная errno устанавливается равной EINV. Пример использования функции gmtime_s():  
std::tm ptm;  
std::time_t t = std::time(0);  
errno_t err = gmtime_s(&ptm, &t);  
if (err) {  
    std::cout << "Error" << std::endl;  
    std::exit(1);  
}  
std::cout << ptm.tm_sec << std::endl; // 56  
std::cout << ptm.tm_min << std::endl; // 54  
std::cout << ptm.tm_hour << std::endl; // 23  
std::cout << ptm.tm_mday << std::endl; // 27  
std::cout << ptm.tm_mon << std::endl; // 8 (сентябрь)  
std::cout << ptm.tm_year << std::endl; // 110 (2010 г.)  
std::cout << ptm.tm_wday << std::endl; // 1 (понедельник)  
std::cout << ptm.tm_yday << std::endl; // 269  
std::cout << ptm.tm_isdst << std::endl; // 0  

locale () — возвращает указатель на структуру tm с локальным временем или нулевой указатель в случае ошибки. В качестве параметра указывается адрес переменной, которая содержит количество секунд, прошедших с начала эпохи. Чтобы получить текущую дату в качестве параметра следует передать результат выполнения функции time(). Прототип функции:  
#include <ctime>  
struct tm *localtime(const time_t *Time);  
Пример использования функции:  
std::tm *ptm = 0;  
std::time_t t = std::time(0);  

Листинги на странице http://unicross.narod.ru/cpp/
ptm = std::localtime(&t);
if (!ptm) {
    std::cout << "Error" << std::endl;
    std::exit(1);
}
std::cout << ptm->tm_sec << std::endl;  // 56
std::cout << ptm->tm_min << std::endl;  // 54
std::cout << ptm->tm_hour << std::endl;  // 4
std::cout << ptm->tm_mday << std::endl;  // 28
std::cout << ptm->tm_mon << std::endl;  // 8 (сентябрь)
std::cout << ptm->tm_year << std::endl;  // 110 (2010 г.)
std::cout << ptm->tm_wday << std::endl;  // 2 (вторник)
std::cout << ptm->tm_yday << std::endl;  // 270
std::cout << ptm->tm_isdst << std::endl;  // 1 (летнее время)

В VC++ вместо функции localtime() можно использовать функции __localtime32() и __localtime64(). Число перед открывающей круглой скобкой означает количество бит. Прототипы функций:

```c
struct tm *__localtime32(const __time32_t *Time);
struct tm *__localtime64(const __time64_t *Time);
```

В VC++ при использовании функции localtime() (а также при использовании функций __localtime32() и __localtime64()) выводится предупреждающее сообщение ("warning C4996"), поэтому вместо функции localtime() следует использовать функцию localtime_s() (__localtime32_s() и __localtime64_s() вместо localtime32() и localtime64() соответственно). Прототипы функций:

```c
#include <ctime>

errno_t localtime_s(struct tm *Tm, const time_t *Time);
errno_t __localtime32_s(struct tm *Tm, const __time32_t *Time);
errno_t __localtime64_s(struct tm *Tm, const __time64_t *Time);
```

Если ошибок нет, то функции возвращают значение 0. При наличии ошибки возвращается значение макроса EINVAL (значение равно 22) и переменная errno устанавливается равной EINVAL. Пример использования функции localtime_s():

```c
std::tm ptm;
```
std::time_t t = std::time(0);
errno_t err = localtime_s(&ptm, &t);
if (err) {
    std::cout << "Error" << std::endl;
    std::exit(1);
}
std::cout << ptm.tm_mday << std::endl; // 28
std::cout << ptm.tm_mon << std::endl; // 8 (сентябрь)
std::cout << ptm.tm_year << std::endl; // 110 (2010 г.)

mtime() — возвращает количество секунд, прошедших с начала эпохи. В качестве параметра передается указатель на структуру tm. В случае ошибки возвращается значение -1. Прототип функции:

#include <ctime>
time_t mtime(struct tm *Tm);

Пример использования функции:

std::tm ptm;
std::time_t t1 = std::time(0), t2 = 0;
errno_t err = localtime_s(&ptm, &t1);
if (err) {
    std::cout << "Error" << std::endl;
    std::exit(1);
}
t2 = std::mtime(&ptm);
std::cout << t1 << std::endl; // 1285636288
std::cout << t2 << std::endl; // 1285636288

В VC++ вместо функции mtime() можно использовать функции _mtime32() и _mtime64(). Число перед открывающей круглой скобкой означает количество бит. Прототипы функций:

__time32_t _mtime32(struct tm *Tm);
__time64_t _mtime64(struct tm *Tm);

Листинги на странице http://unicross.narod.ru/cpp/
```cpp
difftime() — возвращает разность между двумя датами (\(Time_1 - Time_2\)). В случае ошибки возвращается значение 0 и переменная \(errno\) устанавливается равной \(EINVAL\). Прототип функции:
#include <ctime>

double difftime(time_t Time1, time_t Time2);

Пример:
std::time_t t1 = std::time(0), t2 = 1233368623;
double result = std::difftime(t1, t2);
std::cout << result << std::endl;

В VC++ вместо функции difftime() можно использовать функции _difftime32() и _difftime64(). Число перед открывающей круглой скобкой означает количество бит. Прототипы функций:
double _difftime32(_time32_t Time1, _time32_t Time2);
double _difftime64(_time64_t Time1, _time64_t Time2);

Выведем текущую дату и время таким образом, чтобы день недели и месяц были написаны по-русски (листинг 7.1).

| Листинг 7.1. Вывод текущей даты и времени |

```cpp
#include <iostream>
#include <locale>
#include <ctime>
#include <iomanip> // Для setfill() и setw()

int main() {
    std::setlocale(LC_ALL, "Russian_Russia.1251");
    std::tm ptm;
    std::time_t t = std::time(0);
    char d[](25) = {"воскресенье", "понедельник", "вторник", "среда", "четверг", "пятница", "суббота"};
    errno_t err = localtime_s(&ptm, &t); // Получаем текущее время
    std::cout << std::setw(3) << std::setfill('0') << (ptm.tm_mon + 1) << '.' << (ptm.tm_mycday) << '.' << ptm.tm_year + 1900;
    std::cout << " " << m[ptm.tm_mon] << " " << d[ptm.tm_wday] << std::endl;
    return 0;
}
```
Листинги на странице http://unicross.narod.ru/cpp/
if (err) {
    std::cout << "Error" << std::endl;
    std::exit(1);
}
std::cout << "Сегодня:" << std::endl
    << d[ptm.tm_wday] " " ptm.tm_mday " "
    << m[ptm.tm_mon] " " (ptm.tm_year + 1900);
std::cout << std::setfill('0') " " ptm.tm_hour "::" << std::setw(2)
    << ptm.tm_min ":" << std::setw(2) << ptm.tm_sec << std::endl
    << std::setw(2) << ptm.tm_mday
    << "." << std::setw(2) << (ptm.tm_mon + 1) " "
    << (ptm.tm_year + 1900) << std::endl;
std::cin.get();
return 0;
}
Результат выполнения:
Сегодня:
вторник 28 сентября 2010 16:01:05
28.09.2010
В этом примере мы использовали манипуляторы setfill() и setw(), объявленные в заголовочном файле iomanip. Манипулятор setfill() предназначен для указания символа-заполнителя, а манипулятор setw() — для указания ширины поля. Если эти манипуляторы не использовать, то время "16:01:05" будет выведено так: "16:1:5".

Форматирование даты и времени

Получить форматированный вывод даты и времени позволяют следующие функции:

→ asctime() — возвращает указатель на C-строку специального формата или нулевой указатель в случае ошибки. В конец строки вставляется символ перевода строки (\n) и нулевой символ (\0). Прототип функции:
#include <ctime>
char *asctime(const struct tm *Tm);
Пример использования функции:
std::tm ptm;

Листинги на странице http://unicross.narod.ru/cpp/
char *p = 0;
std::time_t t = std::time(0);
errno_t err = localtime_s(&ptm, &t);
if (err) {
    std::cout << "Error" << std::endl;
    std::exit(1);
}
p = std::asctime(&ptm);
if (!p) {
    std::cout << "Error" << std::endl;
    std::exit(1);
}
std::cout << p << std::endl;
// Tue Sep 28 17:08:56 2010
В VC++ при использовании функции asctime() выводится предупреждающее сообщение ("warning C4996"), поэтому вместо функции asctime() следует использовать функцию localtime_s(). Прототип функции:
#include <ctime>
erroo_t asctime_s(char *Buf, size_t SizeInBytes,
    const struct tm *Tm);
В первом параметре передается указатель на строку, во втором параметре — максимальный размер строки, а в третьем параметре — указатель на структуру tm. Если ошибок нет, то функция возвращают значение 0. При наличии ошибки возвращается значение макроса EINVAL (значение равно 22). Пример использования функции:
std::tm ptm;
const short SIZE = 80;
char str[SIZE] = {0};
std::time_t t = std::time(0);
erroo_t err = localtime_s(&ptm, &t);
if (err) {
    std::cout << "Error" << std::endl;

    Листинги на странице http://unicross.narod.ru/cpp/
std::exit(1);

} err = asctime_s(str, SIZE, &ptm);
if (err) {
    std::cout << "Error" << std::endl;
    std::exit(1);
}
std::cout << str << std::endl;
// Tue Sep 28 17:20:32 2010

ctime() — функция аналогична asctime(), но в качестве параметра принимает
количество секунд, прошедших с начала эпохи. Прототип функции:
#include <ctime>
char *ctime(const time_t *Time);

Пример использования функции:
char *p = 0;
std::time_t t = std::time(0);
p = std::ctime(&t);
if (!p) {
    std::cout << "Error" << std::endl;
    std::exit(1);
}
std::cout << p << std::endl;
// Tue Sep 28 17:34:13 2010

В VC++ при использовании функции ctime() выводится предупреждающее
сообщение ("warning C4996"), поэтому вместо функции ctime() следует
использовать функцию ctime_s(). Прототип функции:
#include <ctime>
errno_t ctime_s(char *Buffer, size_t SizeInBytes,
    const time_t *Time);

Если ошибок нет, то функция возвращает значение 0. При наличии ошибки
возвращается значение макроса EINVAL. Пример использования функции:

Листинги на странице http://unicross.narod.ru/cpp/
const short SIZE = 80;
char str[SIZE] = {0};
std::time_t t = std::time(0);
errno_t err = ctime_s(str, SIZE, &t);
if (err) {
    std::cout << "Error" << std::endl;
    std::exit(1);
}
std::cout << str << std::endl;
// Tue Sep 28 17:40:13 2010

- strftime() — записывает строковое представление даты в соответствии со строкой формата в C-строку buf. Прототип функции:

```
#include <ctime>

size_t strftime(char *Buf, size_t SizeInBytes, 
                const char *Format, const struct tm *Tm);
```

В первом параметре передается указатель на символьный массив, в который будет записан результат выполнения функции. В параметре SizeInBytes задается максимальный размер символьного массива. В параметре Format указывается строка специального формата, а в последнем параметре передается указатель на структуру tm с представлением даты. Функция возвращает количество записанных символов. В случае ошибки возвращается значение 0 и переменная errno устанавливается равной EINVAL.

- wcftime() — функция аналогична функции strftime(), но записывает строковое представление даты не в C-строку, а в расширенную строку buf. Прототип функции:

```
#include <ctime>

size_t wcftime(wchar_t *Buf, size_t SizeInWords, 
                const wchar_t *Format, const struct tm *Tm);
```

В параметре Format в функциях strftime() и wcftime() помимо обычных символов могут быть указаны следующие комбинации специальных символов:

Листинги на странице http://unicross.narod.ru/cpp/
8y — год из двух цифр (от "00" до "99");
8Y — год из четырех цифр (например, "2010");
8m — номер месяца с предваряющим нулем (от "01" до "12");
8b — аббревиатура месяца в зависимости от настроек локали (например, "сен" в локали "Russian_Russia.1251" или "Sep" в локали "C" для сентября);
8v — название месяца в зависимости от настроек локали (например, "Сентябрь" или "September");
8d — номер дня в месяце с предваряющим нулем (от "01" до "31");
8j — день с начала года (от "001" до "366");
8u — номер недели в году (от "00" до "53"). Неделя начинается с воскресенья;
8w — номер недели в году (от "00" до "53"). Неделя начинается с понедельника;
8z — номер дня недели ("0" — для воскресенья, "6" — для субботы);
8a — аббревиатура дня недели в зависимости от настроек локали (например, "Ср" или "Wed" для среды);
8A — название дня недели в зависимости от настроек локали (например, "среда" или "Wednesday");
8n — часы в 24-часовом формате (от "00" до "23");
8i — часы в 12-часовом формате (от "01" до "12");
8m — минуты (от "00" до "59");
8s — секунды (от "00" до "59", изредка до "61");
8p — эквивалент значениям AM и PM в текущей локали;
8c — представление даты и времени в текущей локали (например, "29.09.2010 14:37:02" или "09/29/10 14:38:05");
8%c — расширенное представление даты и времени в текущей локали (например, "29 Сентябрь 2010 г. 14:46:08" или "Wednesday, September 29, 2010 14:46:36");
8x — представление даты в текущей локали (например, "29.09.2010" или "09/29/10");
→ %x — расширенное представление даты в текущей локали (например, "29 Сентябрь 2010 г." или "Wednesday, September 29, 2010")

→ %x — представление времени в текущей локали (например, "14:41:24");

→ %Z — название часового пояса или пустая строка (например, "Московское время (лето)");

→ %8 — символ "8".

Если после символа % указан символ # в комбинациях %#d, %#H, %#I, %#j, %#m, %#M, %#s, %#u, %#w, %#W, %#y и %#Y, то предваряющие нули выводиться не будут. В качестве примера использования функции strftime() выведем текущую дату и время (листинг 7.2).

### Листинг 7.2. Форматирование даты и времени

```cpp
#include <iostream>
#include <locale>
#include <ctime>

int main() {
    std::setlocale(LC_ALL, "Russian_Russia.1251");
    std::tm ptm;
    const short SIZE = 100;
    char str[SIZE] = {0};
    std::time_t t = std::time(0);
    int err = localtime_s(&ptm, &t);
    if (err) {
        std::cout << "Error" << std::endl;
        std::exit(1);
    }
    err = std::strftime(str, SIZE,
                        "Сегодня: %d %b %Y %H:%M:%S\n%d.%m.%Y", &ptm);
    if (!err) {
        std::cout << "Error" << std::endl;
        std::exit(1);
    }
    std::cout << str << std::endl;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
std::cin.get();
return 0;
}

Результат выполнения:
Сегодня:
среда 29 сен 2010 14:57:11
29.09.2010

"Засыпание" программы

WinAPI-функция sleep() позволяет прерывать выполнение программы на указанное время. По истечении срока программа продолжит работу. Прототип функции:
#include <windows.h>
void Sleep(DWORD dwMilliseconds);

В параметре dwMilliseconds указывается количество миллисекунд, на которое прерывается выполнение программы. Тип данных DWORD определен так:
typedef unsigned long DWORD;

Для примера выведем числа от 1 до 10 (листинг 7.3). Между выводом чисел "заснем" на одну секунду.

Листинг 7.3. "Засыпание" программы

#include <iostream>
#include <windows.h>

int main() {
    for (int i=1; i<=10; ++i) {
        std::cout << "\r... " << i << "%";
        Sleep(1000); // "Засыпаем" на 1 секунду
    }
    std::cout << "\rEnd     " << std::endl;
    std::cin.get();
    return 0;
}
Измерение времени выполнения фрагментов кода

В некоторых случаях необходимо измерить время выполнения фрагментов кода, например, с целью оптимизации программы. Измерить время выполнения позволяет функция clock(). Прототип функции:

```c
#include <ctime>
clock_t clock(void);
```

Функция возвращает приблизительное время выполнения программы до вызова функции. Если время получить не получилось, функция возвращает значение -1. Чтобы измерить время выполнения фрагмента следует вызвать функцию перед фрагментом кода и сохранить результат. Затем вызвать функцию после фрагмента и вычислить разность между двумя значениями. Чтобы получить значение в секундах необходимо разделить результат на значение макрока CLOCKS_PER_SEC. Определение макрока выглядит следующим образом:

```c
#define CLOCKS_PER_SEC 1000
```

Для примера имитируем фрагмент кода с помощью WinAPI-функции Sleep() и произведем измерение времени выполнения (листинг 7.4).

**Листинг 7.4. Измерение времени выполнения**

```c
#include <iostream>
#include <ctime>
#include <windows.h>

int main() {
    std::clock_t t1, t2, t3;
    t1 = std::clock(); // Метка 1
    std::cout << "t1 = " << t1 << std::endl;
    Sleep(2000); // Имитация фрагмента кода
    t2 = std::clock(); // Метка 2
    std::cout << "t2 = " << t2 << std::endl;
    t3 = t2 - t1; // Разница между метками
    std::cout << "t3 = " << t3 << std::endl;
    std::cout << (t3 / CLOCKS_PER_SEC) << " sec." << std::endl;
    std::cin.get();
}
```

Листинги на странице http://unicross.narod.ru/cpp/
return 0;
}
Программирование на C++ в Visual Studio® 2010 Express

Глава 8.
Пользовательские функции

© Прохоренок Н.А., 2010 г., unicross@mail.ru

Этот DjVu-файл предоставляется КАК ЕСТЬ. Автор не несет никакой ответственности за прямые или косвенные проблемы, связанные с использованием данного файла. ВЫ ИСПОЛЬЗУЕТЕ ЕГО НА СВОЙ СТРАХ И РИСК.

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Запрещено:
- перепечатывать всю книгу или отдельные главы без письменного разрешения Автора;
- декомпилировать данный файл и преобразовывать его в любой другой формат.
Функция — это фрагмент кода, который можно неоднократно вызывать из любого места программы. В предыдущих главах мы уже не один раз использовали встроенные функции, входящие в состав стандартной библиотеки. Например, с помощью функции strlen() получали количество символов в C-строке. В этой главе мы рассмотрим создание пользовательских функций, которые позволяют уменьшить избыточность программного кода и повысить его структурированность.

Создание функций и ее вызов

Описание функции состоит из двух частей: объявления и определения. Объявление функции (называемое также прототипом функции) содержит информацию о типе. Используя эту информацию компилятор может найти несоответствие типа и количества параметров. Формат прототипа функции:

```
<Tип результата> <Название функции> ( [<Тип> [ <Название параметра1> ]
  [, ..., <Тип> [ <Название параметраN> ] ] ) ;
```

Параметр <Тип результата> задает тип значения, которое возвращает функция с помощью оператора return. Если функция не возвращает никакого значения, то вместо типа указывается ключевое слово void. Название функции должно быть допустимым идентификатором, к которому предъявляются такие же требования как и к названиям переменных. После названия функции, внутри круглых скобок, указывается тип и название параметров через запятую. Названия параметров в прототипе функции можно не задавать вообще, так как компилятор интересует только тип данных и количество параметров. Если функция не принимает параметров, то указываются только круглые скобки или внутри круглых скобок задается ключевое слово void (в языке C ключевое слово void является обязательным). После объявления функции должна стоять точка с запятой. Пример объявления функций:

```
int sum(int x, int y); // или int sum(int, int);
void print(const char *str); // или void print(const char *);
void print_ok(); // или void print_ok(void);
```

Определение функции содержит описание типа и названия параметров, а также реализацию. Определение функции имеет следующий формат:

```
<Tип результата> <Название функции> ( [<Тип> <Название параметра1> ]
  [, ..., <Тип> <Название параметраN> ] )
{
  <Тело функции>
```

Листинги на странице http://unicross.narod.ru/cpp/
[return[ <Возвращаемое значение>];]
}

В отличие от прототипа в определении функции после типа обязательно должно быть указано название параметра, которое является локальной переменной. Эта переменная создается при вызове функции, а после выхода из функции она удаляется. Таким образом, локальная переменная видна только внутри функции. Если название локальной переменной совпадает с названием глобальной переменной, то все операции будут производиться с локальной переменной, а значение глобальной не изменится. Чтобы в этом случае обратиться к глобальной переменной, необходимо перед названием указать оператор ::. Пример:

```cpp
int sum(int x, int y) {
    int z = x + y; // Обращение к локальной переменной x
    z = ::x + y;  // Обращение к глобальной переменной x
    return z;
}
```

После описания параметров, внутри фигурных скобок, размещаются инструкции, которые будут выполняться при каждом вызове функции. Фигурные скобки указываются в любом случае, даже если тело функции состоит только из одной инструкции. Точка с запятой после закрывающей фигурной скобки не указывается.

Вернуть значение из функции позволяет оператор return. После исполнения этого оператора выполнение функции останавливается и управление передается обратно в точку вызова функции. Это означает, что инструкции после оператора return никогда не будут выполнены. При использовании оператора return не должно быть неоднозначных ситуаций. Например, в этом случае возвращаемое значение зависит от условия:

```cpp
int sum(int x, int y) {
    if (x > 0) {
        return x + y;
    }
}
```

Если переменная x имеет значение больше нуля, то все будет нормально, но если переменная равна нулю или имеет отрицательное значение, то возвращаемое значение не определено и функция вернет произвольное значение, так называемый "мусор". В этом случае при компиляции выдаётся предупреждающее сообщение: "warning C4715: sum: значение возвращается не при всех путях выполнения". Чтобы избежать подобных

Листинги на странице http://unicross.narod.ru/cpp/
неоднозначностей, следует в конце тела функции вставить оператор return со значением по умолчанию:

```c
int sum(int x, int y) {
    if (x > 0) {
        return x + y;
    }
    return 0;
}
```

Если перед названием функции указано ключевое слово void, то оператора return может не быть. Однако, если необходимо досрочно прервать выполнение функции, то оператор return указывается без возвращаемого значения. Пример:

```c
void print_ok() {
    std::cout << "OK" << std::endl;
    return; // Преждевременное завершение функции
    std::cout << "Эта инструкция никогда не будет выполнена!!!";
}
```

При вызове функции из программы указывается название функции, после которого внутри круглых скобок передаются значения. Если функция не принимает параметров, то указываются только круглые скобки. Если функция возвращает значение, то его можно присвоить переменной или просто пропускать. Необходимо заметить, что количество и тип параметров в определении функции должны совпадать с количеством и типом параметров при вызове, иначе будет выведено сообщение об ошибке. Пример вызова трех функций:

```c
print ("Message"); // Функция выведет сообщение Message
print_ok(); // Вызываем функцию без параметров
z = sum(10, 20); // Переменной z будет присвоено значение 30
```

Переданные значения присваиваются переменным, расположенным в той же позиции в определении функции. Так при использовании функции `sum()` переменной `x` будет присвоено значение 10, а переменной `y` — значение 20. Результат выполнения функции присваивается переменной `z`.

В качестве примера создадим три разные функции и вызовем их (листинг 8.1).

Листинг 8.1. Создание функций и их вызов

```cpp
#include <iostream>

// Объявления функций
int sum(int x, int y); // или int sum(int, int);
void print(const char *str); // или void print(const char *);
void print_ok(); // или void print_ok(void);

int main() {
    // Вызов функций
    print("Message"); // Message
    print_ok(); // OK
    std::cout << sum(10, 20) << std::endl; // 30
    std::cin.get();
    return 0;
}

// Определения функций
int sum(int x, int y) { // Два параметра
    return x + y;
}

void print(const char *str) { // Один параметр
    std::cout << str << std::endl;
}

void print_ok() { // Без параметров
    std::cout << "OK" << std::endl;
}
```

Расположение объявлений и определений функций

Все инструкции в программе выполняются последовательно сверху вниз. Это означает, что прежде чем использовать функцию в программе, ее необходимо предварительно объявить. Поэтому объявление функции должно быть расположено перед вызовом функции. Обратите внимание на то, что размещать определение одной функции внутри другой нельзя. Таким образом, название функции всегда является глобальным идентификатором.

Листинги на странице http://unicross.narod.ru/cpp/
В небольших программах допускается объявление функции не указывать, при условии, что определение функции расположено перед функцией main() (листинг 8.2). Кстати, функция main() также не требует объявления, так как она вызывается первой.

Листинг 8.2. Расположение определения функции перед функцией main()

```cpp
#include <iostream>

int sum(int x, int y) { 
    return x + y;
}

int main() {
    std::cout << sum(10, 20) << std::endl; // 30
    std::cin.get();
    return 0;
}
```

При увеличении количества функций возникает ситуация, когда внутри функции вызывается вторая функция, а внутри второй вызывается третья и т.д. В результате приходится решать вопрос что было раньше курица или яйцо. Чтобы избежать такой ситуации объявления функций следует размещать в начале программы перед функцией main(), а определения — после функции main() (листинг 8.3). В этом случае порядок следования определений функций не имеет значения.

Листинг 8.3. Расположение объявлений и определений функций

```cpp
#include <iostream>

int sum(int, int);     // Объявление

int main() {
    std::cout << sum(10, 20) << std::endl; // 30
    std::cin.get();
    return 0;
}

int sum(int x, int y) { // Определение
    return x + y;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
При увеличении размера программы объявлений и определений функций становится все больше и больше. В этом случае программу разделяют на несколько отдельных файлов. Объявления функций выносят в заголовочный файл с расширением h (иногда hpp), а определения функций размещают в одноименном файле с расширением cpp. Все файлы располагают в одной папке с основным файлом, содержащим функцию main(). В дальнейшем с помощью директивы #include подключают заголовочный файл во всех остальных файлах. Если в директиве #include название заголовочного файла указывается внутри угловых скобок, то поиск файла осуществляется в системных папках. Если название указано внутри кавычек, то поиск вначале производится в папке с основным файлом, а затем в системных папках.

В качестве примера вынесем функцию sum() в отдельный файл (например, с названием "mymodule"), а затем подключим его к основному файлу. Для создания файла mymodule.cpp в окне Обозреватель решений щелкаем правой кнопкой мыши на пункте Файлы исходного кода и из контекстного меню выбираем пункт Добавить | Создать элемент. В результате откроется окно Добавление нового элемента. Выделяем пункт Файл C++ (.cpp), вводим название файла ("mymodule") и нажимаем кнопку Добавить. В результате файл будет добавлен в папку проекта и его название отобразится в окне Обозреватель решений, а сам файл будет открыт на отдельной вкладке. Вставляем в этот файл код из листинга 8.4.

Листинг 8.4. Файл mymodule.cpp

```cpp
#include "mymodule.h"

int sum(int x, int y) { // Определение
    return x + y;
}
```

Для создания файла mymodule.h в окне Обозреватель решений щелкаем правой кнопкой мыши на пункте Заголовочные файлы и из контекстного меню выбираем пункт Добавить | Создать элемент. В результате откроется окно Добавление нового элемента. Выделяем пункт Заголовочный файл (.h), вводим название файла ("mymodule") и нажимаем кнопку Добавить. Вставляем в этот файл код из листинга 8.5.

Листинги на странице http://unicross.narod.ru/cpp/
Листинг 8.5. Файл mymodule.h

```c
#ifndef MYMODULE_H
#define MYMODULE_H
#include <iostream>
int sum(int x, int y);
#endif
```

Все содержимое файла mymodule.h расположено внутри условия, которое проверяется перед компиляцией. Условие выглядит следующим образом:

```c
#ifndef MYMODULE_H
// Инструкции
#endif
```

Это условие следует читать так: "если не существует макроопределения MYMODULE_H, то вставить инструкции в то место, где подключается файл". Название макроопределений обычно совпадает с названием заголовочного файла. Только все буквы указываются в верхнем регистре и точка заменяется символом подчеркивания. Условие начинается с директивы `ifndef` и заканчивается директивой `endif`. Все это необходимо, чтобы объявления идентификаторов не вставлялись дважды. Для этого в первой инструкции внутри условия объявляется макроопределение `MYMODULE_H` с помощью директивы `define`. В этом случае повторная проверка условия вернет ложное значение. Для однократного включения файла можно также воспользоваться директивой `pragma`:

```c
#pragma once
```

Теперь, чтобы использовать функцию `sum()`, достаточно подключить файл mymodule.h к основной программе. Пример подключения приведен в листинге 8.6.

Листинг 8.6. Файл main.cpp

```c
#include "mymodule.h"

int main() {
    std::cout << sum(10, 20) << std::endl; // 30
    std::cin.get();
    return 0;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
В директиве `#include` допускается указывать не только название файла, но и абсолютный или относительный путь к нему. Это позволяет размещать файлы по различным папкам. Примеры указания абсолютного и относительного пути:

```
#include "C:\book\test\test\mymodule.h"
#include "C:/book/test/test/mymodule.h"
#include "/book/test/test/mymodule.h"
#include "folder/mymodule.h"
```

При использовании больших программ создают статическую (файлы с расширением `lib` в VC++) или динамическую (файлы с расширением `dll`) библиотеку. Статические библиотеки становятся частью программы при компиляции, а динамические библиотеки подгружаются во время выполнения программы. Процесс создания библиотек мы будем изучать немного позже, а сейчас рассмотрим способ указания пути к папкам со статическими библиотеками.

При использовании статических библиотек заголовочные файлы обычно размещают в папке `include`, а сами библиотеки — в папке `lib`. Далее добавляют пути к этим папкам в свойства проекта. Для этого запускаем проект и из меню **Вид** выбираем пункт **Диспетчер свойств**. В окне **Диспетчер свойств** раскрываем список, например, **Debug\Win32**, и щелкаем правой кнопкой мыши на строке **Microsoft.Cpp.Win32.user**. Из контекстного меню выбираем пункт **Свойства**. В открывшемся окне выделяем пункт **Каталоги VC++**. Выделяем строку **Каталоги включения**. Щелкаем мышью на появившейся кнопке со стрелкой вниз и выбираем пункт **Изменить**. Добавляем путь к папке `include` и нажимаем кнопку **OK**. Далее выделяем строку **Каталоги библиотеки**. Щелкаем мышью на появившейся кнопке со стрелкой вниз и выбираем пункт **Изменить**. Добавляем путь к папке `lib` и нажимаем кнопку **OK**. Сохраняем все файлы проекта. В результате все изменения будут сохранены в файле C:\Documents and Settings\<Пользователь>\Local Settings\Application Data\Microsoft\MSBuild\v4.0\Microsoft.Cpp.Win32.user. Этот файл будет подключаться ко всем проектам определенного пользователя. Содержимое файла выглядит примерно так:

```xml
<?xml version="1.0" encoding="utf-8"?>
<Project DefaultTargets="Build" ToolsVersion="4.0"
 xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <IncludePath>C:\book\include;$(IncludePath)</IncludePath>
  </PropertyGroup>
</Project>
```

Способы передачи параметров в функцию

Как вы уже знаете, после названия функции, внутри круглых скобок, указывается тип и название параметров через запятую. Если функция не принимает параметров, то указываются только круглые скобки или внутри круглых скобок задается ключевое слово void (в языке С ключевое слово void является обязательным). В определении функции после типа обязательно должно быть указано название параметра, которое является локальной переменной. Эта переменная создается при вызове функции, а после выхода из функции она удаляется. Таким образом, локальная переменная видна только внутри функции и ее значение между вызовами не сохраняется (исключением являются статические переменные). Если название локальной переменной совпадает с названием глобальной переменной, то все операции будут производиться с локальной переменной, а значение глобальной не изменится. Чтобы в этом случае обратиться к глобальной переменной, необходимо перед ее названием указать оператор ::.

При вызове функции указывается название функции, после которого внутри круглых скобок передаются значения. Количество и тип параметров в определении функции должны совпадать с количеством и типом параметров при вызове, иначе будет выведено сообщение об ошибке. Переданные значения присваиваются переменным, расположенным в той же позиции в определении функции. Так при вызове функции sum(10, 20) (прототип int sum(int x, int y)) переменной x будет присвоено значение 10, а переменной y — значение 20. Если функция не принимает параметров, то указываются только круглые скобки.

По умолчанию в функцию передается копия значения переменной, а не ссылка на переменную. Таким образом, изменение значения внутри функции не затронет значение исходной переменной. Пример передачи параметра по значению приведен в листинге 8.7.

<table>
<thead>
<tr>
<th>Листинг 8.7. Передача параметра по значению</th>
</tr>
</thead>
</table>

```cpp
#include <iostream>

void func(int x);
```

Листинги на странице http://unicross.narod.ru/cpp/
```cpp
int main() {
    int x = 10;
    func(x);
    std::cout << x << std::endl; // 10, а не 30
    std::cin.get();
    return 0;
}
void func(int x) {
    x = x + 20; // Значение никогда не сохраняется!
}
```

Передача копий значения для чисел является хорошим решением, но при использовании массивов и объектов, а также при необходимости изменить значение исходной переменной, лучше применять передачу указателя. Для этого при вызове функции перед названием переменной указывается оператор & (взятие адреса), а в прототипе функции объявляется указатель. В этом случае в функцию передается не значение переменной, а ее адрес. Внутри функции вместо переменной используется указатель. Пример передачи указателя приведен в листинге 8.8.

Листинг 8.8. Передача указателя в функцию

```cpp
#include <iostream>

void func(int *y);

int main() {
    int x = 10;
    func(&x); // Передаем адрес
    std::cout << x << std::endl; // 30, а не 10
    std::cin.get();
    return 0;
}
void func(int *y) {
    *y = *y + 20; // Изменяется значение переменной x
}
```

Однако, если функция большая, то разыменовывать указатель при каждой операции не очень удобно. В языке C++ существует еще один способ передачи параметров —

Листинги на странице http://unicross.narod.ru/cpp/
mekhanizm slyolk. Sut' ego zakluchayetsya v tom, chto vnutri funktsii peremennaya
jevlyayetsya psevdonimom isходnoj peremennoj. Lyubye izmeneniya psevdonima
otrajatsya na isходnoj peremennoj. Pri ispol'zovanii mekhanizma slyolk pered
nazyvaniem peremennoj v ob'явlenii i oprodenii funktsii ukazuyeetsya operator &,
a pri vyove — toльko nazyvanie peremennoj. Primer ispol'zovanii mekhanizma slyolk
primeden v liztinge 8.9.

Листинг 8.9. Использование механизма ссылок

```cpp
#include <iostream>

void func(int &y);

int main() {
    int x = 10;
    func(x);
    std::cout << x << std::endl; // 30, а не 10
    std::cin.get();
    return 0;
}
void func(int &y) {
    // Переменная y является псевдонимом переменной x
    y = y + 20; // Изменяется значение переменной x
}
```

Передача массивов в функцию

Передача одномерных массивов и C-строк осуществляется с помощью указателей.
Obратite vnimanie na to, chto pri vyove funktsii pered nazyvaniem peremennoj ne
nужno добавлять operator &, так как название переменной содержит adres pervogo
elementa massiva. Primer передачи C-строки приведен в liztinge 8.10.

Листинг 8.10. Передача C-строки в функцию

```cpp
#include <iostream>

void func1(char *s);
```

Листинги на странице http://unicross.narod.ru/cpp/
void func2(char s[]);

int main() {
    char str[] = "String";
    std::cout << sizeof(str) << std::endl; // 7
    func1(str); // Оператор & перед str не указывается!!!
    func2(str);
    std::cout << str << std::endl; // string
    std::cin.get();
    return 0;
}

void func1(char *s) {
    s[0] = 's'; // Изменяется значение элемента массива str
    std::cout << sizeof(s) << std::endl; // 4, а не 7!!!
}

void func2(char s[]) {
    s[5] = 'G'; // Изменяется значение элемента массива str
}

Объявление char *s эквивалентно объявлению char s[]. И в том и в другом случае объявляется указатель на тип char. Обратите внимание на значения, возвращаемые оператором sizeof вне и внутри функции. Вне функции оператор возвращает размер всего символьного массива, в то время как внутри функции оператор sizeof возвращает только размер указателя. Это происходит потому что внутри функции переменная s является указателем, а не массивом. Поэтому если внутри функции необходимо знать размер массива, то количество элементов (или размер в байтах) следует передавать в дополнительном параметре.

При передаче многомерного массива необходимо явным образом указывать все размеры (например, int a[4][4][4]). Самый первый размер допускается не указывать (например, int a[][4][4]). Пример передачи двухмерного массива в функцию приведен в листинге 8.11.

**Листинг 8.11. Передача двухмерного массива**

```cpp
#include <iostream>

void func(int a[][4]);
```

Листинги на странице http://unicross.narod.ru/cpp/
int main() {
    const short ARR_ROWS = 2, ARR_COLS = 4;
    int i, j;
    int arr[ARR_ROWS][ARR_COLS] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8}
    };
    func(arr);
    // Выводим значения
    for (i=0; i<ARR_ROWS; ++i) {
        for (j=0; j<ARR_COLS; ++j) {
            std::cout.width(2); // Ширина поля
            std::cout << arr[i][j];
        }
        std::cout << std::endl;
    }
    std::cin.get();
    return 0;
}

void func(int a[][4]) { // или void func(int a[2][4])
    a[0][0] = 80;
}

Такой способ передачи многомерного массива нельзя назвать универсальным, так как существует жесткая привязка к размеру. Одним из способов решения проблемы является создание дополнительного массива указателей. В этом случае в функцию передается адрес первого элемента массива указателей, а объявление параметра в функции выглядит так:

int *a[]

или так:

int **a

Пример передачи массива указателей приведен в листинге 8.12.

Листинги на странице http://unicross.narod.ru/cpp/
Листинг 8.12. Передача массива указателей

```cpp
#include <iostream>

void func(int *a[], short rows, short cols);

int main() {
    const short ARR_ROWS = 2, ARR_COLS = 4;
    int arr[ARR_ROWS][ARR_COLS] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8}
    };
    // Создаем массив указателей
    int *parr[] = {arr[0], arr[1]};
    // Передаем адрес массива указателей
    func(parr, ARR_ROWS, ARR_COLS);
    std::cin.get();
    return 0;
}

void func(int *a[], short rows, short cols) {
    // или void func(int **a, short rows, short cols)
    int i, j;
    for (i=0; i<rows; ++i) {
        for (j=0; j<cols; ++j) {
            std::cout.width(2);
            std::cout << a[i][j];
        }
        std::cout << std::endl;
    }
}
```

Однако и этот способ имеет недостаток, так как нужно создавать дополнительный массив указателей. Наиболее приемлемым способом является передача многомерного массива как одномерного. В этом случае в функцию передается адрес первого элемента массива, а в параметре объявляется указатель. Так как все элементы многомерного массива располагаются в памяти последовательно, зная количество элементов или размеры можно вычислить положение текущего элемента, используя адресную

Листинги на странице http://unicross.narod.ru/cpp/
арифметику. Пример передачи двухмерного массива как одномерного показан в листинге 8.13.

Листинг 8.13. Передача двухмерного массива как одномерного

```cpp
#include <iostream>

void func(int *a, short rows, short cols);

int main() {
    const short ARR_ROWS = 2, ARR_COLS = 4;
    int arr[ARR_ROWS][ARR_COLS] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8}
    };
    // Передаем в функцию адрес первого элемента массива
    func(arr[0], ARR_ROWS, ARR_COLS);
    std::cin.get();
    return 0;
}

void func(int *a, short rows, short cols) {
    int i, j;
    for (i=0; i<rows; ++i) {
        for (j=0; j<cols; ++j) {
            std::cout.width(2);
            std::cout << a[i * cols + j]; // Вычисляем положение элемента
        }
        std::cout << std::endl;
    }
}
```

Передача в функцию массива C-строк осуществляется также как передача массива указателей. Чтобы в функцию не передавать количество элементов можно при инициализации массива C-строк последнему элементу присвоить нулевой указатель. Этот элемент будет служить ориентиром конца массива. В качестве примера выведем все строки внутри функции (листинг 8.14).

Листинги на странице http://unicross.narod.ru/cpp/
Листинг 8.14. Передача массива C-строк

```cpp
#include <iostream>

void func(char *s[]);

int main() {
    char *str[] = {"String1", "String2", "String3", 0}; // Вставляем нулевой указатель, чтобы был ориентир
    func(str);
    std::cin.get();
    return 0;
}
void func(char *s[]) { // или void func(char **s)
    while (*s) { // Выводим все строки
        std::cout << *s << std::endl;
        ++s;
    }
}
```

Необязательные параметры

Чтобы сделать некоторые параметры необязательными, следует в объявлении функции присвоить параметру начальное значение. В этом случае, если при вызове функции параметр не указан, то переменной будет присвоено начальное значение. Обратите внимание на то, что начальное значение указывается только в объявлении функции. Повторять его в определении функции не нужно. Следует также заметить, что необязательные параметры должны следовать после обязательных, иначе будет выведено сообщение об ошибке. В качестве примера сделаем второй параметр функции `sum()` необязательным (листинг 8.15).

Листинг 8.15. Необязательные параметры

```cpp
#include <iostream>

int sum(int x, int y=2); // Указываем начальное значение
```

Листинги на странице http://unicross.narod.ru/cpp/
```cpp
int main() {
    std::cout << sum(10, 20) << std::endl; // 30
    std::cout << sum(10) << std::endl; // 12
    std::cin.get();
    return 0;
}

int sum(int x, int y) {
    // Начальное значение в определении не указывается!!!
    return x + y;
}
```

### Переменное количество параметров

Количество параметров в функции может быть произвольным, при условии, что существует один обязательный параметр. В объявлении и определении функции переменное число параметров обозначается тремя точками. Получить доступ к этим параметрам внутри функции можно несколькими способами.

Первый способ заключается в передаче количества параметров в обязательном параметре. В этом случае адрес последнего обязательного параметра и сохраняют его в указателе. Далее с помощью указателя происходит перемещение к следующему параметру. В качестве примера напишем функцию суммирования произвольного количества целых чисел (листинг 8.16).

#### Листинг 8.16. Суммирование произвольного количества целых чисел

```cpp
#include <iostream>

int sum(int x, ...);

int main() {
    std::cout << sum(2, 20, 30) << std::endl; // 50
    std::cout << sum(3, 1, 2, 3) << std::endl; // 6
    std::cout << sum(4, 1, 2, 3, 4) << std::endl; // 10
    std::cin.get();
    return 0;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
```c
int sum(int x, ...) {  
    int result = 0;
    int *p = &x;       // Получаем адрес последнего параметра
    for(int i=0; i<x; ++i) {  
        ++p;       // Перемещаем указатель на следующий параметр
        result += *p; // Прибавляем очередное число
    }
    return result;
}
```

В этом примере тип обязательного параметра совпадает с типом остальных параметров. Если мы захотим просуммировать произвольное количество вещественных чисел, то можно изменить тип обязательного параметра, с помощью которого передается количество параметров, или при перемещении указателя выполнять приведение типов (листинг 8.17).

### Листинг 8.17. Суммирование произвольного количества вещественных чисел

```c
#include <iostream>

double sum(int x, ...);

double sum(int x, ...) {  
    double result = 0.0, *pd = 0;
    int *pi = &x;       // Получаем адрес последнего параметра
    ++pi;       // Перемещаем указатель до приведения типов!!!
    pd = reinterpret_cast<double *>(pi); // Приведение типов
    for(int i=0; i<x; ++i) {  
        result += *pd; // Прибавляем очередное число
        ++pd;       // Перемещаем указатель на следующий параметр
    }
}
```

Листинги на странице http://unicross.narod.ru/cpp/
return result;
}
При передаче в функцию произвольного количества строк в последнем параметре можно задать нулевой указатель. В этом случае передавать количество параметров явным образом не нужно. Для примера передадим в функцию несколько строк (листинг 8.18). Внутри функции выведем содержимое строк в окно консоли.

Листинг 8.18. Передача произвольного количества C-строк

```cpp
#include <iostream>

void func(char *s, ...);

int main() {
    char str1[] = "string1", str2[] = "string2";
    func(str1, str2, 0);       // Нулевой указатель
    func(str1, str2, str1, 0); // в последнем параметре!
    std::cin.get();
    return 0;
}
void func(char *s, ...) {
    char **p = &s;
    while (*p) {
        std::cout << *p << std::endl;
        ++p;
    }
}
```

Следующий способ состоит в использовании специальных макросов va_start(), va_arg() и va_end(). Прототипы макросов:

```cpp
#include <cstdarg>
void va_start(va_list <Указатель>, <Последний параметр>)
<Tип> va_arg(va_list <Указатель>, <Тип данных>)
void va_end(va_list <Указатель>)
```

Тип va_list определен следующим образом:

typedef char * va_list;

Листинги на странице http://unicross.narod.ru/cpp/
Вначале объявляется указатель типа va_list. Далее должна производиться инициализация указателя с помощью макроса va_start(). В первом параметре передается указатель, а во втором — название последнего обязательного параметра. Доступ к параметрам осуществляется с помощью макроса va_arg(), который возвращает значение текущего параметра и перемещает указатель на следующий параметр. В первом параметре макроса va_arg() передается указатель, а во втором — название типа данных. Макрос va_end() сообщает об окончании перебора параметров. Пример использования макросов va_start(), va_arg() и va_end() приведен в листинге 8.19.

### Листинг 8.19. Использование макросов va_start(), va_arg() и va_end()

```cpp
#include <iostream>
#include <cstdarg>

int sum(int x, ...);

int main() {
    std::cout << sum(2, 20, 30) << std::endl; // 50
    std::cout << sum(3, 1, 2, 3) << std::endl; // 6
    std::cout << sum(4, 1, 2, 3, 4) << std::endl; // 10
    std::cin.get();
    return 0;
}

int sum(int x, ...) {
    int result = 0;
    std::va_list p;
    va_start(p, x);
    for(int i=0; i<x; ++i) {
        result += va_arg(p, int);
    }
    va_end(p);
    return result;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
Перегрузка функций

В предыдущих разделах мы создали функцию `sum()`, предназначенную для суммирования двух целых чисел. В один прекрасный момент возникнет ситуация, когда потребуется произвести суммирование вещественных чисел. Что в этом случае делать? Создавать функцию с другим названием? В языке C++ существует решение этой ситуации, называемое перегрузкой функции. Перегрузка функции — это возможность использования одного названия для нескольких функций, различающихся типом параметров или их количеством. Изменение только типа возвращаемого значения недостаточно для перегрузки функции. В качестве примера перегрушим функцию `sum()` таким образом, чтобы ее название можно было использовать для суммирования как целых чисел, так и вещественных (листиг 8.20).

Листинг 8.20. Перегрузка функций

```cpp
#include <iostream>

int sum(int x, int y);
double sum(double x, double y);

int main() {  
   // Суммирование целых чисел
   std::cout << sum(10, 20) << std::endl;  // 30
   // Суммирование вещественных чисел
   std::cout << sum(10.5, 20.7) << std::endl;  // 31.2
   std::cin.get();
   return 0;
}

int sum(int x, int y) {  
   return x + y;
}

double sum(double x, double y) {  
   return x + y;
}
```

При вызове перегруженной функции может возникнуть неоднозначность, при которой компилятор не может выбрать, какую функцию следует вызвать. Виновниками таких ситуаций бывают значения по умолчанию, автоматическое преобразование типа при

Листинги на странице http://unicross.narod.ru/cpp/
передаче значения в функцию, а также возможность множественной записи при объявлении параметра. Пример проблемы при использовании значений по умолчанию:

```c++
int sum(int x);
int sum(int x, int y=2);
...
std::cout << sum(10, 20) << std::endl; // Нормально
std::cout << sum(10) << std::endl;    // Неоднозначность
```

В следующем примере нельзя сделать выбор в какой тип данных (в тип float или double) следует автоматически преобразовать целое число:

```c++
float sum(float x, float y);
double sum(double x, double y);
...
std::cout << sum(10.5, 20.4) << std::endl; // Нормально
std::cout << sum(10, 20) << std::endl;    // Неоднозначность
```

Если бы существовала функция только с параметрами, имеющими тип double, то целое число было бы автоматически преобразовано в тип double. Аналогичная проблема возникает при одновременном использовании знакового и беззнакового типов:

```c++
void print(char ch);
void print(unsigned char ch);
...
print('S');        // Нормально
print(119);        // Неоднозначность
```

Следующий пример демонстрирует проблему из-за возможности множественной записи при объявлении параметра (объявление char *str эквивалентно char str[]):

```c++
void print(char *str);
void print(char str[]);
...
print("String");        // Неоднозначность
```

Кроме того, следует учитывать, что разный способ передачи параметров в функцию также может приводить к неоднозначности:

```c++
void print(int x);
void print(int &x);
...
int n = 25;
print(n);              // Неоднозначность
```

Листинги на странице http://unicross.narod.ru/cpp/
Шаблонные (обобщенные) функции

Если присмотреться к реализации функции `sum()`, то можно заметить, что вне зависимости от типа параметров внутри функции будет одна и та же инструкция `return x + y;`, так как оператор `+` позволяет сложить числа любого типа. Для таких случаев в языке С++ вместо перегрузки функции следует применять шаблонные функции. Шаблонные функции называют также обобщенными. Компилятор на основе шаблонной функции автоматически создаст перегруженные версии функции в зависимости от имеющихся способов её вызова в программе. Описывается шаблонная функция по следующей схеме:

```cpp
template<class T1[, ..., class Tn]>  
Тип Название_функции(Тип Название_переменной1 [, ..., Тип Название_переменнойn])
{
    Инструкции;
}
```

После ключевого слова `template` внутри угловых скобок через запятую указываются обобщенные названия типов. Эти названия используются для описания типов параметров и могут использоваться внутри функции. При компиляции обобщенные типы будут заменены реальными типами данных. Перед названием обобщенного типа могут быть указаны ключевые слова `class` или `typename`, которые обозначают одно и то же. Остальное описание шаблонной функции совпадает с описанием обычной функции, только вместо реальных типов данных указываются названия обобщенных типов, перечислённых после ключевого слова `template`. В качестве примера создадим шаблонную функцию для сложения чисел (листинг 8.21).

### Листинг 8.21. Шаблонные функции

```cpp
#include <iostream>

template<class Type>  
Type sum(Type x, Type y);

int main() {
    std::cout << sum(10, 20) << std::endl; // 30
    std::cout << sum(10.5, 20.4) << std::endl; // 30.9
    std::cout << sum(10.5f, 20.7f) << std::endl; // 31.2
}
```

Листинги на странице http://unicross.narod.ru/cpp/
std::cin.get();
return 0;
}

template<class Type>
Type sum(Type x, Type y) {
    return x + y;
}

Компилятор на основе шаблонной функции и способах ее вызова автоматически создаст следующие перегруженные версии функции:

int sum(int x, int y);
float sum(float x, float y);
double sum(double x, double y);

Шаблонная функция sum() из листинга 8.21 позволяет складывать числа только одного типа. Чтобы можно было складывать числа разных типов, например, int и double, следует объявить два разных обобщенных типа:

template<class Type1, class Type2>
Type1 sum(Type1 x, Type2 y) {
    return x + y;
}

Обобщенные и реальные типы можно смешивать в объявлении параметров шаблонной функции. Шаблонные функции допускается перегружать. Существуют два способа перегрузки. Первый способ заключается в определении обычной функции с конкретными параметрами. Определение функции во втором случае выглядит так:

template<>
Тип Название_функции<Type1, ..., TypeN>(Тип1 Название_переменной1
    [, ..., TypeN Название_переменнойN])
{
    Инструкции;
}

После ключевого слова template указываются пустые угловые скобки, а после названия функции внутри угловых скобок через запятую перечисляются реальные типы данных. Пример перегрузки шаблонных функций приведен в листинге 8.22.

Листинги на странице http://unicross.narod.ru/cpp/
#include <iostream>

template<class Type1, class Type2>
Type1 sum(Type1 x, Type2 y);
int sum(int x, int y);

double sum<double, double>(double x, double y);

int main() {
    std::cout << sum(10, 20) << std::endl;        // 30
    std::cout << sum(10.5, 20.4) << std::endl;    // 30.9
    std::cout << sum(10.5f, 20.7f) << std::endl;   // 31.2
    std::cout << sum(10.5, 20) << std::endl;      // 30.5
    std::cout << sum(10.5, 2.3, 10) << std::endl; // 22.8
    std::cin.get();
    return 0;
}

template<class Type1, class Type2>
Type1 sum(Type1 x, Type2 y) { // Способ 1
    return x + y;
}

int sum(int x, int y) { // Способ 1
    return x + y;
}

template<double, double>
sum(double x, double y) { // Способ 2
    return x + y;
}

// Перегрузка и смешивание обобщенных и явных типов

template<class Type1, class Type2>
Type1 sum(Type1 x, Type2 y, int z) {
    return x + y + z;

Листинги на странице http://unicross.narod.ru/cpp/
Встраиваемые функции

Передача управления в функцию сопряжена с потерей скорости выполнения программы, так как вызов функции, передача в нее параметров и возврат значений требуют дополнительных затрат времени. Если размер функции небольшой, то имеет смысл объявить такую функцию встраиваемой. В этом случае при компиляции содержимое функции будет вставлено в место вызова функции. Тем не менее следует помнить, что при этом происходит увеличение размера исполняемого файла. Поэтому, если большая функция вызывается много раз, то лучше ее оставить обычной функцией, в то время как функция, состоящая из одной инструкции, является первым кандидатом для встраивания.

Чтобы объявить функцию встраиваемой, следует перед функцией добавить ключевое слово inline. Следует учитывать, что ключевое слово inline является лишь рекомендацией компилятору и может быть проигнорировано. В качестве примера объявим функцию sum() встраиваемой (листинг 8.23).

Листинг 8.23. Встраиваемые функции

```cpp
#include <iostream>

inline int sum(int x, int y);

int main() {
    std::cout << sum(10, 20) << std::endl;
    std::cin.get();
    return 0;
}
inline int sum(int x, int y) {
    return x + y;
}
```

Создать встраиваемую функцию можно также с помощью директивы #define. Значение, указанное в этой директиве, подставляется в место вызова функции до компиляции. Название, указанное в директиве #define, принято называть макроопределением или макросом. Директива имеет следующий формат:

```cpp
#define <Название функции>(<Параметры>) <Инструкция>
```

Листинги на странице http://unicross.narod.ru/cpp/
Пример использования директивы `#define` приведен в листинге 8.24.

```cpp
#include <iostream>
#define SUM(x, y) (x + y)

int main() {
    std::cout << SUM(10, 20) << std::endl;
    std::cin.get();
    return 0;
}
```
Обратите внимание на то, что в конце инструкции точка с запятой не указывается. Концом инструкции является конец строки. Если точку с запятой указать, то значение вместе с ней будет вставлено в выражение. Например, если определить макрос так:
```cpp
define SUM(x, y) (x + y);
```
tо после подстановки значения, инструкция
```cpp
std::cout << SUM(10, 20) << std::endl;
```
будет выглядеть следующим образом:
```cpp
std::cout << (10 + 20); << std::endl;
```
Точка с запятой после закрывающей круглой скобки является концом инструкции, поэтому этот код вызовет ошибку при компиляции. Однако в следующем примере ошибки не будет, но результат будет совершенно другим. Инструкция
```cpp
int z = SUM(10, 20) + 40;
```
после подстановки будет выглядеть следующим образом:
```cpp
int z = (10 + 20); + 40;
```
Подобная ситуация приводит к ошибкам, которые трудно найти, так как в этом случае инструкция `+ 40` не возбуждает ошибку при компиляции.

Обратите также внимание на то, что выражение внутри тела макроса расположено внутри круглых скобок. Если скобки не указать, то это может привести к недоразумениям, так как никакого вычисления выражения не производится. Все выражение внутри тела макроса после подстановки формальных параметров целиком вставляется в место вызова макроса. Например, если определить макрос следующим образом:
```cpp
#define SUM(x, y) x + y
```
Листинги на странице http://unicross.narod.ru/cpp/
то после подстановки значения, инструкция

```c
int z = SUM(10, 20) * 40;
```

будет выглядеть следующим образом:

```c
int z = 10 + 20 * 40;
```

Приоритет оператора умножения выше приоритета оператора сложения, поэтому число 20 будет умножено на 40, а затем к результату прибавлено число 10. Таким образом, результат будет 810, а не 1200.

При указании длинного выражения внутри тела функции следует учитывать, что определение макроса должно быть расположено на одной строке. Если нужно разместить выражение на нескольких строках, то в конце строки необходимо добавить обратную косую черту. После косой черты не должно быть никаких символов, в том числе и комментариев. Пример:

```c
#define SUM(x, y) \
(x + y)
```

**Константные параметры**

Если внутри функции значение параметра не изменяется, то такой параметр следует объявить константным. Для этого при объявлении перед параметром указывается ключевое слово `const`. Например, функция `sum()`, предназначенная для суммирования чисел, не производит изменение значений параметров, поэтому параметры можно объявить константными:

```c
int sum(const int x, const int y) {
    return x + y;
}
```

При использовании указателей важно учитывать местоположение ключевого слова `const`. Например, следующие объявления не эквивалентны:

```c
void print(const char *s);
void print(char const *s);
void print(char * const s);
void print(const char * const s);
```

Первые два объявления являются эквивалентными. В этом случае изменить значение, на которое ссылается указатель, нельзя, но указателю можно присвоить другой адрес:

```c
void print(const char *s) {
    char s2[] = new;
```

s = s2;        // Нормально
s[0] = 's';     // Ошибка
std::cout << s << std::endl;
}

При третьем объявлении изменить значение, на которое ссылается указатель, можно, но указателю нельзя присвоить другой адрес:

void print(char * const s) {
    char s2[] = "New";
    s = s2;        // Ошибка
    s[0] = 's';     // Нормально
    std::cout << s << std::endl;
}

Четвертое объявление запрещает изменение значения, на которое ссылается указатель, и присвоение другого адреса:

void print(const char * const s) {
    char s2[] = "New";
    s = s2;        // Ошибка
    s[0] = 's';     // Ошибка
    std::cout << s << std::endl;
}

С помощью оператора const_cast можно отменить действие ключевых слов const и volatile. Формат оператора:
const_cast< <Тип результата> > ( <Выражение> )

Пример приведения константного указателя в обычный внутри функции:

void print(const char *s) {
    // s[0] = 's';        // Ошибка
    char *p = const_cast<char *> (s);
    p[0] = 's';     // Нормально
    std::cout << s << std::endl;
}

Пример удаления действия ключевого слова const у ссылки внутри функции:

void func(const int &x) {
    // x = 50;  // Ошибка!!!
    // Теперь можно изменить значение
    const_cast<int &> (x) = 30;

    Листинги на странице http://unicross.narod.ru/cpp/
Статические переменные

Переменные, указанные в параметрах, а также переменные, объявленные внутри функции, являются локальными переменными. Эти переменные создаются при вызове функции, а после выхода из функции они удаляются. Таким образом, локальная переменная видна только внутри функции. Если внутри функции при объявлении локальной переменной не было присвоено начальное значение, то переменная будет содержать произвольное значение, так называемый "мусор". Если название локальной переменной совпадает с названием глобальной переменной, то все операции будут производиться с локальной переменной, а значение глобальной не изменится. Чтобы в этом случае обратиться к глобальной переменной, необходимо перед названием переменной указать оператор ::. Пример сохранения промежуточного значения между вызовами функции в глобальной переменной приведен в листинге 8.25.

Листинг 8.25. Сохранение промежуточного значения в глобальной переменной

```cpp
#include <iostream>

int x = 0;
void sum(int x);

int main() {
    sum(10);
    sum(20);
    std::cout << x << std::endl; // 30
    std::cin.get();
    return 0;
}
void sum(int x) {
    ::x += x;
}
```

Статические переменные позволяют отказаться от использования глобальных переменных, для сохранения промежуточных значений между вызовами функции. Инициализация статической переменной производится только при первом вызове функции. Если при объявлении статической переменной не присвоено начальное

Листинги на странице http://unicross.narod.ru/cpp/
значение, то переменная автоматически инициализируется нулевым значением. После завершения работы функции статическая переменная сохраняет свое значение, которое доступно при следующем вызове функции. При объявлении статической переменной перед типом данных указывается ключевое слово static. В качестве примера переделаем предыдущий пример и используем статическую переменную вместо глобальной (листинг 8.26).

<table>
<thead>
<tr>
<th>Листинг 8.26. Использование статических переменных</th>
</tr>
</thead>
</table>

#include <iostream>

int sum(int y);

int main() {
    std::cout << sum(10) << std::endl; // 10
    std::cout << sum(20) << std::endl; // 30
    std::cin.get();
    return 0;
}

int sum(int y) {
    static int x = 0; // Статическая переменная
    x += y;
    return x;
}

Способы возврата значения из функции

Вернуть значение из функции позволяет оператор return. После исполнения этого оператора выполнение функции останавливается и управление передается обратно в точку вызова функции. Это означает, что инструкции после оператора return никогда не будут выполнены. Если внутри функции нет оператора return, то по достижении закрывающей фигурной скобки управление будет передано в точку вызова функции. В этом случае возвращаемое значение не определено.

Если функция не возвращает никакого значения, то вместо типа данных в объявлении и определении функции указывается ключевое слово void. Внутри такой функции оператора return может не быть, однако его можно использовать без указания значения для преждевременного выхода из функции. Вызов функции, не возвращающей
ничего значения, нельзя размещать внутри какой-либо инструкции. Только в отдельной инструкции. Пример функции, которая не возвращает никакого значения:

```cpp
void print(const char *s) {
    std::cout << s << std::endl;
}
```

В остальных случаях в объявлении и определении функции перед названием функции задается возвращаемый тип данных. Значение этого типа должно быть указано в операторе return. В этом случае возвращается копия значения. Функция может вернуть значение любого типа, кроме массива. Работать с массивами необходимо через параметры функции, передавая указатель на него, или возвращая указатель на конкретный элемент. Вызов функции, возвращающей какое-либо значение, можно разместить внутри выражения с правой стороны от оператора =. Если функция возвращает ссылку, то допускается размещение функции и с левой стороны от оператора =. Возвращаемое значение можно присвоить переменной или просто проигнорировать. Пример функции, возвращающей копию значения:

```cpp
int sum(const int x, const int y) {
    return x + y;
}
```

При использовании оператора return не должно быть неоднозначных ситуаций. Например, в этом случае возвращаемое значение зависит от условия:

```cpp
int sum(int x, int y) {
    if (x > 0) {
        return x + y;
    }
}
```

Если переменная x имеет значение больше нуля, то все будет нормально, но если переменная равна нулю или имеет отрицательное значение, то возвращаемое значение не определено и функция вернет произвольное значение, так называемый "мусор". В этом случае при компиляции выводится предупреждающее сообщение: "warning C4715: sum: значение возвращается не при всех путях выполнения". Чтобы избежать подобных неоднозначностей, следует в конце тела функции вставить оператор return со значением по умолчанию:

```cpp
int sum(int x, int y) {
    if (x > 0) {
        return x + y;
    }
    return 0;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
Функция может возвращать указатель. В этом случае в объявлении и определении функции указывается соответствующий тип указателя. В качестве примера вернем указатель на последний символ строки или нулевой указатель, если строка пустая (листинг 8.27).

```cpp
#include <iostream>
#include <cstring>

char *func(char *s);

int main() {
    char *p = 0, str[] = "String";
    p = func(str);
    if (p) { // Если не нулевой указатель
        std::cout << *p << std::endl; // g
    } else std::cout << "NULL" << std::endl;
    std::cin.get();
    return 0;
}
char *func(char *s) {
    int len = std::strlen(s); // Получаем длину строки
    if (!len) return 0; // Нулевой указатель, если пусто
    else return &s[len - 1]; // Указатель на последний символ
}
```

В языке C++ функции могут возвращать также ссылки. В этом случае допускается размещение функции как с правой, так и с левой стороны от оператора =. Чтобы функция возвращала ссылку необходимо в объявлении и определении после типа данных указать оператор &. В качестве примера напишем функцию, которая позволяет получить или изменить значение элемента массива по указанному индексу (листинг 8.28).

Листинги на странице http://unicross.narod.ru/cpp/
Листинг 8.28. Возврат ссылки

```cpp
#include <iostream>

int &at(int *a, int i);

int main() {
    int arr[] = {10, 20, 30};
    at(arr, 0) = 800;
    for (int i=0; i<3; ++i) {
        std::cout << at(arr, i) << " ";
    } // 800 20 30
    std::cout << std::endl;
    std::cin.get();
    return 0;
}
int &at(int *a, int i) {
    return a[i];
}
```

Указатели на функции

Функции также как и переменные имеют адрес, который можно сохранить в указателе. В дальнейшем через указатель можно вызвать эту функцию. Кроме того, допускается передавать указатель на функцию в качестве параметра другой функции. Объявление указателя на функцию выглядит так:

```cpp
<Tип> (*<Название указателя>)( [<Тип1>[, ..., <ТипN>]] );
```

Чтобы присвоить указателю адрес функции, необходимо указать название функции без параметров и круглых скобок справа от оператора =. Тип параметров указателя и функции должен совпадать. Пример объявления указателя и присваивания адреса функции:

```cpp
int (*p)(int, int);
p = sum;
```

Вызвать функцию через указатель можно двумя способами:

```cpp
x = (*p)(10, 20); // Аналогично x = sum(10, 20);
y = p(30, 10);
```

Листинги на странице http://unicross.narod.ru/cpp/
Пример объявления указателя, вызова функции через указатель и передачи указателя в качестве параметра функции приведен в листинге 8.29.

**Листинг 8.29. Указатели на функции**

```cpp
#include <iostream>

int sum(int x, int y);
int func(int x, int y, int (*p)(int, int));

int main() {
    int (*p)(int, int); // Объявление указателя на функцию
    p = sum;           // Присваивание адреса функции
    std::cout << (*p)(10, 20) << std::endl; // 30
    std::cout << p(30, 10) << std::endl;     // 40
    std::cout << func(5, 10, sum) << std::endl; // 15
    std::cin.get();
    return 0;
}
int sum(int x, int y) {
    return x + y;
}
int func(int x, int y, int (*p)(int, int)) {
    return (*p)(x, y);
}

**Рекурсия**

Рекурсия — это возможность функции вызывать саму себя. При каждом вызове функции создается новый набор локальных переменных. Рекурсию удобно использовать для перебора объекта, имеющего заранее неизвестную структуру, или выполнения неопределенного количества операций. Типичным применением рекурсии является вычисление факториала числа (листинг 8.30).

Листинги на странице http://unicross.narod.ru/cpp/
#include <iostream>

unsigned long long factorial(unsigned long n);

int main() {
    for (int i=3; i<11; ++i) {
        std::cout << i << "! = " << factorial(i) << std::endl;
    }
    std::cin.get();
    return 0;
}

unsigned long long factorial(unsigned long n) {
    if (n <= 1) return 1;
    else return n * factorial(n - 1);
}

Результат выполнения:
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800

Листинги на странице http://unicross.narod.ru/cpp/
Программирование на C++
в Visual Studio® 2010 Express

Глава 9. ООП

© Прохоренок Н.А., 2010 г., unicross@mail.ru

Этот DjVu-файл предоставляется КАК ЕСТЬ. Автор не несет никакой
ответственности за прямые или косвенные проблемы, связанные с
использованием данного файла. ВЫ ИСПОЛЬЗУЕТЕ ЕГО НА СВОЙ
СТРАХ И РИСК.

Все названия программных продуктов являются зарегистрированными
торговыми марками соответствующих фирм.

Запрещено:
- перепечатывать всю книгу или отдельные главы без письменного разрешения
  Автора;
- декомпилировать данный файл и преобразовывать его в любой другой формат.
Объектно-ориентированное программирование (ООП) — это способ организации программы, позволяющий использовать один и тот же код многократно. В отличие от функций ООП позволяет не только разделить программу на фрагменты, но и описать предметы реального мира в виде объектов, а также организовать связи между этими объектами.

Основные понятия

Основным "кирпичиком" ООП является класс. Класс включает набор переменных и функций для управления этими переменными. Переменные называются атрибутами (свойствами, переменными-членами или данными-членами), а функции — методами (или функциями-членами). В совокупности атрибуты и методы называются членами класса. После создания класса его название становится новым типом данных. Тем самым пользовательские классы расширяют возможности языка C++.

На основе класса можно создать множество объектов. При этом для каждого объекта создается свой набор локальных переменных. Например, при изучении строк мы рассматривали класс string. При объявлении переменной название класса указывалось вместо типа данных:

```cpp
std::string str1("String1"), str2("String2");
```

В этом примере определены два объекта str1 и str2, которые являются экземплярами класса string. Каждый объект хранит информацию о своей строке, указанной при инициализации. Изменение одного объекта не затрагивает значение другого объекта. Внутри каждого объекта, незаметно для нас, производятся операции по выделению, увеличению и освобождению объема памяти, необходимого для хранения строки. Тем самым при создании объекта или его изменении нам не нужно указывать конкретный объем памяти и следить за освобождением памяти.

Чтобы иметь возможность манипулировать строкой класс string предоставляет множество методов. Например, с помощью метода size() мы получали количество символов в строке. При вызове метода его название указывается после названия OBJECTA через точку:

```cpp
std::cout << str1.size() << std::endl;
```

Помимо методов, класс string перегружает некоторые операторы. Например, для конкатенации используется оператор +:

```cpp
str1 = str1 + str2;
```

Листинги на странице http://unicross.narod.ru/cpp/
Перегрузка операторов производится с помощью методов, имеющих специальные названия. Определен в классе "операторный" метод можно изменить поведение оператора по своему вкусу. Например, перегрузив оператор + можно вместо сложения производить вычитание. Однако пользователи класса вряд ли положительно оценивают такую инициативу. Тем не менее в некоторых случаях изменение смысла оператора довольно полезно. Например, класс cout перегружает оператор побитового сдвига << для вывода данных в окно консоли.

Проводя аналогию с реальным миром можно сказать, что телевизор является экземпляром класса (объектом), а проект по которому создавался телевизор является классом. По этому проекту можно создать множество телевизоров (множество экземпляров класса). Кнопки и разъемы на корпусе телевизора, да и сам корпус являются отдельными объектами соответствующего класса. Сам процесс нажатия кнопки, приводящий к включению или выключению телевизора, переключению канала и т. д. выполняется с помощью методов. Для сохранения текущего состояния кнопки, а также размеров кнопки, ее цвета и др. предназначены атрибуты. Одна и та же кнопка может выполнять разные действия. Например, одиночное нажатие кнопки выполняет одно действие, а удержание кнопки (перегрузка метода) в течение пяти секунд приводит к выполнению совершенно другого действия.

Все содержимое телевизора находится внутри корпуса и скрыто от глаз. Чтобы пользоваться телевизором абсолютно не нужно знать как он устроен внутри, достаточно иметь кнопки (интерфейс доступа) и руководство пользователя (документация к классу). Точно также разработчик класса может предоставить интерфейс доступа, а остальную часть кода защитить от изменения (сокрытие данных в ООП назвывается инкапсуляцией). В дальнейшем разработчик класса имеет возможность изменить внутреннюю реализацию класса, при этом не изменения интерфейс доступа. При необходимости пользователь может попытаться изменить класс. Это можно сравнить с попыткой снять заднюю крышку телевизора и попытаться что-то внутри сделать. Результат этого действия зависит от квалификации исполнителя. Если это мастер, то телевизор возможно будет лучше работать, но если любитель, то скорее всего придется покупать новый телевизор.

В один прекрасный момент разработчик телевизора решает выпустить новую модель, просто добавив внутрь корпуса DVD-плеер. Большинство блоков внутри телевизора останутся от предыдущей модели. Основные изменения коснутся корпуса. Таким образом, телевизор с DVD-плеером наследует конструкцию предыдущей модели, добавляет новую функциональность (DVD-плеер) и переопределяет некоторые методы (изменяет корпус).

Листинги на странице http://unicross.narod.ru/cpp/
В результате наследования в производном классе может измениться поведение какого-либо метода. Например, автобус и гусеничный трактор являются наследниками класса автомобиль, в котором определен метод, описывающий способ движения. Совершенно очевидно, что движение автобуса отличается от движения гусеничного трактора. Поэтому класс автобус может наследовать метод от класса автомобиль не переопределяя его, а класс гусеничный трактор должен переопределить метод. Во всех этих классах теперь имеется доступ к одноименному методу, но реализация этого метода отличается. Тем не менее, этот метод выполняет одно и то же действие — движение транспортного средства. В ООП такое явление называется полиморфизмом.

Итак, мы познакомились с тремя основными понятиями ООП:

➔ инкапсуляция — сокрытие данных внутри класса;
➔ наследование — возможность создания производных классов на основе базового класса. При этом производный класс автоматически получает возможности базового класса и может добавить новую функциональность или переопределить некоторые методы;
➔ полиморфизм — смысл действия, которое выполняет одноименный метод, зависит от объекта, над которым это действие выполняется.

Что же должно быть представлено в виде классов, а что в виде методов или атрибутов? Если слово является существительным (автомобиль, телевизор, кнопка и т. д.), то оно может быть описано как класс. Метод описывает изменение объекта, например, автомобиль начал движение, непрерывно движется, остановился. Атрибут предназначен для сохранения текущего состояния объекта и его характеристистик, например, размер кнопки и ее цвет, признак нажата или нет.

В заключение этого краткого обзора следует заметить, что именно поддержка ООП отличает язык С++ от языка С, который поддерживает только процедурный стиль программирования. В отличии от языков С# и Java, ООП-стиль программирования в языке С++ не является обязательным стилем. В своих программах вы можете использовать только процедурный стиль. И в процедурном и в ООП-стиле есть свои преимущества и недостатки. Какой стиль использовать зависит от конкретной ситуации. Но чем больше будет ваша программа, тем более очевидными будут достоинства ООП-стиля.

**Объявление класса**

Класс объявляется с помощью ключевого слова `class` по следующей схеме:

class [<Название класса>] {
    <Объявления закрытых членов класса>;
public:
    <Объявления открытых членов класса>;
private:
    <Объявления закрытых членов класса>;
protected:
    <Объявления защищенных членов класса>;
} [<Объявления переменных через запятую>];

После ключевого слова class задается название класса, которое становится новым типом данных. Название класса должно быть допустимым идентификатором, к которому предъявляются такие же требования как и к названиям переменных. Допустимо не задавать название класса, если после закрывающей фигурной скобки указано объявление переменной. Точка с запятой в конце объявления класса является обязательной.

Внутри фигурных скобок располагаются объявления членов класса — атрибутов и методов. Перед объявлениями можно указать название спецификатора доступа:

→ public — открытый. Идентификатор доступен для внешнего использования;
→ private — закрытый. Идентификатор доступен только внутри данного класса;
→ protected — защищенный. Идентификатор недоступен для внешнего использования, но доступен для данного класса и для потомков этого класса.

**Обратите внимание**

Если спецификатор доступа не указан, то идентификатор является закрытым.

Спецификаторы доступа предназначены для контроля значения атрибутов класса, а также для запрещения использования методов, которые предназначены только для внутренней реализации класса. Например, если в атрибуте предполагается хранение определенных значений, то перед присвоением значения мы можем проверить соответствие значения некоторому условию. Если же любой пользователь будет иметь возможность ввести что угодно, минуя нашу проверку, то ни о каком контроле не может быть и речи. Такая концепция сокрытия данных называется инкапсуляцией.

Объявление класса только описывает новый тип данных, а не определяет переменную, поэтому память под нее не выделяется. Чтобы объявить переменную, ее название указывается после закрывающей фигурной скобки при объявлении класса или отдельно, используя название класса в качестве типа данных:

Листинги на странице http://unicross.narod.ru/cpp/
<Название класса> <Название переменной>;
Пример одновременного объявления класса с открытыми атрибутами и переменной:
class Point {
    public:
        int x;
        int y;
    } point1;
Пример отдельного объявления переменной:
Point point2;
Обратите внимание на наличие спецификатора public. Если этот спецификатор не указан, то получить доступ к атрибутам x и y будет нельзя, так как по умолчанию члены класса являются закрытыми и доступны только внутри класса. Действие спецификатора длится до следующего спецификатора или до закрывающей фигурной скобки. В объявлении класса может быть несколько одинаковых спецификаторов. Пример:
class Point {
    public:
        int x;
    public:
        int y;
    } point1;
Определения классов можно вкладывать внутри функции или другого класса при условии, что все методы имеют реализацию внутри класса. В этом случае класс виден только внутри функции. Пример:
void func() {
    class Point {
        public:
            int x;
            int y;
        } point;
    point.x = 10;
    point.y = 40;
    std::cout << point.x << std::endl; // 10
    std::cout << point.y << std::endl; // 40
}
Внутри одного класса допускается использование объектов другого класса. В качестве примера объявим класс `Point` (точка), а затем используем его для описания координат прямоугольника внутри класса `Rect` (прямоугольник) (листинг 9.1).

**Листинг 9.1. Пример использования классов**

```cpp
#include <iostream>

class Point {
    public:
        int x;
        int y;
};

class Rect {
    public:
        Point top_left;
        Point bottom_right;
};

int main() {
    Rect rect;
    rect.top_left.x = 0;
    rect.top_left.y = 0;
    rect.bottom_right.x = 100;
    rect.bottom_right.y = 100;
    std::cout << rect.top_left.x << " "
             << rect.top_left.y << std::endl
             << rect.bottom_right.x << " "
             << rect.bottom_right.y << std::endl;
    std::cin.get();
    return 0;
}
```

**Объявление атрибутов**

Объявление атрибута внутри класса производится аналогично объявлению обычной переменной, только перед типом данных нельзя указывать спецификатор хранения, а

также производить инициализацию атрибута. Присвоить или получить значение атрибута через экземпляр класса можно с помощью точечной нотации:

```cpp
<Переменная>.<Название атрибута> = <Значение>;
```

```cpp
<Значение> = <Переменная>.<Название атрибута>;
```

Пример:

```cpp
point1.x = 10;
point1.y = 40;
std::cout << point1.x << std::endl; // 10
std::cout << point1.y << std::endl; // 40
```

В языке С++ структуры имеют функционал, аналогичный классам. Единственное отличие состоит в том, что доступ к членам структуры является открытым, а доступ к членам класса по умолчанию является закрытым. И в том и другом случае поведение по умолчанию можно изменить. Объявление структуры `point` выглядит так:

```cpp
struct Point { // Объявление структуры и переменной
    int x;
    int y;
} point;
```

Также как и в структурах внутри класса допускается использование битовых полей. Битовые поля предоставляют доступ к отдельным битам, позволяя тем самым хранить в одной переменной несколько значений, занимающих указанное количество бит.

Объявление битового поля имеет следующий формат:

```cpp
<Tип данных> [<Название поля>]:<Длина в битах>;
```

В одном классе можно использовать одновременно битовые поля и обычные атрибуты. Обратите внимание на то, что название битового поля можно не указывать, кроме того, если длина поля составляет один бит, то перед названием поля следует указать ключевое слово `unsigned`. Пример объявления класса с битовыми полями:

```cpp
class Status {
public:
    unsigned :3;
    unsigned a:1;
    unsigned b:1;
    unsigned c:1;
} status;
```

Доступ к битовому полю осуществляется также как и к атрибуту класса:

```cpp
status.a = 1;
```

status.b = 0;
status.c = 1;

Объявление и определение методов

Объявление методов внутри класса осуществляется также как и объявление обычной функции. Определение метода обычно размещают вне класса, указывая название класса перед названием метода. Между двумя названиями вставляется оператор ::. При обращении к методам класса используется следующий формат:

<Экземпляр класса>.<Имя метода>(<[Параметры]>)

Определение метода можно разместить внутри объявления класса. В этом случае содержимое метода вставляется в место вызова метода. Такие методы называются встраиваемыми. Встраиваемыми имеет смысл делать все небольшие методы. Пример объявления и определения методов приведен в листините 9.2.

```cpp
#include <iostream>

class C {
    int x_;                        // Закрытый атрибут
public:
    void set_x(int y) {x_ = y;}  // Определение встраиваемого метода
    int get_x();                  // Определение обычного метода
} obj;

int main() {  
    // Изменение значения закрытого атрибута
    obj.set_x(10);
    // Получение значения
    std::cout << obj.get_x() << std::endl; // 10
    std::cin.get();
    return 0;
}

int C::get_x() {                      // Определение обычного метода
    return x_;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
В этом примере атрибут \_x\_ является закрытым. Получить доступ к такому атрибуту можно только внутри методов класса. Обратиться к атрибуту напрямую через экземпляр класса нельзя. Чтобы обозначить, что атрибут закрытый, последним символом названия является символ подчеркивания. Некоторые программисты в этом случае указывают символ подчеркивания в самом начале названия атрибута. Для изменения и получения значения атрибута предназначены методы set\_x() и get\_x() соответственно. Внутри метода set\_x() можно контролировать значение, присваиваемое закрытому атрибуту. Таким образом соблюдается принцип сокрытия данных, называемый инкапсуляцией.

Метод set\_x() определен внутри класса. В этом случае метод является встраиваемым и его содержимое вставляется в место вызова. Объявление метода get\_x() находится внутри объявления класса, а определение — вне объявления класса. В этом случае метод будет вызываться, а не встраиваться. Перед названием метода в определении указывается названия класса и оператор :: (c::get\_x()). Методы set\_x() и get\_x() содержат только одну инструкцию, поэтому на практике оба метода имеет смысл сделать встраиваемыми.

Как видно из примера, внутри методов обращение к атрибутам (даже к закрытым) происходит как к обычным локальным переменным. Обратиться к атрибутам явным образом можно через указатель this, который неявно передается в каждый метод. Доступ к атрибуту через указатель this осуществляется с помощью оператора ->. Пример:

```cpp
void set\_x(int y) { this->\_x = y; }
```

Определение метода можно записать по-другому:

```cpp
void set\_x(int y) { (*this).\_x = y; }
```

В этом случае вместо оператора -> используется точечная нотация. Круглые скобки являются обязательными, так как приоритеты операторов -> и точка одинаковые, но выполняются они справа налево, а не наоборот. Если скобки убрать, то вначале будет сделана попытка выполнить оператор точка, что приведет к ошибке.

Некоторые методы могут получать в качестве параметра объекты другого класса. Если используются только два класса, то объявление одного класса достаточно вставить перед объявлением другого класса. Однако при увеличении количества классов возникает ситуация, в результате которой приходится решать вопрос что было раньше курица или яйцо. Чтобы избежать такой ситуации в языке C++ предусмотрено **неполное объявление класса**, имеющее следующий формат:

```cpp
class <Название класса>;
```

Листинги на странице http://unicross.narod.ru/cpp/
Пример использования неполного объявления класса приведен в листинге 9.3.

**Листинг 9.3. Неполное объявление класса**

```cpp
#include <iostream>

class A; // Неполное объявление. Если убрать, то будет ошибка

class B {
public:
    int func(A &obj); // Объявление метода
};
class A {
public:
    int x;
    A(int a) { x = a; } // Конструктор
};
int B::func(A &obj) { // Определение метода
    return obj.x;
}

int main() {
    A a(20);
    B b;
    std::cout << b.func(a) << std::endl; // 20
    std::cin.get();
    return 0;
}
```

**Конструкторы и деструктор**

Чтобы при создании экземпляра класса присвоить начальные значения каким-либо атрибутам, необходимо создать метод, имеющий такое же название как и название класса. Тип возвращаемого значения не указывается. Такой метод называется **конструктором**. Конструктор всегда автоматически вызывается при создании объекта.

Листинги на странице http://unicross.narod.ru/cpp/
Конструктор может иметь перегруженные версии, отличающиеся типом параметров или их количеством. Если внутри класса нет конструктора, то автоматически создается конструктор без параметров. В этом случае объект объявляется так:

```cpp
<Название класса> <Название переменной>;
```

или так:

```cpp
<Название класса> <Название переменной> = <Название класса>();
```

Если внутри класса объявлен пользовательский конструктор, то конструктор по умолчанию не создается. Это означает, что если вы создали конструктор с одним параметром, то при создании объекта обязательно нужно будет указывать параметр. Чтобы иметь возможность создания объекта без указания параметров следует дополнительно создать конструктор без параметров. Объявление объекта с указанием параметров выглядит следующим образом:

```cpp
<Название класса> <Название переменной>({<Параметр1>[, ..., <Параметр N>]});
```

Существует также альтернативный вариант объявления объекта:

```cpp
<Название класса> <Название переменной> =

   <Название класса>({<Параметр1>[, ..., <Параметр N>]});
```

Создание класса с несколькими конструкторами и различные способы создания экземпляра класса показаны в листинге 9.4.

### Листинг 9.4. Способы создания объектов и перегрузка конструкторов

```cpp
#include <iostream>

class C {
    int x_, y_;

public:
    C() { x_ = 0; y_ = 0; }    // Конструктор 1
    C(int x) { x_ = x; y_ = 0; }    // Конструктор 2
    C(int x, int y) { x_ = x; y_ = y; }    // Конструктор 3
    int get_x() { return x_; }
};

int main() {
    C obj1;    // Вызывается конструктор 1
    C obj2 = C();    // Вызывается конструктор 1
    C obj3(10);    // Вызывается конструктор 2
}
```

Листинги на странице http://unicross.narod.ru/cpp/
С объ4 = 20;  // Вызывается конструктор 2
С объ5 = С(30);  // Вызывается конструктор 2
С объ6(40, 30);  // Вызывается конструктор 3
С объ7 = С(50, 5); // Вызывается конструктор 3
//С объ8();    // Ошибка
std::cout << объ1.get_x() << std::endl; // 0
std::cout << объ2.get_x() << std::endl; // 0
std::cout << объ3.get_x() << std::endl; // 10
std::cout << объ4.get_x() << std::endl; // 20
std::cout << объ5.get_x() << std::endl; // 30
std::cout << объ6.get_x() << std::endl; // 40
std::cout << объ7.get_x() << std::endl; // 50
std::cin.get();
return 0;
}

Как видно из примера (см. создание объекта объ4), если конструктор принимает только один параметр, то становится возможным следующий способ создания экземпляра класса:

<Название класса> <Название переменной> = <Значение>;

В этом случае происходит ненавязное преобразование значения в объект класса. Чтобы предотвратить такое преобразование перед конструктором, принимающим один параметр, следует указать ключевое слово explicit (листинг 9.5).

### Листинги 9.5. Ключевое слово explicit

```cpp
#include <iostream>

class C {
    int x_, y_

    public:
        explicit C(int x) { x_ = x; }  y_ = 0; }
        int get_x() { return x_; }
}

int main() {
    C объ1(10);  // Нормально

    Листинги на странице http://unicross.narod.ru/cpp/
```
// c obj2 = 20; // Ошибка
std::cout << obj1.get_x() << std::endl; // 10
std::cin.get();
return 0;
}

Существует альтернативный способ присваивания начальных значений атрибутам класса, который заключается в указании списка инициализации после двоеточия между списком параметров и телом конструктора. Внутри списка инициализации указывается название атрибута после которого внутри круглых скобок задается значение. Пример использования списка инициализации приведен в листинге 9.6.

|Листинг 9.6. Использование списка инициализации|

```c++
#include <iostream>

class C {
public:
    int a, b;
    C(int x, int y): a(x), b(y) {} // Конструктор
};

int main() {
    C obj(50, 20);
    std::cout << obj.a << std::endl; // 50
    std::cout << obj.b << std::endl; // 20
    std::cin.get();
    return 0;
}
```

Если конструктор вызывается при создании объекта, то перед уничтожением объекта автоматически вызывается метод, называемый деструктором. Внутри деструктора можно закрыть ранее открытый файл, освободить динамически выделенную память и др. Название деструктора совпадает с названием класса и конструктора, но перед названием добавляется знак тильда (~). Тип возвращаемого значения не указывается. В качестве примера продемонстрируем последовательность вызова конструкторов и деструкторов при создании и удалении нескольких объектов (листинг 9.7). Чтобы увидеть результат выполнения программы следует нажать комбинацию клавиш <Ctrl>+<F5> или запустить программу из командной строки.

Листинги на странице http://unicross.narod.ru/cpp/
Листинг 9.7. Порядок вызова конструкторов и деструкторов

#include <iostream>

class C {
  int x_;  
public:
  C(int x) { // Конструктор
    x_ = x;
    std::cout << "C(int x) " << x_ << std::endl;
  }
  ~C() {    // Деструктор
    std::cout << "~C() " << x_ << std::endl;
  }
} obj1(1), obj2(2); // Создание глобальных объектов

int main() {
  C obj3(3);    // Создание локального объекта
  std::cout << "main()" << std::endl;
  return 0;
}

Результат выполнения:
C(int x) 1
C(int x) 2
C(int x) 3
main()
~C() 3
~C() 2
~C() 1

Как видно из результата конструкторы глобальных объектов (расположены после
объявления класса) вызываются до передачи управления в функцию main().
Конструкторы локальных объектов вызываются в порядке объявления объектов внутри
функции. Вызов деструкторов производится в обратном порядке.

Листинги на странице http://unicross.narod.ru/cpp/
Конструктор копирования

В случае присваивания одного объекта другому при объявлении, а также при передаче объекта в качестве параметра в функцию и возврате объекта из функции вызывается особый конструктор — *конструктор копирования*. Если конструктор копирования не определен, то используется конструктор копирования по умолчанию. В этом случае создается побитовая копия объекта. Если внутри класса производится динамическое выделение памяти, то обязательно следует определить конструктор копирования явным образом, так как создание побитовой копии в этом случае недопустимо. Конструктор копирования имеет следующий формат:

```cpp
<Nазвание класса>(const <Nазвание класса> &<Nазвание объекта>);
```

Название конструктора копирования совпадает с названием класса. В качестве параметра метод принимает ссылку на копируемый объект. Тип возвращаемого значения не указывается. Продемонстрируем использование конструктора копирования на примере (листинг 9.8).

### Листинг 9.8. Конструктор копирования

```cpp
#include <iostream>

class C {
    int x_, y_

public:
    C(int x, int y) { x_ = x; y_ = y; } // Обычный конструктор
    C(const C &c) {
        x_ = c.x_; y_ = c.y_
    }
    int get_x() { return x_; }
};

void func1(C obj) { } // Передача объекта через параметр
C func2(int x) {
    C obj(x, 40);
    return obj; // Возврат объекта из функции
}

int main() {
    C obj1(40, 30);
```
С obj2 = obj1;  // Вызывается конструктор копирования
func1(obj1);   // Вызывается конструктор копирования
obj1 = func2(60);  // Вызывается конструктор копирования
obj2 = obj1;      // Конструкторы не вызываются!!!
                     // Требуется перегрузка оператора =
                     // По умолчанию побитовая копия
std::cout << obj1.get_x() << std::endl;  // 60
std::cout << obj2.get_x() << std::endl;  // 60
std::cin.get();
return 0;
}

Обратите внимание на то, что при простом присваивании объектов (вне объявления) конструктор копирования не вызывается. В этом случае нужно перегружать оператор присваивания. По умолчанию создается побитовая копия объекта.

Статические атрибуты и методы

Внутри класса можно создать атрибут или метод, которые будут доступны без создания экземпляра класса. Для этого перед объявлением атрибута или метода следует указать ключевое слово static. Статические члены класса существуют в единственном экземпляре, независимо от количества созданных объектов.

Создание статического атрибута выполняется в несколько этапов. Вначале атрибут объявляется внутри класса, далее в глобальном пространстве производится определение, а затем в локальной области присваивается значение. Статический атрибут по умолчанию инициализируется нулевым значением. Обращение к статическому атрибуту внутри класса осуществляется также как к обычной переменной. Чтобы обратиться к статическому атрибуту вне класса следует перед его названием указать название класса и оператор ::. В качестве примера использования статических атрибутов произведен подсчет количества созданных экземпляров класса (листинг 9.9).

**Листинг 9.9. Подсчет количества созданных экземпляров класса**

```cpp
#include <iostream>

class C {
public:
    static int count; // Объявление статического атрибута

    Листинги на странице http://unicross.narod.ru/cpp/
```
C() { // Конструктор
    ++count; // Обращение к атрибуту внутри класса
}
~C() { --count; } // Деструктор
);

int C::count; // Определение статического атрибута

void func() {
    C obj1;
    std::cout << C::count << std::endl; // 3
}

int main() {
    C::count = 0; // Обращение к атрибуту вне класса
    C obj1;
    std::cout << C::count << std::endl; // 1
    C obj2;
    std::cout << C::count << std::endl; // 2
    func();
    std::cout << C::count << std::endl; // 2
    std::cin.get();
    return 0;
}

Обращение к статическому методу внутри класса осуществляется также как к обычной функции. Чтобы обратиться к статическому методу вне класса следует перед его названием указать название класса и оператор ::. Кроме того, обратиться к статическому методу можно через экземпляр класса. Обратите внимание на то, что статические методы не имеют доступа к нестатическим членам класса, так как в них не передается указатель this. Пример использования статических методов приведен в листинге 9.10.

Листинги на странице http://unicross.narod.ru/cpp/
#include <iostream>

class C {
public:
    static int sum1(int x, int y) { // Статический метод
        return x + y;
    }

    int sum2(int x, int y) { // Обычный метод
        return sum1(x, y); // Вызов статического метода внутри класса
    }
};

int main() {
    C obj;
    // Вызов статического метода вне класса
    std::cout << C::sum1(10, 20) << std::endl; // 30
    // Вызов обычного метода
    std::cout << obj.sum2(20, 5) << std::endl; // 25
    // Вызов статического метода через экземпляр класса
    std::cout << obj.sum1(30, 10) << std::endl; // 40
    std::cin.get();
    return 0;
}

Создание констант внутри класса

Создать константу внутри класса можно несколькими способами:

- объявив статическую константу. Статическая константа существует в единственном экземпляре, поэтому это лучший способ. Присваивание значения статической константе внутри класса допускается только для типов int, long, short, bool и char. Статические константы остальных типов необходимо дополнительно определять вне класса;

Листинги на странице http://unicross.narod.ru/cpp/
объявив обычную константу. Присвоить значение обычной константе допускается только в списке инициализации конструктора. Список указывается после двоеточия между списком параметров и телом конструктора. Пример:

```cpp
const int F;
const double G;
C(): F(50), G(15.3) {} // Конструктор
```

Значение обычной константы может отличаться у разных экземпляров класса, если при инициализации указана переменная, которая передается в параметре конструктора. Пример:

```cpp
const int F; // Может иметь разные значения
const double G;
C(int i): F(i), G(15.3) {} // Конструктор
```

Если значение константы должно быть одинаковым для всех экземпляров класса, то вместо обычных констант следует использовать статические константы. Это позволит более экономично использовать память и компилятор сможет произвести оптимизацию программы.

объявив константу внутри анонимного перечисления. Таким способом можно создать только целочисленную константу.

Различные варианты создания константы и способы доступа к ним вне класса показаны в листинге 9.11.

### Листинг 9.11. Создание констант внутри класса

```cpp
#include <iostream>

class C {
    public:
        // Только целочисленные значения внутри перечисления
        enum { A = 10 };
        // Инициализировать внутри класса можно только статические
        // константы типа int, long, short, bool и char.
        static const int B = 20;
        static const int D; // Определение вне класса
        static const double E; // Определение вне класса
```
// Присваивание значения константе в конструкторе
const int F;
const double G;
C(int i): F(i), G(15.3) {} // Конструктор
;
const int C::D = 30;
const double C::E = 40.3;

int main() {  
  C obj(50);
  // Обратиться можно и без создания экземпляра класса
  std::cout << C::A << std::endl;  // 10
  std::cout << obj.A << std::endl;  // 10
  std::cout << C::B << std::endl;  // 20
  std::cout << obj.B << std::endl;  // 20
  std::cout << C::D << std::endl;  // 30
  std::cout << obj.D << std::endl;  // 30
  std::cout << C::E << std::endl;  // 40.3
  std::cout << obj.E << std::endl;  // 40.3
  // Обращение только через экземпляр
  std::cout << obj.F << std::endl;  // 50
  std::cout << obj.G << std::endl;  // 15.3
  C obj2(70);
  std::cout << obj2.F << std::endl;  // 70
  std::cin.get();
  return 0;
}

Константные методы

Если внутри метода не производится изменение значений атрибутов, то его можно объяви́ть константным. Для этого после списка параметров указывается ключевое слово const. Внутри константного метода любая попытка изменить значение атрибута приведет к выводу сообщения об ошибке. Если при объявлении атрибута указано ключевое слово mutable, то внутри константного метода допускается изменение этого

Листинги на странице http://unicross.narod.ru/cpp/
атрибута. Пример использования константного метода и применение ключевого слова mutable приведен в листинге 9.12.

**Листинг 9.12. Константные методы**

```cpp
#include <iostream>

class C {
    int a, b;
    mutable int m;  // mutable
    C(int x, int y, int z) {  // Конструктор
        a = x; b = y; m = z;
    }
    void func() const {  // Константный метод
        // a = 100; b = 100;  // Ошибка
        m = 100;  // Нормально
        std::cout << a << " " << b
                   << " " << m << std::endl;
    }
};

int main() {
    C obj(50, 20, 8);
    obj.func();  // 50 20 100
    std::cin.get();
    return 0;
}
```

Если атрибут может изменять свое значение извне, то после списка параметров указывается ключевое слово volatile. Это ключевое слово предотвращает проведение оптимизации программы, при котором предполагается, что значение изменяется только в программе. Ключевое слово volatile может быть указано как отдельно, так и совместно с ключевым словом const. Пример указания:

```cpp
void func() volatile;
```

Листинги на странице http://unicross.narod.ru/cpp/
Дружественные функции и классы

Можно создать функцию, которая не является членом класса, но внутри которой будет доступ ко всем членам класса, в том числе к закрытым и защищенным. Такая функция называется дружественной. Чтобы объявить функцию дружественной следует разместить прототип функции, перед которым указано ключевое слово friend, внутри объявления класса. Одним из параметров дружественной функции должен быть экземпляр класса или ссылка на него, чтобы иметь возможность изменять значения атрибутов в переданном объекте. Пример создания дружественной функции показан в листинге 9.13.

Листинг 9.13. Дружественные функции

```cpp
#include <iostream>

class C {
    int x_; // Закрытый атрибут

public:
    explicit C(int x) { x_ = x; }
    int get_x() { return x_; }

    // Объявление дружественной функции
    friend int func(C &obj, int x);
};

    // Дружественная функция
    int func(C &obj, int x) {
        obj.x_ = x; // Можно изменить значение закрытого атрибута
        return obj.x_;
    }

    int main() {
        C obj(10);
        std::cout << func(obj, 10) << std::endl; // 20
        std::cout << obj.get_x() << std::endl; // 20
        std::cin.get();
        return 0;
    }
```

Листинги на странице http://unicross.narod.ru/cpp/
Сделать другом можно не только отдельную функцию, но и метод какого-либо класса. Если в одном классе дружественных методов несколько, то вместо объявления отдельных методов допускается объявить сразу весь класс дружественным. В этом случае все методы класса считаются дружественными. Чтобы объявить класс дружественными следует разместить неполное объявление класса, перед которым указано ключевое слово friend, внутри объявления другого класса (листинг 9.14).

Листинг 9.14. Дружественные классы

```cpp
#include <iostream>

class A {
private:
    int x_, y_;  // Закрытые атрибуты
public:
    A(int x, int y) { x_ = x; y_ = y; }
    friend class B; // Объявление дружественного класса
};
class B {
public:
    int get_x(A &obj) { return obj.x_; }
    int get_y(A &obj) { return obj.y_; }
};

int main() {
    A a(20, 50);
    B b;
    std::cout << b.get_x(a) << std::endl; // 20
    std::cout << b.get_y(a) << std::endl; // 50
    std::cin.get();
    return 0;
}
```

Массивы объектов

Объявление массива объектов ничем не отличается от объявления массива значений, имеющих элементарный тип. Отличие заключается в способе передачи параметров и

Листинги на странице http://unicross.narod.ru/cpp/
вызове конструктора. Если существует конструктор, который не принимает параметров, то массив объектов можно объявить так:

<Название класса> <Название переменной>[<Количество объектов>];

Однако при отсутствии конструктора без параметров это объявление приведет к ошибке. В этом случае необходимо явно передать значения конструктору при объявлении массива объектов.

Если конструктор принимает только один параметр, то при создании массива объектов можно передать список значений через запятую внутри фигурных скобок. Если перед объявлением конструктора стоит ключевое слово explicit, то этот способ использовать уже нельзя. Пример создания массива объектов и указания списка инициализации показан в листинге 9.15.

Листинг 9.15. Инициализация массива объектов

```cpp
#include <iostream>

class C {
    int x_; 
public:
    C(int x) { x_ = x; }
    int get_x() { return x_; }
}

int main() {
    std::cout << obj[0].get_x() << std::endl; // 10
    std::cout << obj[1].get_x() << std::endl; // 20
    std::cin.get();
    return 0;
}
```

Если конструктор принимает несколько параметров или перед объявлением конструктора, принимающего один параметр, указано ключевое слово explicit, то необходимо применять полную форму инициализации (листинг 9.16).

Листинги на странице http://unicross.narod.ru/cpp/
#include <iostream>

class C {
   int x_, y_;  
public:
   C(int x, int y) { x_ = x; y_ = y; }
   int get_x() { return x_; }
};

int main() {
   C obj[2] = { C(10, 20), C(30, 40) };
   std::cout << obj[0].get_x() << std::endl; // 10
   std::cout << obj[1].get_x() << std::endl; // 30
   std::cin.get();
   return 0;
}

Динамическое создание объектов

Для динамического создания объектов в языке C++ предназначен оператор new. Оператор new выделяет объем памяти, необходимый для хранения объекта, и возвращает его адрес. Работать в дальнейшем с объектом можно с помощью указателя. Существуют три способа динамического создания объекта:

<Указатель> = new <Название класса>;<
<Указатель> = new <Название класса>(<Значения для конструктора>);<
<Указатель> = new <Название класса> [<Количество объектов>];

В первом способе вызывается конструктор без параметров. Во втором способе внутри круглых скобок можно передать значения конструктору класса. Третий способ позволяет создать массив объектов. Обратите внимание на то, что в этом случае нельзя передать значения конструктору, поэтому необходимо предусмотреть конструктор без параметров. Примеры:

A *pA = new A; // Способ 1
B *pB = new B(20); // Способ 2
C *pC = new C [2]; // Способ 3

Листинги на странице http://unicross.narod.ru/cpp/
Чтобы обратиться к членам класса с помощью указателя при использовании способов 1 и 2 следует вместо точечной нотации применять оператор ->. В третьем способе необходимо внутри квадратных скобок указать индекс элемента, а затем с помощью точечной нотации обратиться к члену класса. Примеры:

```
pA->x = 100; // Для способов 1 и 2
pA[0].x = 100; // Для способа 3
```

Управление динамической памятью полностью лежит на плечах программиста, поэтому после завершения работы с памятью необходимо самим возвратить память операционной системе с помощью оператора `delete`. При уничтожении объекта вызывается деструктор класса. Если память выделялась для массива объектов, то после оператора `delete` следует указать квадратные скобки. В этом случае вызываются деструкторы всех объектов. Примеры:

```
delete <Указатель>; // Для способов 1 и 2
delete [] <Указатель>; // Для способа 3
```

Если память не возвратить операционной системе, то участок памяти станет недоступным для дальнейшего использования. Подобные ситуации приводят к утечке памяти. Сделать эти участки опять доступными можно только после перезагрузки компьютера. После использования оператора `delete` указатель по-прежнему будет содержать прежний адрес. Поэтому указатель принято обнулять.

При выделении памяти может возникнуть ситуация нехватки памяти. В ранних версиях C++ в этом случае возвращался нулевой указатель. Согласно новой версии стандарта в случае ошибки оператор `new` должен возбуждать исключение `bad_alloc` (объект исключения определен в файле `new`). Обработать это исключение можно с помощью конструкции `try...catch`. Пример выделения памяти с обработкой исключения:

```
#include <new>
// ... Фрагмент опущен ...
A *pA = 0; // Создаем указатель
try {
    pA = new A; // Выделяем память
}
catch (std::bad_alloc err) {
    // Обработка исключения
}
```

Выделение памяти производится внутри блока `try`. Если при этом возникнет исключение `bad_alloc`, то управление будет передано в блок `catch`. После

Листинги на странице http://unicross.narod.ru/cpp/
выполнения инструкций в блоке catch управление передается инструкции, расположенной сразу после блока. Иными словами, компилятор считает, что вы обработали исключение и можно продолжить выполнение программы. Следует учитывать, что пользоваться указателем после обработки нельзя, поэтому внутри блока catch обычно выводят сообщение об ошибке и завершают выполнение программы. Если исключение не обработать, то программа аварийно завершится. Если исключение не возникло, то инструкции внутри блока catch не выполняются.

Обратите внимание на то, что объявление указателя производится вне блока try. Если объявление разместить внутри блока, то область видимости переменной будет ограничена этим блоком. После выхода из блока переменная автоматически уничтожается, а выделенная память операционной системе не возвращается. Поэтому, объявление указателя должно находиться перед блоком, а не внутри него.

Различные способы выделения динамической памяти под объект, варианты обращения к членам класса, а также порядок вызова конструкторов и деструкторов показаны в листинге 9.17.

### Листинг 9.17. Динамическое создание объектов

```cpp
#include <iostream>
#include <new>    // Для bad_alloc

class A {
public:
    int x;
    A() { x = 0; std::cout << "A::A()" << std::endl; }
    ~A() { std::cout << "A::~A()" << std::endl; }
};
class B {
public:
    int y;
    B(int a) { y = a; std::cout << "B::B()" << std::endl; }
    ~B() { std::cout << "B::~B()" << std::endl; }
};

int main() {
    A *pA = 0;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
B *pB = 0;
try { pA = new A; }  // Способ 1
catch (std::bad_alloc err) {
    std::cout << "Error" << std::endl; return 1;
}
pA->x = 100;
std::cout << pA->x << std::endl;  // 100
delete pA;
pA = 0;
try { pB = new B(20); }  // Способ 2
catch (std::bad_alloc err) {
    std::cout << "Error" << std::endl; return 1;
}
std::cout << pB->y << std::endl;  // 20
delete pB;
pB = 0;
try { pA = new A[2]; }  // Способ 3
catch (std::bad_alloc err) {
    std::cout << "Error" << std::endl; return 1;
}
pA[0].x = 100;
pA[1].x = 200;
std::cout << pA[0].x << std::endl;  // 100
std::cout << pA[1].x << std::endl;  // 200
delete [] pA;
pA = 0;
return 0;

Указатели на объекты и члены класса

Адрес объекта можно сохранить в указателе. Объявление указателя на объект производится также как и на любой другой тип данных. Для получения адреса объекта используется оператор & а для доступа к членам вместо точки применяется оператор ->. Пример использования указателя на объект приведен в листинге 9.18.

Листинги на странице http://unicross.narod.ru/cpp/
Листинг 9.18. Использование указателя на объект

```cpp
#include <iostream>

class Point {
public:
    int x, y;
    void print();
};

void Point::print() { std::cout << x << " " << y << std::endl; }

int main() {
    Point point;     // Объявление объекта
    Point *p = &point;  // Объявление указателя на объект
    int *pY = &point.y; // Объявление указателя на атрибут
    p->x = 10;         // Доступ к атрибуту через указатель на объект
    *pY = 20;          // Доступ к атрибуту через указатель на атрибут
    p->print();       // Вызов метода через указатель на объект
    std::cin.get();
    return 0;
}
```

Как видно из примера, сохранить в указателе можно не только адрес объекта, но и адрес открытого атрибута. Помимо указателей на объекты и открытые атрибуты язык C++ позволяет создавать указатели на члены класса. Указатель на член класса хранит не адрес члена внутри объекта, а его смещение внутри класса. Объявление указателя на член класса выглядит так:

```
<Тип> <Название класса>::*<Название указателя>;
```

После объявления необходимо присвоить указателю смещение определенного атрибута. Выполняется это следующим образом:

```
<Название указателя> = &<Название класса>::<Название атрибута>;
```

Объявление указателя на мето класса выглядит так:

```
<Тип> (<Название класса>::*<Название указателя>) ([<Тип1>[, ...,<ТипN>]]);
```

Присвоить указателю смещение метода внутри класса можно так:

Листинги на странице http://unicross.narod.ru/cpp/
<Название указателя> = &<Название класса>::<Название метода>;
Для доступа к члену через объект используется оператор .*, а для доступа через указатель применяется оператор ->*. Пример использования указателей на члены класса приведен в листине 9.19.

Листинг 9.19. Указатели на члены класса

```
#include <iostream>

class Point {
    public:
        int x, y;
        void print();
    }

void Point::print() { std::cout << x << " " << y << std::endl; }

int main() {
    Point point; // Объявление объекта
    Point *p = &point; // Объявление указателя на объект

    int Point::*pX; // Объявление указателя на член
    pX = &Point::*x; // Получаем смещение атрибута x
    point.*pX = 10; // Доступ к атрибуту через объект

    int Point::*pY; // Объявление указателя на член
    pY = &Point::*y; // Получаем смещение атрибута y
    p->*pY = 20; // Доступ к атрибуту через указатель

    void (Point::*func)(); // Объявление указателя на функцию-член
    func = &Point::print; // Получаем смещение метода print()
    (point.*func)(); // Вызов метода через объект
    (p->*func)(); // Вызов метода через указатель

    std::cin.get();
    return 0;
```

Листинги на странице http://unicross.narod.ru/cpp/
Обратите внимание на круглые скобки внутри которых размещается название объекта (или название указателя на объект) и указатель на метод. Эти скобки являются обязательными. Вторые круглые скобки используются для вызова метода.

```cpp
(point.*func)(); // эквивалентно point.print();
(p->*func)();    // эквивалентно p->print();
```

Передача объектов в функцию и возврат объектов

Существует три способа передачи объекта в функцию:

- передача копии. При этом способе изменение объекта внутри функции не затрагивает исходный объект. Обратите внимание на то, что в этом случае вызываемся конструктор копирования, а не обычный конструктор. Это важно учитывать, когда внутри класса используется динамическое выделение памяти. Если конструктор копирования не определен, то выполняется побитовая копия объекта. Следует учитывать, что передача копии объекта в большинстве случаев является плохим решением, особенно если объект содержит много данных;

- передача указателя на объект. Для этого при вызове функции перед названием переменной указывается оператор & (взятие адреса), а в прототипе функции объявляется указатель. В этом случае в функцию передается не значение переменной, а ее адрес, поэтому внутри функции можно изменить исходный объект. Для доступа к членам объекта используется оператор ->;

- использование механизма ссылок. Этот способ используется наиболее часто, так как внутри функции переменная является псевдонимом исходного объекта. Любые изменения псевдонима отражаются на исходном объекте. Если внутри функции объект не должен изменяться, то его следует объявить константным. При использовании механизма ссылок перед названием переменной в объявлении и определении функции указывается оператор &, а при вызове — только название объекта. Для доступа к членам объекта используется оператор точка.

В качестве значения функция может возвращать объект. В этом случае создается временный объект, а когда результат присваивается экземпляру класса выполняется копирование, при этом вызывается конструктор копирования, а не обычный конструктор. Это важно учитывать, когда внутри класса используется динамическое выделение памяти. Пример передачи объектов в функцию и возврат объектов приведен в листинге 9.20.

Листинги на странице http://unicross.narod.ru/cpp/
Листинг 9.20. Передача объектов в функцию и возврат объектов

```cpp
#include <iostream>

class Point {
public:
    int x, y;
    void print() { std::cout << x << " " << y << std::endl; }
    Point(int a, int b) { // Обычный конструктор
        x = a; y = b;
    }
    Point(Point &obj) { // Конструктор копирования
        x = obj.x; y = obj.y;
    }
};
void func1(Point obj) { // Передача копии
    std::cout << obj.x << " " << obj.y << std::endl;
}
void func2(Point *p) { // Передача указателя
    p->x = 4; p->y = 12;
}
Point func3(Point &p) { // Механизм ссылок
    Point obj(p.x + 10, p.y + 20);
    return obj; // Возврат объекта. Вызывается
                // конструктор копирования!!
}

int main() {
    Point point(10, 20); // Объявление объекта
    func1(point); // Передача копии. Вызывается
                   // конструктор копирования!!
    func2(&point); // Передача указателя. Можно изменить
                    // значение исходного объекта
    point.print(); // 4 12
    point = func3(point); // Механизм ссылок. Можно изменить
}
```

Листинги на странице http://unicross.narod.ru/cpp/
// значение исходного объекта
point.print();    // 14 32
std::cin.get();
return 0;
}

Наследование

Наследование — это возможность создания производных классов на основе базового класса. При этом производный класс автоматически получает возможности базового класса и может добавить новую функциональность или переопределить некоторые методы. Пример создания иерархии классов приведен в листинге 9.21.

Листинг 9.21. Наследование

#include <iostream>

class A {    // Базовый класс
public:
    void func1() { std::cout << "A::func1()" << std::endl; }
};
class B: public A {    // Класс B наследует класс A
public:
    void func2() { std::cout << "B::func2()" << std::endl; }
};
class C: public B {    // Класс C наследует классы A и B
public:
    void func3() { std::cout << "C::func3()" << std::endl; }
};

int main() {
    C c;    // Создаем экземпляр класса C
c.func1();   // A::func1()
c.func2();   // B::func2()
c.func3();   // C::func3()
std::cin.get();
return 0;
}

Листинги на странице http://unicross.narod.ru/cpp/
В этом примере вначале класс `B` наследует все члены класса `A`. Затем объявляется класс `C`, который наследует все члены и класса `B` и класса `A`. Каждый класс добавляет одиннный метод. Таким образом экземпляр класса `C` имеет доступ ко всем методам классов `A` и `B`. Класс `A` называется базовым классом или суперклассом, а класс `B` — производным классом или подклассом. В то же время класс `B` является базовым для класса `C`. При наследовании используется следующий формат объявления класса:

class <Производный класс>: [ [<Спецификатор доступа>] <Базовый класс> {  
   <Определения членов класса>;
} [<Определения переменных через запятую>] ];

В параметре `<Спецификатор доступа>` можно указать следующие спецификаторы:

- **public** — открытое наследование. Все открытые и защищенные члены базового класса становятся соответственно открытыми и защищенными членами производного класса;

- **private** — закрытое наследование. Все открытые и защищенные члены базового класса становятся закрытыми членами производного класса;

- **protected** — защищенное наследование. Все открытые и защищенные члены базового класса становятся защищенными членами производного класса.

**Обратите внимание**

Закрытые члены базового класса не наследуются. Если спецификатор доступа не указан, то используется закрытое наследование.

При использовании закрытого наследования можно изменить уровень доступа некоторых открытых и защищенных членов базового класса. Сделать это можно двумя способами:

```cpp
<class> ::<class>;
using <class> ::<class>;
```

Вариант с ключевым словом `using` является более предпочтительным. Эти объявлении следует добавить в соответствующий раздел в объявлении производного класса. Пример изменения уровня доступа отдельных членов класса при закрытом наследовании показан в листинге 9.22.

Листинги на странице http://unicross.narod.ru/cpp/
Листинг 9.22. Изменение уровня доступа при закрытом наследовании

```cpp
#include <iostream>

class A {
public:
    void func1() { std::cout << "A::func1()" << std::endl; }
    void func2() { std::cout << "A::func2()" << std::endl; }
    void func3() { std::cout << "A::func3()" << std::endl; }
};
class B: private A { // Закрытое наследование
public:
    A::func2; // Открываем доступ к func2()
    using A::func3; // Открываем доступ к func3()
    void func4() { func1(); }
};

int main() {
    B obj;
    // obj.func1(); // Ошибка. Метод func1() - закрытый член класса!
    obj.func2(); // A::func2()
    obj.func3(); // A::func3()
    obj.func4(); // A::func1(). Вызов метода func1() из B::func4()
    std::cin.get();
    return 0;
}
```

При создании экземпляра производного класса в случае иерархического наследования вначале вызывается конструктор самого первого базового класса в цепочке наследования. Далее вызываются последовательно конструкторы всех остальных базовых классов и лишь затем вызывается конструктор производного класса. Деструкторы вызываются в обратном порядке. Для передачи параметров конструкторам базовых классов используется следующий синтаксис:

```
<Название конструктора производного класса>({<Параметры>}) : 
    <Название конструктора базового класса>({<Значения>})
```

// Тело конструкора

Листинги на странице http://unicross.narod.ru/cpp/
Значения конструктору базового класса передаются через список инициализации, который должен располагаться после двоеточия между списком параметров и телом конструктора. Внутри списка инициализации указывается название базового класса после которого внутри круглых скобок передаются значения. В качестве примера продемонстрируем порядок вызова конструкторов и деструкторов, а также передачу значений конструкторам базовых классов (лисинг 9.23).

Листинг 9.23. Порядок вызова конструкторов и деструкторов

```cpp
#include <iostream>

class A {
public:
    int x;
    A(int a) {        // Конструктор
        x = a; std::cout << "A::A()" << std::endl;
    }
    ~A() { std::cout << "A::~A()" << std::endl; }
};
class B: public A {
public:
    int y;
    B(int a, int b) : A(b) {           // Конструктор
        y = a; std::cout << "B::B()" << std::endl;
    }
    ~B() { std::cout << "B::~B()" << std::endl; }
};
class C: public B {
public:
    int z;
    C(int a, int b, int c) : B(b, c) {  // Конструктор
        z = a; std::cout << "C::C()" << std::endl;
    }
    ~C() { std::cout << "C::~C()" << std::endl; }
};
```

Листинги на странице http://unicross.narod.ru/cpp/
```cpp
int main() {
    C obj(10, 20, 30);
    std::cout << "A::x = " << obj.x << " B::y = " << obj.y
    << " C::z = " << obj.z << std::endl;
    return 0;
}
```

Результат выполнения программы:
A::A()
B::B()
C::C()
A::x = 30 B::y = 20 C::z = 10
C::~C()
B::~B()
A::~A()

Множественное наследование

Язык C++ поддерживает также множественное наследование. В этом случае используется следующий формат объявления производного класса:
```cpp
class <Производный класс>: [<Спецификатор доступа>] <Базовый класс1>,
    [<Спецификатор доступа>] <Базовый класс2>,
    ...,
    [<Спецификатор доступа>] <Базовый класс N>
{
    <Объявления членов класса>;
}
```

При множественном наследовании вначале вызывается конструктор базового класса, название которого расположено в списке слева. После этого вызывается конструктор базового класса, название которого расположено в списке наследования правее и т.д. Список наследования просматривается слева направо. Лишь затем вызывается конструктор производного класса. Деструкторы вызываются в обратном порядке. Для передачи параметров конструкторам базовых классов используется следующий синтаксис:
```cpp
<Название конструктора производного класса>(<Параметры>) : 
    <Название конструктора базового класса1>(<Значения>),
```

Листинги на странице http://unicross.narod.ru/cpp/
<Название конструктора базового класса > (<Значения>)
...
<Название конструктора базового класса > (<Значения>)
{
    // Тело конструктора
}

Значения конструкторам базовых классов передаются через список инициализации, внутри которого указываются названия базовых классов через запятую после которых внутри круглых скобок передаются значения. В качестве примера продемонстрируем порядок вызова конструкторов и деструкторов при множественном наследовании, а также передачу значений конструкторам базовых классов (листинг 9.24).

**Листинг 9.24. Множественное наследование**

```
#include <iostream>

class A {
    public:
        int x;
        A(int a) { // Конструктор
            x = a; std::cout << "A::A()" << std::endl;
        }
        ~A() { std::cout << "A::~A()" << std::endl; }
        void func1() { std::cout << "A::func1()" << std::endl; }
    };

class B {
    public:
        int y;
        B(int a) { // Конструктор
            y = a; std::cout << "B::B()" << std::endl;
        }
        ~B() { std::cout << "B::~B()" << std::endl; }
        void func2() { std::cout << "B::func2()" << std::endl; }
    };

class C: public A, public B { // Класс C наследует классы A и B
    public:
```

int z;
C(int a, int b, int c) : A(b), B(c) { // Передаем значения
    z = a; std::cout << "C::C()" << std::endl;
}
~C() { std::cout << "C::~C()" << std::endl; }
};

int main() {
    C obj(10, 20, 30);
    obj.func1();
    obj.func2();
    std::cout << "A::x = " << obj.x << " B::y = " << obj.y
    << " C::z = " << obj.z << std::endl;
    return 0;
}

Результат выполнения программы:
A::A()
B::B()
C::C()
A::func1()
B::func2()
A::x = 20 B::y = 30 C::z = 10
C::~C()
B::~B()
A::~A()

Если производный класс наследует несколько классов, которые в свою очередь наследуют один и тот же базовый класс, то возникает неоднозначность, так как один и тот же член базового класса будет присутствовать в нескольких классах. Рассмотрим эту ситуацию на примере (листинг 9.25).

Листинг 9.25. Неоднозначность при множественном наследовании

#include <iostream>

class A {
public:

Листинги на странице http://unicross.narod.ru/cpp/
void func1() { std::cout << "A::func1()" << std::endl; }
};

class B : public A {
public:
    void func2() { std::cout << "B::func2()" << std::endl; }
};

class C : public A {
public:
    void func3() { std::cout << "C::func3()" << std::endl; }
};

class D : public B, public C { // Класс D наследует классы B и C
public:
};

int main() {
    D obj;
    // obj.func1(); // Неоднозначность. Из B или из C?
    obj.B::func1(); // Берем из класса B
    obj.C::func1(); // Берем из класса C
    obj.func2(); // Однозначно
    obj.func3(); // Однозначно
    std::cin.get();
    return 0;
}

В этом примере классы B и C являются наследниками класса A, в котором объявлен метод func1(). Класс D наследует классы B и C. Таким образом метод func1() существует в двух экземплярах. Если создать экземпляр класса D и попытаться обратиться к методу, то компилятор выведет сообщение об ошибке. Один из способов разрешения неоднозначности является явное указание класса и оператора :: перед названием метода при вызове:

obj.B::func1(); // Берем из класса B
obj.C::func1(); // Берем из класса C

Нужны ли два одинаковых метода? Скорее всего нет. Чтобы метод был только в единственном экземпляре следует его объявить классы B и C с виртуальными. Для этого в списке наследования перед спецификатором доступа необходимо указать ключевое слово virtual. Пример:

Листинги на странице http://unicross.narod.ru/cpp/
class B : virtual public A {
public:
    void func2() { std::cout << "B::func2()" << std::endl; }
};
class C : virtual public A {
public:
    void func3() { std::cout << "C::func3()" << std::endl; }
};

В этом случае никакой неоднозначности не будет, поэтому можно вызвать метод без явного указания базового класса:
D obj;
obj.func1();

Указатели на объекты производных классов

Указатель, имеющий тип базового класса, может ссылаться на объект любого производного класса. Следует помнить, что такой указатель имеет доступ только к наследуемым членам базового класса и виртуальным методам. Чтобы получить доступ к членам производного класса нужно выполнить приведение типов (листинг 9.26). Кроме того, при использовании адресной арифметики необходимо учитывать, что указатель будет перемещен на размер базового класса, а не производного. Это может привести к ошибке.

Листинг 9.26. Указатели на объекты производных классов

```cpp
#include <iostream>

class A {
public:
    void A::func1();
};
class B : public A {
public:
    void B::func2();
};

void A::func1() { std::cout << "A::func1()" << std::endl; }
```

Листинги на странице http://unicross.narod.ru/cpp/
void B::func2() { std::cout << "B::func2()" << std::endl; }

int main() {
    A *pA;
    B b;
    pA = &b;  // Адрес производного класса
    pA->func1();  // A::func1()
    // pA->func2();  // Ошибка. Нужно выполнить приведение типов
    ((B *)&pA)->func2();  // B::func2()
    std::cin.get();
    return 0;
}

Виртуальные методы

Если имя метода в производном классе совпадает с методом из базового класса, то будет использоваться метод из производного класса. В этом случае тип и количество параметров у методов должны совпадать, в противном случае производится перегрузка метода, а не переопределение. Чтобы вызвать метод базового класса из метода производного класса, следует указать перед методом название базового класса и оператор ::. Рассмотрим переопределение методов на примере (листиング 9.27).

Листинг 9.27. Переопределение методов

#include <iostream>

class A {
public:
    void func();
};
class B : public A {
public:
    void func();
};
void A::func() { std::cout << "A::func()" << std::endl; }
void B::func() {
    std::cout << "B::func()" << std::endl;
}

Листинги на странице http://unicross.narod.ru/cpp/
// A::func(); // Вызов метода из базового класса

int main() {
    B obj;
    obj.func(); // B::func()
    std::cin.get();
    return 0;
}

В этом примере в классах A и B объявлен метод func(). Класс B наследует класс A. Следовательно метод из класса B переопределяет одноименный метод из класса A. Поэтому при создании объекта класса B будет вызван метод именно из производного класса. Таким образом достигается статический полиморфизм.

Язык C++ поддерживает также динамический полиморфизм, при котором выбор вызываемого метода зависит от объекта, на который ссылается указатель, имеющий тип базового класса. В этом случае используется позднее связывание, которое означает, что выбор метода осуществляется в процессе выполнения программы, а не на этапе компиляции. Чтобы при использовании указателя (имеющего тип базового класса) ссылающегося на объект производного класса был вызван одноименный метод из производного класса, необходимо объявить метод в базовом классе виртуальным. Для этого перед объявлением метода следует добавить ключевое слово virtual.

При наследовании дублировать ключевое слово virtual в производных классах не нужно, так как виртуальные свойства метода автоматически переносятся всем производным классам. Виртуальный метод не обязательно замещать в производном классе. Если виртуальный метод не замещен, то используется метод базового класса. Нельзя объявлять виртуальными статические методы, дружественные функции, а также конструкторы. Деструктор класса можно сделать виртуальным. Таким образом, любой метод, который может быть замещен в производном классе, следует объявить виртуальным, хотя увлекаться этим не стоит, так как при использовании виртуальных методов уменьшается эффективность работы программы. Переделаем код из листинга 9.27 и сделаем метод func() в базовом классе виртуальным (листинг 9.28).

Листинги на странице http://unicross.narod.ru/cpp/
#include <iostream>

class A {
public:
    virtual void func(); // Виртуальный метод
};
class B : public A {
public:
    void func();
};
void A::func() { std::cout << "A::func()" << std::endl; }
void B::func() { std::cout << "B::func()" << std::endl; }

void test(A &obj) { obj.func(); }

int main() {
    A *pA, a;
    B b;
    pA = &a; // Адрес базового класса
    pA->func(); // A::func()
    pA = &b; // Адрес производного класса
    pA->func(); // B::func()
    test(a); // A::func()
    test(b); // B::func()
    std::cin.get();
    return 0;
}

В этом примере дополнительно определена функция test(), которая в качестве параметра принимает ссылку на объект базового класса. Передать в функцию можно объект базового класса, а также объект любого производного класса. Выбор вызываемого внутри функции метода зависит от типа переданного объекта.

Листинги на странице http://unicross.narod.ru/cpp/
Динамическое определение типа объекта

Для динамического определения типа объекта, на который ссылается указатель, предназначен оператор typeid. Результат выполнения для полиморфных и неполиморфных объектов отличается. Прежде чем использовать оператор, необходимо подключить файл typeid. Оператор имеет следующий формат:

<Экземпляр класса type_info> = typeid(<Объект или тип>);

В качестве значения оператор typeid возвращает экземпляр класса type_info. Этот экземпляр можно сравнить с другим экземпляром того же класса, используя операторы == (равно) и != (не равно). Получить название типа объекта позволяет метод name(). Пример использования оператора typeid приведен в листинге 9.29.

Листинг 9.29. Динамическое определение типа объекта

```cpp
#include <iostream>
#include <typeinfo>

class A {
public:
    virtual void func(); // Виртуальный метод
};
class B : public A {
public:
    void func();
};

void A::func() { std::cout << "A::func()" << std::endl; }
void B::func() { std::cout << "B::func()" << std::endl; }

int main() {
    A *pA, a;
    B b;
    pA = &a;
    std::cout << typeid(*pA).name() << std::endl; // class A
    pA = &b;
    std::cout << typeid(*pA).name() << std::endl; // class B
}
```

Листинги на странице http://unicross.narod.ru/cpp/
if (typeid(*pA) == typeid(B)) {
    std::cout << "class B" << std::endl; // class B
}
std::cin.get();
return 0;

Оператор dynamic_cast

Оператор dynamic_cast выполняет приведение типов указателей или ссылок. Применяется для приведения полиморфных типов. Если приведение указателя окончилось неудачей, оператор возвращает нулевой указатель, а если проблема со ссылкой, то генерируется исключение bad_cast (объект исключения объявлен в файле typeinfo). Формат оператора:

`dynamic_cast< <Тип результата> > (<Выражение>)`

Как вы уже знаете, указатель, имеющий тип базового класса, может ссыльаться на объект любого производного класса. Однако, если указатель имеет тип производного класса, то сохранить в нем адрес, объекта базового типа, нельзя. Различные способы приведения и пример проверки корректности приведения показаны в листинге 9.30.

### Листинг 9.30. Оператор `dynamic_cast`

```cpp
#include <iostream>

class A {
public:
    virtual void func(); // Виртуальный метод
};
class B : public A {
public:
    void func();
};

void A::func() { std::cout << "A::func()" << std::endl; }
void B::func() { std::cout << "B::func()" << std::endl; }

int main() {
    // Листинги на странице http://unicross.narod.ru/cpp/
```
A *pA, a;
B *pB, b;
pA = dynamic_cast<A *> (&a); // OK
pA = dynamic_cast<A *> (&b); // OK
pB = dynamic_cast<B *> (&a); // Error
pB = dynamic_cast<B *> (&b); // OK
pA = &a; // OK
pB = dynamic_cast<B *> (pA); // Error
pA = &b; // OK
pB = dynamic_cast<B *> (pA); // OK
// pB = &a; // Error
pB = &b; // OK
pA = dynamic_cast<A *> (pB); // OK
if (pA) {
    std::cout << "OK" << std::endl;
    pA->func(); // B::func()
}
else std::cout << "Error" << std::endl;
std::cin.get();
return 0;

Абстрактные методы и классы

Абстрактные методы содержат только объявление метода без реализации. Предполагается, что производный класс должен переопределить метод и реализовать его функциональность. В языке C++ абстрактные методы реализуются с помощью чисто виртуальных методов. Синтаксис объявления абстрактного метода:

```
virtual <Тип результата> <Название метода>([, ..., <Тип> <Название параметра>]) = 0;
```

Если внутри класса существует хотя бы один абстрактный метод, то весь класс является абстрактным. Создать экземпляр абстрактного класса нельзя. Пример объявления и замещения абстрактного метода приведен в листинге 9.31.

Листинги на странице http://unicross.narod.ru/cpp/
Листинг 9.31. Абстрактные методы и классы

```c++
#include <iostream>

class A {
public:
    virtual void func() = 0; // Абстрактный метод
};
class B : public A {
public:
    void func(); // Замещаем метод
};

void B::func() { std::cout << "B::func()" << std::endl; }

int main() {
    // A a; // Создать объект абстрактного класса нельзя!
    B b;
    b.func(); // B::func()
    std::cin.get();
    return 0;
}
```

Шаблонные (обобщенные) классы

В языке C++ помимо шаблонных функций можно применять шаблонные классы, реализующие одинаковый алгоритм работы, но использующие разные типы данных, которые указываются при создании объекта. Компилятор на основе шаблонного класса автоматически создаст перегруженные версии в зависимости от имеющихся способов создания объектов в программе. Описывается шаблонный класс по следующей схеме:

template<class Тип1[, ..., class ТипN]>
class Название_класса {
    // Тело класса
};

После ключевого слова template внутри угловых скобок через запятую указываются обобщенные названия типов. Эти названия используются для описания типов, которые

Листинги на странице http://unicross.narod.ru/cpp/
могут использоваться внутри класса. При компиляции обобщенные типы будут заменены реальными типами данных. Перед названием обобщенного типа могут быть указаны ключевые слова class или typename, которые обозначают одно и то же.

Внутри объявления шаблонного класса не нужно использовать ключевое слово template для объявления шаблонного метода, так как методы шаблонного класса автоматически становятся шаблонными. Однако перед определением шаблонного метода вне класса необходимо указать ключевое слово template с такими же типами, что и в объявлении шаблонного класса. Схема определения шаблонного метода вне объявления класса выглядит так:

```cpp
template<class Typ1[, ..., class TypN]>
Typ_результата Название_класса<Typ1[, ..., TypN]>::Название_метода(
    [Typ Название_параметра1[, ..., Typ Название_параметраN]])
{
    // Тело метода
}
```

Для создания экземпляра шаблонного класса используется следующий синтаксис:

```cpp
Название_класса<Typ1[, ..., TypN]> Название_переменной;
Название_класса<Typ1[, ..., TypN]> Название_переменной(Значение1
    [, ..., Значение N]);
```

Обратите внимание на то, что при создании объекта внутри угловых скобок указываются реальные типы данных, а не обобщенные. В качестве примера создадим шаблонный класс и объявим несколько объектов с разными типами данных (листинг 9.32).

**Листинг 9.32. Шаблонные классы**

```cpp
#include <iostream>

template<class Type1, class Type2>
class C {
    Type1 x_;
    Type2 y_;

    public:
        C(Type1 x, Type2 y);
        void print();
        Type1 sum();

    Листинги на странице http://unicross.narod.ru/cpp/
```


```cpp
int main() {  
    C<int, int> a(10, 20);  
a.print();  
    std::cout << "sum = " << a.sum() << std::endl;  
    C<double, int> b(54.6, 63);  
b.print();  
    std::cout << "sum = " << b.sum() << std::endl;  
    C<double, double> c(14.8, 47.8);  
c.print();  
    std::cout << "sum = " << c.sum() << std::endl;  
    std::cin.get();  
    return 0;  
}

template<class Type1, class Type2>
C<Type1, Type2>::C(Type1 x, Type2 y) {  
    x_ = x; y_ = y;  
}

template<class Type1, class Type2>
void C<Type1, Type2>::print() {  
    std::cout << "x_ = " << x_ << " y_ = " << y_ << std::endl;  
}

template<class Type1, class Type2>
Type1 C<Type1, Type2>::sum() {  
    return x_ + y_;  
}

Шаблонные классы могут иметь перегруженные версии, в которых тип данных указывается явным образом. Объявление явной специализации выглядит так:

template<>
class Название_класса<Type1[, ..., TypeN]> {  
    // Тело класса
};

Например, в предыдущем примере, если объект создать следующим образом:

Листинги на странице http://unicross.narod.ru/cpp/
C<int, double> a(10, 12.5);
то будет выведено предупреждающее сообщение об усечении значения, так как внутри метода sum() результат сложения будет иметь тип double, а возвращаемый результат должен иметь тип int. Чтобы этого избежать можно реализовать явную специализацию для этой последовательности типов (листинг 9.33).

### Листинг 9.33. Явная специализация класса

```cpp
#include <iostream>

template<class Type1, class Type2>
class C {
   Type1 x_;
   Type2 y_;
public:
   C<Type1 x, Type2 y) { x_ = x; y_ = y; }
   Type1 sum() { return x_ + y_; }
};

template<>
class C<int, double> {
   int x_;
   double y_;
public:
   C(int x, double y) { x_ = x; y_ = y; }
   double sum() { return x_ + y_; }
};

int main() {
   C<int, double> a(10, 12.5);
   std::cout << "sum = " << a.sum() << std::endl;
   C<double, int> b(12.5, 10);
   std::cout << "sum = " << b.sum() << std::endl;
   std::cin.get();
   return 0;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
Обобщенные и реальные типы можно смешивать в объявления шаблонного класса (листинг 9.34). Если в объявления шаблонного класса указан реальный тип и название переменной, то эта переменная внутри класса является константой. При объявлении объекта этой константе необходимо передать конкретное значение. Реальный тип может быть целочисленным (int, long, short, bool и char). Использовать вещественные типы данных нельзя.

### Листинг 9.34. Смешивание обобщенных и реальных типов

```cpp
#include <iostream>

template<class Type, int y>
class C {
    Type x_;

public:
    explicit C(Type x) { x_ = x; }
    void print() {
        std::cout << "x_ = " << x_ << " y = " << y << std::endl;
    }
};

int main() {
    C<int, 20> a(10);
    a.print();
    C<double, 50> b(54.6);
    b.print();
    std::cin.get();
    return 0;
}
```

В объявлении шаблонного класса допускается указывать значения по умолчанию после оператора = для конкретизации типа или значения константы. Если при создании объекта тип или значение не указаны, то будут использоваться значения по умолчанию (листинг 9.35).

Листинги на странице http://unicross.narod.ru/cpp/
Листинг 9.35. Значения по умолчанию в объявлении шаблонного класса

```cpp
#include <iostream>

template<class Type=int, int y=100>
class C {
    Type x_; 
public:
    explicit C(Type x) { x_ = x; }
    void print() {
        std::cout << "x_ = " << x_ << " y = " << y << std::endl;
    }
};

int main() {
    C<> a(10); // Эквивалентно C<int, 100> a(10);
    a.print();
    C<double> b(12.5); // Эквивалентно C<double, 100> b(12.5);
    b.print();
    C<double, 5> c(2.4); // Конкретные значения
    c.print();
    std::cin.get();
    return 0;
}
```

Шаблонные классы можно наследовать, как и обычные классы. При использовании указателей важно помнить, что указатель на объект одной специализации нельзя присвоить указателю у которого специализация отличается. Иными словами, указателю на объект типа в<int> нельзя присвоить адрес объекта типа в<double>. Пример наследования шаблонных классов показан в листинге 9.36.

Листинг 9.36. Наследование шаблонных классов

```cpp
#include <iostream>

template<class Type> class A {
public:
```

Листинги на странице http://unicross.narod.ru/cpp/
Type x;
A<Type a> { x = a; }
virtual void func();
};
// Класс B наследует класс A
template<class Type> class B : public A<Type> {
public:
    Type y;
    B<Type a, Type b> : A<Type>(b) { y = a; }
    void func();
};

template<class Type>
void A<Type>::func() { std::cout << "A::func()" << std::endl; }
template<class Type>
void B<Type>::func() { std::cout << "B::func()" << std::endl; }

int main() {
    A<int> *pA, a(30);
    B<int> b(10, 20);
    B<double> c(10.5, 20.3);
    pA = &a;
    pA->func(); // A::func()
    pA = &b;
    pA->func(); // B::func()
    std::cout << typeid(*pA).name() << std::endl; // B<int>
    std::cout << typeid(c).name() << std::endl; // B<double>
    std::cin.get();
    return 0;
}
Глава 10.
Перегрузка операторов
После создания класса его название становится новым типом данных. Тем самым пользовательские классы расширяют возможности языка C++. С помощью перегрузки операторов можно еще больше расширить возможности объектов, так как перегрузка операторов позволяет экземплярам классов участвовать в обычных выражениях. Например, перегрузив оператор + можно сложить объект с другим объектом или любым элементарным типом данных.

Способы перегрузки операторов

Перегрузка операторов производится с помощью методов, имеющих специальные названия. Определён в классе "операторный" метод можно изменить поведение оператора по своему вкусу. Например, перегрузив оператор + можно вместо сложения производить вычитание. Однако пользователи класса вряд ли положительно оценят такую инициативу. Тем не менее в некоторых случаях изменение смысла оператора довольно полезно. Например, класс cout перегружает оператор побитового сдвига << для вывода данных в окно консоли.

"Операторный" метод имеет следующий формат:

<typename> operator<название оператора>(
    [<тип> <название параметра1>[, ..., <тип> <название параметраN>]]
) {
    // Тело метода
}

Название "операторного" метода состоит из слова "operator", после которого указывается перегружаемый оператор, например, operator+, operator= и т.д. Внутри круглых скобок указывается тип, объект которого может быть расположен в выражении справа от перегружаемого оператора. Следовательно для каждого типа, с которым может взаимодействовать экземпляр класса внутри выражения, необходимо создать отдельный "операторный" метод. Например, чтобы сложить объект с целым числом следует создать метод operator+(int x), а для сложения с вещественным числом — метод operator+(double x). Внутри метода доступен указатель this на объект, который расположен слева от оператора. Таким образом, внутри метода можно напрямую обращаться к атрибутам и методам класса. Пример перегрузки оператора + для сложения объекта с целым числом и возвратом нового объекта:

```cpp
C C::operator+(int x) {       // Объект класса C + целое число
    C newObj;               // Создаем временный объект
    // Обработка
    return newObj;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
newobj.x_ = x_ + x; // x_ - закрытый член класса C
return newobj; // Возвращаем временный объект
}
Метод будет вызван в следующей ситуации:
obj2 = obj1 + 20; // с obj1, obj2;
Однако, если будет следующая ситуация, то метод вызван не будет, так как объект должен быть расположен слева от оператора, а не справа:
obj2 = 20 + obj1; // Метод C::operator+(int x) не вызывается!
Перегрузить оператор + в этом случае позволяют дружественные функции. Параметры дружественной функции передаются явным образом. Объект, расположенный слева от оператора, доступен через первый параметр, а объект, расположенный справа от оператора, доступен через второй параметр. Объявление дружественной функции внутри объявления класса выглядит так:
friend C operator+(int x, C &obj);
Пример определения дружественной функции вне класса:
C operator+(int x, C &obj) { // Целое число + объект класса C
    C newobj;
    newobj.x_ = obj.x_ + x; // Обращение к x_ через параметр
    return newobj;
}
С помощью дружественной функции можно также заменить "операторный" метод. Тем не менее, стоит отдать предпочтение "операторным" методам, а дружественные функции применять в случаях, когда обойтись методом нельзя. В качестве примера перегрузим оператор + для сложения объекта с целым числом:
C operator+(C &obj, int x) { // Объект класса C + целое число
    C newobj;
    newobj.x_ = obj.x_ + x;
    return newobj;
}
При перегрузке операторов следует учитывать, что:
- с помощью дружественных функций нельзя перегрузить операторы =, (), -> и[]. Для перегрузки этих операторов нужно использовать "операторные" методы;
- вообще нельзя перегрузить операторы . (точка), ::, ?: и .*;
- нельзя изменить приоритет операторов;
Листинги на странице http://unicross.narod.ru/cpp/
нельзя изменить количество параметров в "операторном" методе или дружественной "операторной" функции;

нельзя использовать значения по умолчанию в параметрах. Исключением является перегрузка оператора {};

нельзя добавить новые операторы.

Перегрузка бинарных операторов

При использовании бинарных операторов в выражении участвуют два операнда. Примеры бинарных операторов: + (плюс), - (минус), * (умножение), / (деление), % (остаток от деления), & (двоичное И), | (двоичное ИЛИ), ^ (двоичное исключающее ИЛИ), << (сдвиг влево), >> (сдвиг вправо) и оператор "запятая". К бинарным операторам также относятся операторы сравнения: == (равно), != (неравно), < (меньше), > (больше), <= (меньше или равно), >= (больше или равно), && (логическое И) и || (логическое ИЛИ).

При перегрузке бинарных операторов через единственный параметр в "операторном" методе доступен operand, расположенный от оператора справа, а указатель на сам объект, расположенный от оператора слева, передается нейвным образом. Таким образом внутри метода ко всем членам класса можно обращаться напрямую или через указатель this. Дружественной функции передаются два параметра. Через первый параметр доступен operand, расположенный от оператора слева, а через второй параметр доступен operand, расположенный от оператора справа. Тип возвращаемого значения, а также внутренняя реализация зависит от предпочтений программиста. Обратите внимание на то, что в обычных выражениях при использовании бинарных операторов возвращается новое значение, а значения operandов внутри выражения не изменяются. Хотя это не правило, а лишь рекомендация. В качестве примера перегрузим операторы + и == (листинг 10.1).

```
Листинг 10.1. Перегрузка бинарных операторов

#include <iostream>

class C {
  int x_;
public:
  C() { x_ = 0; } // Конструктор по умолчанию

Листинги на странице http://unicross.narod.ru/cpp/
```
explicit C(int x) { x_ = x; } // Обычный конструктор
C(const C &c) { x_ = c.x_; } // Конструктор копирования
int get_x() { return x_; }
C operator+(const C &obj); // obj1 + obj2
C operator+(int x); // obj + целое число
friend C operator+(int x, const C &obj); // целое число + obj
bool operator==(const C &obj); // obj1 == obj2
bool operator==(int x); // obj == целое число
friend bool operator==(int x, const C &obj); // целое число == obj
};

int main() {
    C obj1(10), obj2(20), obj3;
    obj3 = obj1 + obj2;
    std::cout << obj3.get_x() << std::endl; // 30
    obj3 = obj3 + 50;
    std::cout << obj3.get_x() << std::endl; // 80
    obj2 = 20 + obj3;
    std::cout << obj2.get_x() << std::endl; // 100
    std::cout << (obj1 == obj2) << std::endl; // 0
    std::cout << (obj1 == 10) << std::endl; // 1
    std::cout << (100 == obj2) << std::endl; // 1
    std::cin.get();
    return 0;
}

C C::operator+(const C &obj) { // obj1 + obj2
    C newobj;
    newobj.x_ = x_ + obj.x_;
    return newobj;
}

C C::operator+(int x) { // obj + целое число
    C newobj;
    newobj.x_ = x_ + x;
    return newobj;
}

Листинги на странице http://unicross.narod.ru/cpp/
Перегрузка унарных операторов

При использовании унарных операторов в выражении участвует только один operand — сам объект. Примеры унарных операторов: + (унарный плюс), - (унарный минус), * (разыменование), & (взятие адреса), ~ (двоичная инверсия) и ! (логическое отрицание).

При перегрузке унарных операторов "опператорный" метод не принимает параметров. Внутри метода ко всем членам объекта можно обращаться напрямую или через указатель this. Дружественной функции передается один параметр — объект класса. В качестве примера перегрузим унарный минус с помощью "операторного" метода, а оператор ! с помощью дружественной функции (листинг 10.2).

Листинг 10.2. Перегрузка унарных операторов

```cpp
#include <iostream>

class C {
    int x_; 

public:
    C() { x_ = 0; } // Конструктор по умолчанию
    explicit C(int x) { x_ = x; } // Обычный конструктор
    C(const C &c) { x_ = c.x_; } // Конструктор копирования

    operator-() { return -x_; } // Унарный минус
    operator!() { return !x_; } // Логическое отрицание
};
```
```cpp
int get_x() { return x_; }
C operator-(); // -obj
friend bool operator!(const C &obj); // !obj

int main() {
  C obj1(10), obj2;
  obj2 = -obj1;
  std::cout << obj2.get_x() << std::endl; // -10
  std::cout << (!obj1) << std::endl; // 0
  std::cin.get();
  return 0;
}

C C::operator-() { // -obj
  C newobj;
  newobj.x_ = -x_;
  return newobj;
}
bool operator!(const C &obj) { // !obj
  return !(obj.x_);
}
```

Перегрузка операторов инкремента и декремента

Операторы инкремента (++) и декремента (--) также являются унарными операторами, но они изменяют сам объект. Кроме того, операторы могут использоваться в постфиксной или префиксной формах. При постфиксной форме (x++) возвращается значение переменной перед операцией, а при префиксной форме (++x) — вначале производится операция и только потом возвращается значение.

Прототипы "операторных" методов:

- `<Название класса> &operator++();` // ++obj
- `<Название класса> &operator--();` // --obj
- `<Название класса> operator++(int);` // obj++
- `<Название класса> operator--(int);` // obj--

Прототипы дружественных функций:

Конечно, если вы хотите больше информации, вы можете посетить страницу http://unicross.narod.ru/cpp/
friend <Название класса> &operator++(<Название класса> &);       // ++obj
friend <Название класса> &operator--(<Название класса> &); // --obj
friend <Название класса> operator++(<Название класса> &); // obj++
friend <Название класса> operator--(<Название класса> &); // obj--

Целочисленный параметр, используемый в прототипах при постфиксной форме, можно пропустить. По умолчанию этот параметр имеет значение 0. Пример перегрузки оператора инкремента приведен в листинге 10.3.

### Листинг 10.3. Перегрузка оператора инкремента

```cpp
#include <iostream>

class C {
  int x_;

public:
  C() { x_ = 0; }    // Конструктор по умолчанию
  explicit C(int x) { x_ = x; } // Обычный конструктор
  C(const C &c) { x_ = c.x_; }  // Конструктор копирования
  int get_x() { return x_; }
  C &operator++();          // ++obj
  C operator++(int x);     // obj++
};

int main() {
  C obj1(10), obj2;
  obj2 = ++obj1;
  std::cout << obj1.get_x() << std::endl; // 11
  std::cout << obj2.get_x() << std::endl; // 11
  obj2 = obj1++;
  std::cout << obj1.get_x() << std::endl; // 12
  std::cout << obj2.get_x() << std::endl; // 11
  ++obj2;
  std::cout << obj1.get_x() << std::endl; // 12
  std::cout << obj2.get_x() << std::endl; // 12
  std::cin.get();
  return 0;
}```

C &C::operator++() { // ++obj
    ++x_;  
    return *this;
}
C C::operator++(int x) { // obj++
    C tmp = *this;
    ++*this;
    return tmp;
}

Перегрузка операторов присваивания

Перегрузка оператора = и сокращенных операторов присваивания (например, +=, -=,
*=, /= и др.) осуществляется также как и перегрузка бинарных операторов. Главное
отличие заключается в том, что операторы присваивания изменяют сам объект, а не
возвращают новый. При использовании "операторных" методов ссылка на объект,
расположенный слева от оператора присваивания, передается неявным образом, а
объект, расположенный справа от оператора, доступен через параметр. Дружественной
функции передаются два параметра. Через первый параметр доступен объект,
расположенный от оператора слева, а через второй параметр доступен объект,
расположенный от оператора справа. В качестве результата обычно возвращается
указатель на исходный объект.

При перегрузке оператора = следует учитывать:

➔ если оператор не перегружен, то автоматически создается оператор присваивания по
умолчанию, который создает побитовую копию объекта. Это важно учитывать, если
внутри класса производится динамическое выделение памяти;
➔ при использовании оператора присваивания при инициализации объекта
вызывается конструктор копирования, а не перегруженная версия оператора =;
➔ перегрузить оператор = с помощью дружественной функции нельзя.

Пример перегрузки операторов присваивания приведен в листинге 10.4.

Листинги на странице http://unicross.narod.ru/cpp/
Листинг 10.4. Перегрузка операторов присваивания

```cpp
#include <iostream>

class C {
    int x_

public:
    C() { x_ = 0; } // Конструктор по умолчанию
    explicit C(int x) { x_ = x; } // Обычный конструктор
    C(const C &c) { x_ = c.x_; } // Конструктор копирования
    int get_x() { return x_; }
    C &operator=(C &obj); // obj2 = obj1
    C &operator=(int x); // obj = целое число
    C &operator+=(int x); // obj += целое число
};

int main() {
    C obj1, obj2;
    obj1 = 10; // obj = целое число
    obj2 = obj1; // obj2 = obj1
    obj1 = obj1; // obj = obj
    obj2 += 20; // obj += целое число
    std::cout << obj1.get_x() << std::endl; // 10
    std::cout << obj2.get_x() << std::endl; // 30
    // Множественное присваивание
    obj2 = obj1 = 5;
    std::cout << obj1.get_x() << std::endl; // 5
    std::cout << obj2.get_x() << std::endl; // 5
    // Вызывается конструктор копирования,
    // а не метод C &operator=(C &obj)!!!
    C obj3 = obj1;
    std::cout << obj3.get_x() << std::endl; // 5
    std::cin.get();
    return 0;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
C &C::operator=(C &obj) {  
    if (this == &obj) return *this;  // случай obj = obj
    x_ = obj.x_;  
    return *this;
}

C &C::operator=(int x) {  // obj = целое число
    x_ = x;  
    return *this;
}

C &C::operator+=(int x) {  // obj += целое число
    x_ += x;  
    return *this;
}

Перегрузка оператора ()

Перегрузка оператора () позволяет обработать вызов экземпляра класса как вызов функции. Количество параметров и тип возвращаемого значения в "операторном" методе могут быть произвольными. Внутри метода ко всем членам объекта можно обращаться напрямую или через указатель this. Обратите внимание на то, что перегрузить оператор () с помощью дружественной функции нельзя. Пример перегрузки оператора () приведен в листинге 10.5.

Листинг 10.5. Перегрузка оператора ()

```cpp
#include <iostream>

class C {
    int x_;  

public:
    C() { x_ = 0; }  // Конструктор по умолчанию
    explicit C(int x) { x_ = x; }  // Обычный конструктор
    int get_x() { return x_; }  
    void operator[](int x) { x_ = x; }
    int operator[](int) { return x_; }
};
```

Листинги на странице http://unicross.narod.ru/cpp/
int main() {
    C obj;
    obj(10);
    std::cout << obj() << std::endl;   // 10
    std::cout << (20 + obj()) << std::endl; // 30
    std::cin.get();
    return 0;
}

Перегрузка оператора [ ]

Перегрузка оператора [ ] позволяет обработать доступ к элементу по индексу, причем индекс может быть не только целочисленным. При перегрузке оператора [ ] через единственный параметр в "операторном" методе доступен индекс, расположенный внутри квадратных скобок. Внутри метода ко всем членам класса можно обращаться напрямую или через указатель this. Чтобы можно было присвоить значение элементу, расположенному по указанному индексу, необходимо внутри метода возвратить ссылку на элемент. Обратите внимание на то, что перегрузить оператор [ ] с помощью дружественной функции нельзя. Пример перегрузки оператора [ ] приведен в листинге 10.6.

<table>
<thead>
<tr>
<th>Листинг 10.6. Перегрузка оператора [ ]</th>
</tr>
</thead>
</table>

```cpp
#include <iostream>
#include <cstdlib>

class C {
    static const int size_ = 2;
    int arr_[size_];

public:
    C() { arr_[0] = 0; arr_[1] = 0; }
    C(int x, int y) { arr_[0] = x; arr_[1] = y; }
    int &operator[](int i);
};

int &C::operator[](int i) {
    if (i < 0 || i >= size_) {  // Проверка выхода за границы массива
        return arr_[0];
    }
    return arr_[i];
}
```

Листинги на странице http://unicross.narod.ru/cpp/
std::cout << "Error" << std::endl;
std::exit(1);
}
return arr_[i];
}

int main() {
    C obj1, obj2(30, 40);
    obj1[0] = 10;
    obj1[1] = 20;
    std::cout << obj1[0] << std::endl; // 10
    std::cout << (obj1[1] + obj2[0]) << std::endl; // 50
    std::cin.get();
    return 0;
}

Перегрузка оператора доступа к члену класса

Перегрузка оператора -> существует также как и перегрузка унарных операторов. В качестве значения "операторный" метод должен возвращать указатель this. Обратите внимание на то, что перегрузить оператор -> с помощью дружественной функции нельзя. Пример перегрузки оператора -> приведен в листинге 10.7.

Листинг 10.7. Перегрузка оператора ->

#include <iostream>

class C {
    int x_;
public:
    int y;
    C(int a, int b) { x_ = a; y = b; }
    C *operator->() { return this; }
};

int main() {
    C obj(30, 40);
    Листинги на странице http://unicross.narod.ru/cpp/
Перегрузка операторов new и delete

Операторы new и delete, предназначенные для динамического управления памятью, также можно перегрузить. Допускается перегружать все форматы операторов. Наиболее часто перегружаются следующие форматы операторов:

\[ <Указатель> = \text{new} <\text{Название класса}>(<\text{Начальное значение}>); \]
\[ \text{delete} <\text{Указатель}>; \]
\[ <\text{Указатель}> = \text{new} <\text{Название класса}>[<\text{Количество элементов}>]; \]
\[ \text{delete} [ ] <\text{Указатель}>; \]

Чтобы перегрузить эти форматы операторов необходимо создать "операторные" методы, имеющие следующие прототипы:

\[ \text{void \*operator new(size\_t count);} \]
\[ \text{void operator delete(void \*object);} \]
\[ \text{void \*operator new[](size\_t count);} \]
\[ \text{void operator delete[](void \*object);} \]

Внутри методов new и new[] для выделения памяти обычно используется функция malloc(), а внутри методов delete и delete[] для освобождения памяти — функция free(). Количество байтов, необходимых для хранения одного или нескольких объектов, доступно через параметр count. При неудачном выделении памяти функция malloc() вернет нулевой указатель. В этом случае внутри методов new и new[] следует сгенерировать исключение, являющееся объектом класса bad_alloc (определён в файле new). Генерация исключения выглядит так:

\[ \text{std::bad\_alloc err;} \]
\[ \text{throw err; } \]

При использовании следующих форматов операторов необходимо не генерировать исключение, а возвращать нулевой указатель:

\[ <\text{Указатель}> = \text{new (std::nothrow)} <\text{Название класса}>(<\text{Начальное значение}>); \]
\[ <\text{Указатель}> = \text{new (std::nothrow)} <\text{Название класса}>[<\text{Количество элементов}>]; \]

Листинги на странице http://unicross.narod.ru/cpp/
Чтобы перегрузить эти форматы операторов необходимо создать "операторные" методы, имеющие следующие прототипы:

```cpp
void *operator new(size_t count, const std::nothrow_t &);
void operator delete(void *object, const std::nothrow_t &);
void *operator new[](size_t count, const std::nothrow_t &);
void operator delete[](void *object, const std::nothrow_t &);
```

Обратите внимание на то, что в этом случае при перегрузке операторов delete и delete[] необходимо указать во втором параметре const std::nothrow_t &. Параметр nothrow_t внутри методов можно пропустить. Дополнительно следует перегрузить обычные операторы delete и delete[] с одним параметром.

Пример перегрузки операторов new[] и delete[] с обработкой ошибок, а также последовательность вызова "операторных" методов, конструкторов и деструкторов класса показаны в листинге 10.8.

```
#include <iostream>
#include <cstdlib>
#include <new>

class C {
   int x_

public:
   C() { x_ = 0; std::cout << "C()" << std::endl; }
   ~C() { std::cout << "~C()" << std::endl; }
   void set_x(int x) { x_ = x; }
   int get_x() { return x_; }
   void *operator new[](size_t count);
   void operator delete[](void *p);
};

void *C::operator new[](size_t count) {
   std::cout << "new[]" << std::endl;
   void *p = std::malloc(count); // Выделяем память
   if (!p) { // Проверяем на корректность
      std::bad_alloc err; // Создаем объект исключения
   }
```

Листинги на странице http://unicross.narod.ru/cpp/
throw err;  // Генерируем исключение
}
return p;  // Возвращаем указатель
}

void C::operator delete[](void *p) {
    std::cout << "delete[]" << std::endl;
    std::free(p);  // Освобождаем память
}

int main() {
C *obj = 0;
try {
    obj = new C[2];  // Выделяем память для двух объектов
}
catch (std::bad_alloc err) {
    std::cout << "Error" << std::endl;
    std::exit(1);  // Выходим при ошибке
}

obj[0].set_x(10);  // Пользуемся памятью
obj[1].set_x(20);
std::cout << obj[0].get_x() << std::endl;
std::cout << obj[1].get_x() << std::endl;
delete [] obj;  // Освобождаем память
obj = 0;  // Обнуляем указатель
std::cin.get();
return 0;
}

Результат выполнения:
new[]
C()
C()
10
20
~C()
~C()
delete []
Как видно из результата, при успешном выделении памяти автоматически вызываются конструкторы. При вызове оператора `delete[]` вначале вызываются деструкторы всех объектов, а затем управление передается перегруженному методу `operator delete[]`(). Не забывайте о том, что если для выделения памяти используется оператор `new[]`, то при освобождении памяти следует вызывать оператор `delete[]`, а не `delete`. И, наоборот, если для выделения памяти используется оператор `new`, то при освобождении памяти следует вызывать оператор `delete`, а не `delete[]`.

Вместо `std::nothrow` внутри круглых скобок при вызове операторов `new` и `new[]` можно указать несколько значений через запятую:

```
<Указатель> = new (<Структурное_значение>) (<Название_класса>)
<Указатель> = new (<Структурное_значение>)
    (<Название_класса>)<Количество_элементов>;
```

Чтобы перегрузить эти форматы операторов необходимо создать "операторные" методы, имеющие следующие прототипы:

```cpp
void *operator new(size_t count, <Тип> <Параметр1>
    [, ..., <Тип> <Параметр N>]);
void operator delete(void *object, <Тип> <Параметр1>
    [, ..., <Тип> <Параметр N>]);
void *operator new[](size_t count, <Тип> <Параметр1>
    [, ..., <Тип> <Параметр N>]);
void operator delete[](void *object, <Тип> <Параметр1>
    [, ..., <Тип> <Параметр N>]);
```

Тип и количество дополнительных параметров может быть произвольным. Обратите внимание на то, что операторы выделения памяти и операторы освобождения памяти отличаются только первым параметром. Помимо этих методов дополнительно следует создать перегруженные версии методов для обычных операторов `delete` и `delete[]` с одним параметром. Например, если определены следующие методы:

```cpp
void *operator new(size_t count, int x, double y);
void operator delete(void *object);
void operator delete(void *object, int x, double y);
```

то выделить и освободить память можно так:

```cpp
C *obj = new(10, 25.8) c;     // Выделяем память
// ...
delete obj;                  // Освобождаем память
```

Листинги на странице http://unicross.narod.ru/cpp/
Значения 10 и 25.8, указанные внутри круглых скобок при выделении памяти, будут доступны в методе operator new() через параметры int x и double y соответственно.

Если внутри класса перегружены операторы new и delete, то при выделении и освобождении памяти под объекты этого класса будут вызываться "операторные" методы этого класса. При выделении памяти под другие объекты вызываются их перегруженные операторы или глобальные операторы при отсутствии перегруженных версий. Чтобы отказаться от перегруженных операторов и обратиться к глобальному оператору следует написать так:

```c++
C *obj = ::new C;  // Обращение к глобальному оператору new
// ...
::delete obj;     // Обращение к глобальному оператору delete
```

В этом примере перед ключевыми словами new и delete расположен оператор разрешения области видимости ::, поэтому вместо перегруженных версий будут вызваны глобальные операторы new и delete.

Перегружаются операторы new и delete допускается не только для отдельного класса, но и глобально для всех объектов, включая встроенные типы данных. Для этого следует создать глобальные "операторные" функции без привязки к конкретному классу. В этом случае стандартные операторы игнорируются и вызываются перегруженные версии. Если в каком-либо классе операторы new и delete перегружены, то вызываются именно перегруженные версии, а не глобальные.

### Перегрузка операторов `<<` и `>>`

В языке C++ операторы `<<` и `>>` перегружены для вывода и ввода данных соответственно. Чтобы объекты пользовательского класса имели возможность взаимодействовать с потоками ввода/вывода следует перегрузить эти операторы с помощью дружественных функций. Использовать в этом случае "операторные" методы нельзя. Перегрузка оператора `<<` осуществляется следующим образом:

```cpp
std::ostream &operator<<(std::ostream &stream, <Класс> &obj) {
     // Выводим данные в поток используя переменную stream
     return stream;               // Возвращаем ссылку на поток
}
```

Перегрузка оператора `>>` выглядит так:

```cpp
std::istream &operator>>(std::istream &stream, <Класс> &obj) {

Листинги на странице http://unicross.narod.ru/cpp/```
// Вводим данные из потока используя переменную stream
return stream; // Возвращаем ссылку на поток
}

В качестве параметров функции принимают ссылку на соответствующий поток и ссылку на объект. Внутри функции необходимо вернуть ссылку на поток, чтобы иметь возможность составлять цепочки из вызовов операторов. Пример перегрузки операторов «<<» и «»» показан в листинге 10.9.

Листинг 10.9. Перегрузка операторов «<<» и «>>»

```cpp
#include <iostream>

class C {
    int x_;  

public:
    int y;
    C(int a, int b) { x_ = a; y = b; }
    friend std::ostream &operator<<(std::ostream &stream, C &obj);
    friend std::istream &operator>>(std::istream &stream, C &obj);
};

std::ostream &operator<<(std::ostream &stream, C &obj) {
    stream << "Class C { x_ = " << obj.x_ << ", y = "
           << obj.y << "}";
    return stream; // Возвращаем ссылку на поток
}

std::istream &operator>>(std::istream &stream, C &obj) {
    std::cout << "y = ";
    stream >> obj.y;
    return stream; // Возвращаем ссылку на поток
}

int main() {
    C obj(10, 20);
    std::cin >> obj; // Например, ввели число 500
    std::cout << obj << std::endl; // Результат:
                              // Class C { x_ = 10, y = 500 }

    Листинги на странице http://unicross.narod.ru/cpp/
```
std::cin.ignore(255, '\n').get();
return 0;
}

Преобразование объекта в другой тип данных

Помимо операторов, язык C++ позволяет перегружать операцию приведения типов. Для этого используется следующий формат "операторного" метода:
operator <Тип данных>()
    return <Значение, точно соответствующее типу>;
}

Название типа данных, в который производится приведение типа, задается после ключевого слова operator. Внутри метода необходимо вернуть значение, точно соответствующее этому типу. Метод не имеет типа возвращаемого значения и параметров.

Если внутри класса определен один "операторный" метод перегрузки приведения, то при указании объекта внутри выражения производится автоматическое преобразование объекта в другой тип, используя этот метод. Например, если существует перегрузка операции приведения к типу int, то объект можно преобразовать и в тип double. Однако, если существуют несколько методов, то компилятор уже не сможет выбрать метод. Поэтому перед объектом следует указать в какой тип данных необходимо его преобразовать с помощью операторов приведения типов. В качестве примера определим внутри класса три метода, позволяющих преобразовать объект в типы int, double и char * (листинг 10.10).

Листинг 10.10. Перегрузка операции приведения типов

#include <iostream>
#include <stdio.h>

class C {
    int x_;
public:
    char buf[15];
    C(int x) { x_ = x; }
    operator int() { return x_; }
    operator double() { return static_cast<double>{x_}; }

Листинги на странице http://unicross.narod.ru/cpp/
operator char*() {
    sprintf_s(buf, 15, "%d", x_);
    return buf;
}

int main() {
    C obj(10);
    int x = static_cast<int>(obj) + 47;
    std::cout << x << std::endl; // 57
    x = (int)obj + 12;
    std::cout << x << std::endl; // 22
    double y = static_cast<double>(obj) + 22.5;
    std::cout << y << std::endl; // 32.5
    std::cout << static_cast<char*>(obj) << std::endl; // 10
    std::cin.get();
    return 0;
}
Глава 11.
Обработка ошибок

© Прохоренок Н.А., 2010 г., unicross@mail.ru
Этот DjVu-файл предоставляется КАК ЕСТЬ. Автор не несет никакой ответственности за прямые или косвенные проблемы, связанные с использованием данного файла. ВЫ ИСПОЛЬЗУЕТЕ ЕГО НА СВОЙ СТРАХ И РИСК.
Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Запрещено:
- перепечатывать всю книгу или отдельные главы без письменного разрешения Автора;
- декомпилировать данный файл и преобразовывать его в любой другой формат.
Если вы когда-нибудь учились водить автомобиль, то наверняка вспомните, что при первой посадке на водительское сиденье все внимание было приковано к трем деталям: рулю, педалям и рычагу переключения передач. Происходящее вне автомобиля уходило на второй план, так как вначале нужно было стронуться с места. По мере практики навыки вождения улучшались и эти три детали постепенно уходили на задний план. Как ни странно, но руль и рычаг переключения передач всегда оказывались там, куда вы не смотря протягивали руки, а ноги сами находили педали. Теперь все внимание стало занимать происходящее на дороге. Иными словами, вы стали профессионалом.

В программировании все абсолютно также. Начинающие программисты больше обращают внимание на первые попавшиеся на глаза операторы, функции и другие элементы языка, а сам алгоритм уходит на задний план. Если программа скомпилировалась без ошибок, то это уже большое счастье, хотя это еще не означает, что программа работает правильно. По мере практики мышление программиста меняется, он начинает обращать внимание на мелочи, на форматирование программы, использует более эффективные алгоритмы и в результате всего этого допускает меньше ошибок. Подводя итоги, можно сказать, что начинающий программист просто пишет программу, а опытный программист пытается найти оптимальный алгоритм и предусмотреть поведение программы в различных ситуациях. Однако, от ошибок никто не застрахован, поэтому очень важно знать как быстро найти ошибку.

**Типы ошибок**

Существуют три типа ошибок в программе:

- **синтаксические** — это ошибки в имени оператора или функции, отсутствие закрывающей или открывающей кавычек и т. д., то есть ошибки в синтаксисе языка.

Как правило, компилятор предупредит о наличии ошибки, а программа не будет выполняться совсем. Пример синтаксической ошибки:

```cpp
std::cout << "Нет завершающей кавычки!;"
std::cin.get();
```

При компиляции будут выведены следующие сообщения об ошибках:

```
main.cpp(6): error C2001: newline в константе
main.cpp(7): error C2143: синтаксическая ошибка: отсутствие ";"
перед "std::cin"
```

Цифра внутри круглых скобок, расположенных после названия компиляруемого файла, обозначает номер строки, в которой обнаружена ошибка. Если после попытки скомпилировать программу в окне Вывод (открыть окно можно из меню Вид, выбрав пункт Вывод) выполнить двойной щелчок мышью на строке с описанием ошибки, то слева от строки с ошибкой внутри программы появится маркер и строка станет активной. Кроме того, обратите внимание на то, что строка с синтаксической ошибкой подсвечивается редактором красной волнистой линией еще до компиляции программы. После слова error указывается номер ошибки. Чтобы получить дополнительную информацию об ошибке с таким кодом следует запустить локальную документацию (в меню Справка выбираем пункт Просмотр справки) и в строку поиска ввести номер ошибки. Если локальная документация не установлена, то ее можно установить выбрав в меню Справка пункт Управление параметрами справки или воспользоваться документацией в Интернете на сайте http://msdn.microsoft.com/;

→ **логические** — это ошибки в логике работы программы, которые можно выявить только по результатам работы программы. Как правило, компилятор не предупреждает о наличии ошибки, а программа будет выполняться, так как не содержит синтаксических ошибок. Такие ошибки достаточно трудно выявить. Основные ошибки в языке C++ связаны с указателями и массивами, так как компилятор не производит никакой проверки корректности указателя и не контролирует выход за границы массива. Весь контроль полностью лежит на плечах программиста;

→ **ошибки времени выполнения** — это ошибки, которые возникают во время работы программы. В одних случаях ошибки времени выполнения являются следствием логических ошибок, а в других случаях причиной являются внешние события, например, нехватка оперативной памяти, отсутствие прав для записи в файл и др.

**Операторы try...catch и throw**

Некоторые операторы стандартной библиотеки в случае ошибки генерируют исключения. Например, исключение может генерироваться оператором new, при невозможности выделении динамической памяти. Если в исходном коде не предусмотрена обработка исключения, то программа аварийно прерывается. Обработать исключение в программе позволяет оператор try...catch. Формат оператора:

```
try {
    <Блок, в котором перехватывается исключение>
}
```

Листинги на странице http://unicross.narod.ru/cpp/
catch (<Тип исключения1> <Переменная>) {
    <Блок, выполняемый при возникновении исключения>
}

...
catch (<Тип исключенияN> <Переменная>) {
    <Блок, выполняемый при возникновении исключения>
}
catch (...) {
    <Блок, который выполняется, если предыдущие блоки catch
    не соответствуют типу исключения>
}

Инструкции, в которых перехвачиваются исключения, должны быть расположены внутри блока try. Если при выполнении этих инструкций возникнет исключение, то управление будет передано в блок catch, который соответствует типу исключения. Типом исключения может выступать встроенный тип или пользовательский класс. Обратите внимание на то, что если исключение не возникло, то инструкции внутри блока catch не выполняются. В качестве примера сгенерируем исключение типа int и обработаем его:

try {
    throw 10;       // Генерируем исключение типа int
    std::cout << "Эта инструкция не будет выполнена!!!";
}
catch (int x) {   // Обработка исключения типа int
    std::cout << "x = " << x << std::endl; // x = 10
}

В этом примере исключение искусственно генерируется с помощью оператора throw. После оператора указывается объект исключения. Тип этого объекта становится типом исключения, а само значение доступно через переменную, заданную в операторе catch. При возникновении исключения внутри блока try управление сразу передается в соответствующий блок catch. Таким образом, код, расположенный после инструкции, сгенерированной исключение, выполнен не будет. После выполнения инструкций в блоке catch управление передается инструкции, расположенной сразу после оператора try...catch. Иными словами, считается, что исключение обработано и можно продолжить выполнение программы.

Листинги на странице http://unicross.narod.ru/cpp/
В некоторых случаях необходимо не продолжить выполнение программы, а прервать ее выполнение. Например, при нехватке памяти не имеет смысла продолжать работу. В этом случае можно внутри блока \texttt{catch} произвести завершающие действия (закрыть файл, освободить динамическую память и т. д.), а затем прервать работу программы с помощью функции \texttt{exit()} или \texttt{abort()}. Однако, в больших программах внутри блока \texttt{catch} не всегда имеется доступ ко всем указателям, хранящим адреса динамически выделенной памяти, так как область видимости переменных ограничена блоком. Если прервать выполнение программы, то эта динамическая память освобождена не будет. Чтобы сообщить остальным частям программы об исключении, следует повторно генерировать исключение внутри блока \texttt{catch}, указав оператор \texttt{throw} без параметра. Пример:

\begin{verbatim}
try {
    try {
        throw 10;   // Генерируем исключение типа int
    }
    catch (int x) { // Обработка исключения типа int
        std::cout << "x = " << x << std::endl; // x = 10
        throw; // Повторно генерируем исключение
    }
    std::cout << "Эта инструкция не будет выполнена!!!";
}
catch (int x) {
    std::cout << "x = " << x << std::endl; // x = 10
}
\end{verbatim}

Как видно из примера, один обработчик исключения можно вложить в другой. При генерации исключения внутри вложенного блока \texttt{try} вначале управление передается во вложенный блок \texttt{catch}. Внутри этого блока выводится сообщение, а затем исключение генерируется повторно с тем же самым типом и значением. В этом случае исключение "всплывает" к обработчику более высокого уровня и управление передается внешнему блоку \texttt{catch}, минуя остальную часть программы. Если в коде нет других обработчиков, то программа аварийно прерывается.

Внутри оператора \texttt{try...catch} можно указывать несколько блоков \texttt{catch} с разными типами исключений. При возникновении исключения производится проверка типа в первом блоке \texttt{catch}. Если тип исключения соответствует, то управление передается в этот блок, а остальные блоки \texttt{catch} игнорируются. Если тип не соответствует, то производится проверка типа в следующем блоке \texttt{catch} и т. д. Если ни один блок \texttt{catch}
не соответствует исключению, то исключение "всплывает" к обработчику более высокого уровня. При отсутствии обработчика программа аварийно завершается. Пример использования нескольких блоков catch:

try {
    throw 10.5; // Генерируем исключение типа double
}
catch (int x) {
    // Этот блок пропускается
    std::cout << "int x = " << x << std::endl;
}
catch (double x) {
    // Управление передается этому блоку
    std::cout << "double x = " << x << std::endl; // double x = 10.5
}
catch (char ch) {
    // Этот блок игнорируется
    std::cout << "char ch = " << ch << std::endl;
}

Если в блоке catch внутри круглых скобок указаны три точки вместо типа, то такой блок перехватывает все исключения. Обычно блок по умолчанию размещают после всех остальных блоков catch. Пример:

try {
    throw 'A'; // Генерируем исключение типа char
}
catch (int x) {
    // Этот блок пропускается
    std::cout << "int x = " << x << std::endl;
}
catch (...) {
    // Управление передается этому блоку
    std::cout << "catch (...)" << std::endl; // catch (...)
}
catch (double x) {
    // Этот блок никогда не будет выполнен!!!
}

Листинги на странице http://unicross.narod.ru/cpp/
**Примечание**
Не все ошибки можно перехватить с помощью оператора `try...catch`. Например, перехватить ошибку деления на ноль нельзя.

### Класс exception

В заголовочном файле `exception` объявлен класс `exception`, который наследует некоторые стандартные классы исключений, например, `bad_exception`, `bad_alloc` и др. Класс имеет четыре конструктора:

```cpp
exception();
explicit exception(const char * const &);
exception(const char * const &, int);
exception(const exception&);
```

Если параметр в конструкторе не задан, то текст сообщения выглядит так: "Unknown exception". С помощью второго и третьего конструктора можно изменить текст сообщения. Получить сообщение об ошибке позволяет метод `what()`. Прототип метода:

```cpp
virtual const char *what() const;
```

В качестве примера сгенерируем исключение класса `exception` и обработаем его с помощью оператора `try...catch` (листинг 11.1).

**Листинг 11.1. Класс exception**

```cpp
#include <iostream>
#include <exception>

int main() { 
    try {
        std::exception err("my_error"); // Создаем объект
        throw err; // Генерируем исключение
    }
    catch (std::exception &err) {
        std::cout << err.what() << std::endl; // my_error
    }
    std::exception err1;
    std::cout << err1.what() << std::endl; // Unknown exception
    std::exception err2("my_error");

    Листинги на странице http://unicross.narod.ru/cpp/
```
std::cout << err2.what() << std::endl;    // my_error
std::cin.get();
return 0;
}

Пользовательские классы исключений

Как вы уже знаете, типом исключения может выступать встроенный тип или пользовательский класс. Основное преимущество использования классов для обработки исключений заключается в возможности указания базового класса для перехвата всех исключений соответствующих классов-потомков. Например, если пользовательский класс наследует стандартный класс exception, то, указав в блоке catch объект класса exception, можно перехватить исключение пользовательского класса. Обратите внимание на то, что блок catch, в котором указан объект производного класса, должен быть расположен перед блоком catch, в котором указан объект базового класса.

При разработке библиотек обычно создается базовый класс, являющийся наследником класса exception, а все остальные классы исключений внутри библиотеки наследуют этот базовый класс (листинг 11.2). Таким образом, разработчик библиотеки получает возможность создания новых классов исключений, создавая иерархию классов. Пользователю в этом случае достаточно указать объект базового класса в блоке catch, чтобы перехватить все исключения, генерируемые внутри библиотеки.

Листинг 11.2. Создание иерархии пользовательских классов исключений

```
#include <iostream>
#include <exception>

// Базовый класс
class MyExceptionBase : public std::exception {
public:
    MyExceptionBase() : exception("Error") { }
    MyExceptionBase(const char *message) : exception(message)
        { }
    virtual ~MyExceptionBase() { }
};
// Производные классы от MyExceptionBase
class MyException1 : public MyExceptionBase {

Листинги на странице http://unicross.narod.ru/cpp/
public:
    MyException1() : MyExceptionBase("Error") { }
    MyException1(const char *message) : MyExceptionBase(message)
    { }
    virtual ~MyException1() { }
};
class MyException2 : public MyExceptionBase {
public:
    MyException2() : MyExceptionBase("Error") { }
    MyException2(const char *message) : MyExceptionBase(message)
    { }
    virtual ~MyException2() { }
};

int main() {
    try {
        MyException2 err("my_error"); // Создаем объект
        throw err; // Генерируем исключение
    }
    catch (MyException1 &err) {
        // Производный класс исключения должен обрабатываться
        // раньше, чем базовый класс!!!
        std::cout << "MyException2: " << err.what() << std::endl;
    }
    catch (MyExceptionBase &err) {
        std::cout << "MyExceptionBase: " << err.what() << std::endl;
    } // Результат: MyExceptionBase: my_error
    std::cin.get();
    return 0;
}
Ограничение типа исключений, генерируемых внутри функции

В объявлении функции после параметров можно указать спецификацию, ограничивающую типы исключений, которые допускается генерировать внутри функции. Спецификация имеет следующий синтаксис:

```
<Tип> <Название функции>([[<Параметры>]]) throw([[<Типы исключений>]]) {
    // Тело функции
}
```

Внутри инструкции `throw()` могут быть указаны допустимые типы исключений через запятую. Если типы исключений не заданы, то функция не должна генерировать исключения вообще. Если вместо типов указаны три точки, то функция может сгенерировать любое исключение. Пример:

```c
void func() throw();    // Функция не должна генерировать исключение
void func() throw(...); // Функция может генерировать любое исключение
void func() throw(int, double); // Допустимы только типы int и double
```

Обратите внимание на то, что в VC++ спецификации исключений являются допустимыми, но не реализованы. Попытка указать внутри круглых скобок конкретные типы приводит к выводу предупреждающего сообщения с номером C4290. Чтобы предотвратить вывод предупреждающего сообщения следует вначале программы вставить следующую инструкцию:

```
#pragma warning( disable: 4290 )
```

Назначение обработчиков верхнего уровня

Если не существует ни одного блока `catch`, соответствующего типу исключения, то вызывается функция `terminate()`. Прототип функции `terminate()`:

```
#include <eh.h>
void terminate(void);
```

По умолчанию эта функция завершает выполнение программы, вызывая функцию `abort()`. С помощью функции `set_terminate()` можно назначить функцию, которая будет вызываться вместо `terminate()`. Прототип функции:

```
#include <eh.h>
terminate_function set_terminate(terminate_function NewPtFunc);
typedef void (*terminate_function)();
```

Листинги на странице http://unicross.narod.ru/cpp/
Через параметр функция принимает указатель на функцию, которая будет вызвана при наступлении исключения верхнего уровня. Внутри этой функции можно произвести завершающие действия, а затем обязательно необходимо прервать выполнение программы. Функция не должна возвращать управление обратно в программу. В качестве значения функция set_terminate() возвращает указатель на прежний обработчик. Пример использования функции приведен в листинге 11.3.

```cpp
#include <iostream>
#include <cstdlib>
#include <eh.h>

void func() {
    std::cout << "func()" << std::endl;
    std::abort(); // Прерываем выполнение программы
}

int main() {
    set_terminate(func); // Назначаем обработчик
    try {
        throw 10;
    } catch (double err) { // Тип не соответствует!
        std::cout << "double err " << err << std::endl;
    }
    std::cin.get();
    return 0;
}
```

Помимо функции terminate() существует еще функция unexpected(), которая вызывается, когда внутри функции генерируется исключение не указанное внутри инструкции throw() в объявлении функции. Обратите внимание на то, что в VC++ 2010 функция в этом случае не вызывается. Прототип функции unexpected():
```cpp
#include <eh.h>
void unexpected(void);
```

Листинги на странице http://unicross.narod.ru/cpp/
Внутри функции `unexpected()` производится вызов функции `terminate()`. Назначить свой обработчик можно с помощью функции `set_unexpected()`. Прототип функции:
```c
#include <eh.h>
unexpected_function set_unexpected(unexpected_function NewPtFunc);
typedef void (*unexpected_function)();
```
Через параметр функция принимает указатель на функцию, которая будет вызвана при наступлении исключения. Внутри этой функции можно произвести завершающие действия, а затем обязательно необходимо прервать выполнение программы. Функция не должна возвращать управление обратно в программу. В качестве значения функция `set_unexpected()` возвращает указатель на прежний обработчик.

### Функция `strerror()`

Некоторые функции из стандартной библиотеки не генерируют исключение, а возвращают некоторое значение и присваивают переменной `errno` номер ошибки. Чтобы сбросить флаг ошибки следует присвоить значение 0 этой переменной. Например, функция `strtol()`, предназначенная для преобразования C-строки в число типа `long`, в случае ошибки присваивает переменной `errno` значение макроса `ERANGE`. Макрос `ERANGE` и остальные макросы с номерами ошибок определены в заголовочном файле `errno.h`. Получить текстовое описание ошибки по ее коду позволяет функция `strerror()`. Прототип функции:
```c
#include <cstring>
char *strerror(int n);
```
Пример вывода сообщения:
```c
cout << strerror(ERANGE) << endl; // Result too large
cout << strerror(22) << endl; // Invalid argument
```
В VC++ при использовании функции `strerror()` выводится предупреждающее сообщение "warning C4996: 'strerror': This function or variable may be unsafe". Чтобы избежать этого сообщения следует вместо функции `strerror()` использовать функцию `strerror_s()`. Прототип функции:
```c
#include <cstring>
errno_t strerror_s(char *Buf, size_t SizeInBytes, int ErrNum);
```
В первом параметре функция принимает указатель на строку, а во втором параметре максимальный размер этой строки. Номер ошибки задается в третьем параметре. Пример использования функции `strerror_s()` приведен в листинге 11.4.

Листинг на странице http://unicross.narod.ru/cpp/
Листинг 11.4. Использование функции strerror_s()

```cpp
#include <iostream>
#include <cstring>
#include <errno.h>

int main() {  
    char err[100] = {0};
    strerror_s(err, 100, 42);
    std::cout << err << std::endl; // Illegal byte sequence
    strerror_s(err, 100, ERANGE);
    std::cout << err << std::endl; // Result too large
    std::cin.get();
    return 0;
}
```

Отключение вывода предупреждающих сообщений

При использовании некоторых стандартных функций в VC++ выводятся предупреждающие сообщения, которые информируют, что функции могут стать причиной различных (иногда очень опасных) ошибок. Например, функция `strncpy()` не производит никакой проверки при копировании символов, поэтому возможна ситуация переполнения буфера. Для каждой такой функции в VC++ существует безопасная замена. Название безопасной функции отличается наличием фрагмента "_s" в конце названия.

Использование безопасных функций решает проблему с переполнением буфера, но делает программу несовместимой с другими компиляторами. Одним из способов решения проблемы несовместимости является перегрузка стандартных функций в безопасные. В этом случае внутри программы указываются стандартные функции, а в самом начале программы добавляются инструкции:

```cpp
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES 1
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES_COUNT 1
```

Если вы уверены в своих действиях, то можно отключить вывод предупреждающих сообщений. Сделать это можно двумя способами:

- определить макрос с названием `_CRT_SECURE_NO_WARNINGS` в самом начале программы, перед подключением заголовочных файлов. Пример:

   Листинги на странице http://unicross.narod.ru/cpp/
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>

вставить программу warning. Внутри фигурных скобок указывается ключевое слово disable, после которого задается номер отключаемого сообщения. Пример:
#pragma warning( disable : 4996 )

Обратите внимание
После директив #define и #pragma точка с запятой не указывается.

Способы поиска ошибок в программе

В предыдущих разделах мы научились обрабатывать ошибки времени выполнения. Однако, наибольшее количество времени программист затрачивает на другой тип ошибок — логические ошибки. В этом случае программа компилируется без ошибок, но результат выполнения программы не соответствует ожидаемому результату. Ситуация еще более осложняется, когда неверный результат проявляется лишь периодически, а не постоянно. Инсценировать такую же ситуацию, чтобы получить этот же неверный результат, бывает крайне сложно и занимает очень много времени. В этом разделе мы рассмотрим лишь "дедовские" (но по прежнему актуальные) способы поиска ошибок, а современные способы отладки приложений, доступные в Microsoft Visual C++ 2010 Express, изучим в следующем разделе.

Первое, на что следует обратить внимание, — это форматирование кода. Начинаяющие программисты обычно не обращают на это никакого внимания, считая этот процесс лишним. А на самом деле зря! Компилятору абсолютно все равно, разместите вы все инструкции на одной строке или выполните форматирование кода. Однако, при поиске ошибок форматирование кода позволит найти ошибку гораздо быстрее.

Перед всеми инструкциями внутри блока должно быть расположено одинаковое количество пробелов. Обычно используется три или четыре пробела. От применения символов табуляции лучше отказаться. Если все же используете, то не следует в одном файле совмещать и пробелы и табуляцию. Для вложенных блоков количество пробелов уменьшают на уровень вложенности. Если для блока первого уровня вложенности использовалось три пробела, то для блока второго уровня вложенности должно использоваться шесть пробелов, для третьего уровня — девять пробелов и т. д. Пример форматирования вложенных блоков приведен в листинге 11.5.

Листинги на странице http://unicross.narod.ru/cpp/
Листинг 11.5. Пример форматирования вложенных блоков

```cpp
const short ARR_ROWS = 2, ARR_COLS = 4;
int arr[][ARR_COLS] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8}
};

int i, j;
for (i=0; i<ARR_ROWS; ++i) {
    for (j=0; j<ARR_COLS; ++j) {
        std::cout.width(3);
        std::cout << arr[i][j];
    }
    std::cout << std::endl;
}
```

Открывающая фигурная скобка может быть расположена как на одной строке с оператором, так и на следующей строке. Какой способ использовать зависит от предпочтений программиста или от требований по оформлению кода, принятых внутри фирмы. Пример размещения открывающей фигурной скобки на отдельной строке:

```cpp
for (i=0; i<ARR_ROWS; ++i) {
    for (j=0; j<ARR_COLS; ++j) {
        std::cout.width(3);
        std::cout << arr[i][j];
    }
    std::cout << std::endl;
}
```

Длина одной строки не должна содержать более 80 символов. Если количество символов больше, то следует выполнить переход на новую строку. При этом продолжение смещается относительно основной инструкции на величину отступа или выравнивается по какому-либо элементу. Иначе приходится пользоваться горизонтальной полосой прокрутки, а это очень неудобно при поиске ошибок.

Если программа слишком большая, то следует задуматься о разделении программы на отдельные функции или классы, которые выполняют логически законченные действия. Помните, что отлаживать отдельную функцию гораздо легче, чем "спагетти"-код.

Листинги на странице http://unicross.narod.ru/cpp/
Причем прежде чем вставить функцию (или класс) в основную программу ее следует протестировать в отдельном проекте, передавая функции различные значения и проверяя результат ее выполнения.

Обратите внимание на то, что форматирование кода должно выполняться при написании кода, а не во время поиска ошибок. Этим вы сократите время поиска ошибки и скорее всего заметите ошибку еще на этапе написания. Если все же ошибка возникла, то вначале следует инсценировать ситуацию, при которой ошибка проявляется. После этого можно начать поиск ошибки.

Причиной периодических ошибок чаще всего являются внешние данные. Например, если числа получаются от пользователя, а затем производится деление чисел, то вполне возможна ситуация, при которой пользователь введет число 0. Деление на ноль приведет к ошибке. Следовательно, все данные, которые поступают от пользователей, должны проверяться на соответствие допустимым значениям. Если данные не соответствуют, то нужно вывести сообщение об ошибке, а затем повторно запросить новое число или прервать выполнение всей программы. Кроме того, нужно обработать возможность того, что пользователь может ввести вовсе не число, а строку. Пример получения числа от пользователя с проверкой корректности данных показан в листинге 11.6.

### Листинг 11.6. Пример получения числа от пользователя с проверкой данных

```cpp
int x = 10, y;
for (;;) {  // Бесконечный цикл
    std::cout << "y = ";  // Выводим подсказку
    std::cin >> y;  // Получаем значение
    if (!std::cin.good()) {  // Проверка успешности операции
        std::cout << std::endl << "Error" << std::endl;
        std::cin.clear();  // Сбрасываем флаг ошибки
        std::cin.ignore(255, '\n');  // Очищаем буфер
    } else if (y == 0) {  // Проверка допустимости значения
        std::cout << "Значение 0 недопустимо" << std::endl;
    } else break;  // Если ошибок нет, то выходим из цикла
}
std::cout << x / y << std::endl;  // Выводим результат
```

Объект `cout` удобно использовать для вывода промежуточных значений. В этом случае значения переменных вначале выводятся в самом начале программы (внутри функции `main()`) и производится проверка соответствия значений. Если значения соответствуют, то инструкция с объектом `cout` перемещается на следующую строку программы и опять производится проверка и т. д. Если значения не совпали, то ошибка возникает в инструкции, расположенной перед инструкцией с объектом `cout`. Если это пользовательская функция, то проверку значений производят внутри функции, каждый раз перемещая инструкцию с выводом значений. На одном из этих многочисленных этапов ошибка обычно обнаруживается. В больших программах можно логически догадаться о примерном расположении инструкции с ошибкой и начать поиск ошибки оттуда, а не с самого начала программы.

Инструкции для вывода промежуточных значений можно расставить уже при написании программы, не дожидаясь возникновения ошибки. В этом случае в начале программы объявляется макрос, а внутри программы производится проверка наличия макрока. Если макрос определен, то выводятся значения. В противном случае инструкция просто игнорируется. Создать макрос позволяет директива `#define`:

```cpp
#define MY_DEBUG 1
```

Проверить существование макрока позволяет следующая конструкция:

```cpp
#ifdef MY_DEBUG
  // Макрос определен. Здесь размещаем инструкции вывода значений
#endif
```

Проверить значение макрока можно так:

```cpp
#if MY_DEBUG == 1
  // Макрос имеет значение 1
#endif
```

Вместо определения собственного макрока можно воспользоваться уже существующим макросом `_DEBUG`. Этот макрос определен, если используется режим `debug`. Если используется режим `Release`, то макрос определен не будет. В качестве примера создадим несколько макротов (листинг 11.7).

### Листинг 11.7. Использование макрсов

```cpp
#include <iostream>

#ifdef _DEBUG
    #define MY_DEBUG1 1
```

Листинги на странице http://unicross.narod.ru/cpp/
#endif
#define MY_DEBUG2 1

int main() {
    int x = 10;
    #ifdef MY_DEBUG1
        // Эта инструкция будет выполнена только при отладке
        std::cout << "x = " << x << std::endl;
    #endif
    #ifdef MY_DEBUG2 == 1
        // Эта инструкция будет выполнена только если макрос
        // MY_DEBUG2 равен 1
        std::cout << "x = " << x << std::endl;
    #endif
    std::cin.get();
    return 0;
}

В этом примере макрос MY_DEBUG1 будет определен только в режиме debug. Вместо него можно напрямую использовать макрос _DEBUG, но определение нескольких собственных макросов, существующих только в режиме Debug, позволит выводить те или иные значения, а не все сразу. Чтобы прекратить вывод промежуточных значений достаточно закомментировать определение конкретного макроса. Кроме того, в начале скрипта определен макрос MY_DEBUG2. Этот макрос будет определен в любых режимах. Если макрос имеет значение 1, то будет производиться вывод промежуточных значений. Чтобы отключить вывод следует изменить значение макроса в начале программы.

Для проверки условия можно также воспользоваться макрофункцией assert(), которая имеет следующий прототип:
#include <cassert>
assert(<Выражение>);

Если выражение возвращает значение 0, то выводится сообщение об ошибке с указанием названия файла и номера строки, а затем выполнение программы прерывается с помощью функции abort(). В противном случае никаких действий не производится. В качестве выражения обычно используется логическое выражение. Макрофункция срабатывает только в режиме debug, в режиме Release она

Листинги на странице http://unicross.narod.ru/cpp/
игнорируется. Пример использования макрофункции `assert()` приведен в листинге 11.8.

Листинг 11.8. Использование макрофункции `assert()`

```cpp
#include <iostream>
#include <cassert>

int main() {
    int x = 10;
    assert(x >= 0); // Ошибка не генерируется
    std::cout << "x = " << x << std::endl;
    assert(x < 0); // Генерируется ошибка в режиме Debug
    std::cin.get();
    return 0;
}
```

Результат в режиме Debug выглядит так:

```
x = 10
Assertion failed: x < 0, file c:\book\test\test\main.cpp, line 8
```

При поиске ошибок часто приходится переходить от кода внутри функции `main()` к определению какой-либо функции в том же самом файле. Постоянно пользоваться в этом случае вертикальной полосой прокрутки очень неудобно. Одним из способов решения проблемы в VC++ является использование закладок. Чтобы пометить строку следует сделать ее активной, а затем из меню **Правка** выбрать пункт **Закладки** | **Закладка** или нажать соответствующую кнопку на панели инструментов. Слева от строки появится специальный маркер. Чтобы удалить закладку следует повторно выбрать тот же пункт меню. Перемещаться между закладками можно с помощью клавиши `<F2>` или с помощью соответствующих кнопок на панели инструментов. Ещё одним способом решения проблемы является разделение окна на две части. Для этого в меню **Окно** следует выбрать пункт **Разделить**. Чтобы вернуть окно в первоначальное состояние следует в меню **Окно** выбрать пункт **Снять разделение**.

Отладка программы в Microsoft Visual C++ 2010 Express

Сделать поиск ошибок более эффективным позволяет отладчик, встроенный в Microsoft Visual C++ 2010 Express. С его помощью можно выполнять программу по шагам, при

этом контролируя значения переменных на каждом шагу. Отладчик позволяет также проверить, соответствует ли порядок выполнения инструкций разработанному ранее алгоритму.

Прежде чем начать отладку необходимо пометить строки внутри программы с помощью **точек останова**. Для добавления точки останова делаем строку активной, а затем из контекстного меню выбираем пункт **Точка останова | Вставить точку останова**. Слева от строки появится красный кружок, обозначающий точку останова. Добавить точку останова можно еще быстрее. Для этого достаточно щелкнуть слева от строки левой кнопкой мыши. Повторный щелчок позволяет удалить точку останова. Кроме того, для добавления или удаления точки отстанова можно воспользоваться клавишей **<F9>**. Чтобы изменить свойства точки, отключить или удалить ее следует щелкнуть на ней правой кнопкой мыши и в открывшемся контекстном меню выбрать соответствующий пункт.

Когда точки останова расставлены можно начать отладку. Для этого из меню **Отладка** выбираем пункт **Начать отладку** или нажимаем клавишу **<F5>**. При достижении точки останова выполнение программы прерывается и отладчик ожидает дальнейших действий программиста. Инструкция, которая будет выполняться на следующем шаге, помечается желтой стрелкой слева от строки.

В режиме прерывания можно посмотреть значения различных переменных в окнах **Видимые** и **Локальные**. Если окна не отображаются, то отобразить их можно выбрав в меню **Отладка** пункт **Окна**, а затем соответствующий пункт. Посмотреть значение переменной можно также если навести указатель мыши на переменную. Значение переменной отобразится во всплывающей подсказке.

При отладке можно контролировать значения отдельных переменных, а не всех сразу. Для этого следует добавить название переменной в окне **Контрольные значения 1** в поле **Имя**. Чтобы отобразить окно из меню **Отладка** выбираем пункт **Окна | Контрольные значения | Контрольные значения 1**. Можно также выделить название переменной и из контекстного меню выбрать пункт **Добавить контрольное значение**.

Для пошагового выполнения программы предназначены следующие пункты в меню **Отладка** или соответствующие кнопки на панели инструментов **Отладка**:

- **Продолжить** (клавиша **<F5>**): выполняет инструкции до следующей точки останова или до конца программы при отсутствии точек;
- **Шаг с заходом** (клавиша **<F11>**): выполняет одну инструкцию. Если в этой инструкции производится вызов функции, то производится заход внутрь функции;

Шаг с обходом (клавиша <F10>) — выполняет одну инструкцию. Если в этой инструкции производится вызов функции, то функция выполняется и отладчик переходит в режим ожидания после выхода из функции;

Шаг с выходом (комбинация клавиш <Shift>+<F11>) — при заходе в функцию этот пункт позволяет выполнить функцию и выйти из нее. Отладчик переходит в режим ожидания после выхода из функции;

Перезапустить (комбинация клавиш <Ctrl>+<Shift>+<F5>) — начинает отладку заново;

Остановить отладку (комбинация клавиш <Shift>+<F5>) — останавливает отладку.

Кроме того, можно произвести выполнение программы до текущей позиции. Для этого в режиме отладки щелкаем правой кнопкой мыши на строке и из контекстного меню выбираем пункт Выполнить до текущей позиции или делаем строку активной и нажимаем комбинацию клавиш <Ctrl>+<F10>.

Чтобы сделать все точки останов неактивными следует из меню Отладка выбрать пункт Выключить все точки останова. В этом случае точки временно отключаются, но не удаляются. В отключенном состоянии отображается лишь контур кружка без заливки. Чтобы опять сделать все точки активными следует из меню Отладка выбрать пункт Включить все точки останова. Для удаления точек предназначен пункт Удалить все точки останова в меню Отладка.

Управлять отдельными точками останова или всеми сразу можно в окне Точки останова. Чтобы отобразить окно следует из меню Отладка выбрать пункт Окна | Точки останова или нажать комбинацию клавиш <Alt>+<F9>. В этом окне можно изменить свойства точки останова. Например, чтобы указать, что точка должна срабатывать только на N-ый раз следует щелкнуть на точке правой кнопкой мыши и из контекстного меню выбрать пункт Число попаданий. В открывшемся окне из списка выбираем нужный пункт, вводим число в текстовое поле, а затем нажимаем кнопку OK. После этого в красном кружке, обозначающем точку останова, появится белый плюс.

В предыдущем разделе мы рассматривали поиск ошибки с помощью вставки в код инструкций для вывода промежуточных значений. С помощью точек останова можно выполнить такое же действие, но без вставки инструкций в исходный код. Для этого делаем строку активной, щелкаем на ней правой кнопкой мыши и из контекстного меню выбираем пункт Точка останова | Вставить точку отслеживания. В открывшемся окне вводим текст сообщения, используя специальные символы, информация о которых содержится в окне, а затем нажимаем кнопку OK. После этого слева от строки с

Листинги на странице http://unicross.narod.ru/cpp/
инструкцией отобразится красный ромб. Текст сообщения при отладке будет отображаться в окне Вывод.
Программирование на C++ в Visual Studio® 2010 Express

Глава 12.

Ввод и вывод данных

© Прокохренок Н.А., 2010 г., unicross@mail.ru

Этот DjVu-файл предоставляется КАК ЕСТЬ. Автор не несет никакой ответственности за прямые или косвенные проблемы, связанные с использованием данного файла. ВЫ ИСПОЛЬЗУЕТЕ ЕГО НА СВОЙ СТРАХ И РИСК.

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Запрещено:
- перепечатывать всю книгу или отдельные главы без письменного разрешения Автора;
- декомпилировать данный файл и преобразовывать его в любой другой формат.
Язык C++ поддерживает две системы ввода/вывода. Первая система позаимствована от языка C. Вторая система является объектно-ориентированной и доступна только в языке C++. В этой главе мы рассмотрим обе системы.

**Примечание**
В гл. I мы уже изучали ввод/вывод данных в окно консоли. В этой главе описываются только дополнительные возможности применительно к файловой системе, а также форматированный вывод данных.

**Работа с файлами в языке C**
Язык C++ поддерживает операции ввода/вывода из языка C, объявленные в файле stdio.h. В языке C++ вместо этого файла можно подключать файл cstdio:
```
#include <cstdio>
```
Следует учитывать, что рассматриваемые далее функции применяются в языке C. Хотя их можно использовать и в языке C++, тем не менее для ввода/вывода данных стоит отдать предпочтение объектно-ориентированной системе ввода/вывода языка C++. Описание функций языка C приведено в этой книге лишь для того, чтобы вы могли разобраться в чужом коде.

**Открытие и закрытие файла**
Для открытия файла предназначена функция fopen(). Прототип функции:
```
#include <cstdio>
FILE *fopen(const char *Filename, const char *Mode);
```
Количество одновременно открытых файлов ограничено значением макроса fopen_max. Определение макроса выглядит так:
```
#define FOPEN_MAX 20
```
В качестве результата функция возвращает указатель на структуру FILE. Этот указатель следует передавать функциям, которые предназначены для работы с уже открытым файлом. Если открыть файл не удалось, функция возвращает нулевой указатель и присваивает глобальной переменной errno значение макроса EINVAL. Структура FILE определена следующим образом:
```
struct _iobuf {
    char * __ptr;
    int    __cnt;
```

Листинги на странице http://unicross.narod.ru/cpp/
char *_base;
int _flag;
int _file;
int _charbuf;
int _bufsiz;
char * _tmpfname;
}

typedef struct _iobuf FILE;

В параметре filename указывается путь к файлу. Длина пути ограничена значением макрока FILENAME_MAX. Определение макрока:

#define FILENAME_MAX 260

Путь может быть абсолютным или относительным. При указании абсолютного пути следует учитывать, что слеш является специальным символом. По этой причине его необходимо удваивать. Пример:

std::FILE *fp;
fp = std::fopen("C:\\temp\\new\\file.txt", "w");

Если вместо двух слешей использовать только один, то в пути будут присутствовать сразу три специальных символа: \t, \n и \f. После преобразования специальных символов путь будет выглядеть следующим образом:

C:\Табуляция>emp<Перевод строки>ew<Перевод формата>ile.txt

Вместо абсолютного пути к файлу можно указать относительный путь. В этом случае путь определяется с учетом местоположения текущего рабочего каталога. Обратите внимание на то, что текущий рабочий каталог не всегда совпадает с каталогом, в котором находится исполняемый файл. При запуске из редактора VC++ текущим рабочим каталогом будет каталог проекта, а не каталог, в котором находится исполняемый файл. Если файл запускается с помощью двойного щелчка на значке, то каталоги будут совпадать. Если же файл запускается из командной строки, то текущим рабочим каталогом будет каталог, из которого запускается файл. Получить строковое представление текущего рабочего каталога в VC++ позволяет функция _getcwd(). Прототип функции:

#include <direct.h>

char * _getcwd(char * DstBuf, int SizeInBytes);

В первом параметре функция принимает указатель на буфер, а во втором параметре — максимальный размер буфера. Пример:

char buf[200] = {0};

Листинг на странице http://unicross.narod.ru/cpp/
_getcwd(buf, 200);
std::cout << buf << std::endl; // C:\book\test\test
Если в качестве параметров указаны нулевые значения, то строка создается динамически с помощью функции malloc(). В этом случае функция _getcwd() возвращает указатель на эту строку или нулевой указатель в случае ошибки. После окончания работы со строкой следует освободить память с помощью функции free().
Пример:
char *buf = 0;
buf = _getcwd(0, 0);
if (buf) {
    std::cout << buf << std::endl; // C:\book\test\test
    std::free(buf);
    buf = 0;
}
Возможны следующие варианты указания относительного пути в Windows:

→ если открываемый файл находится в текущем рабочем каталоге, то можно указать только название файла или перед названием файла добавить одну точку и прямой или обратный слэш. Пример:
fp1 = std::fopen("file.txt", "w");
fp2 = std::fopen("./file.txt", "w");
fp3 = std::fopen("\file.txt", "w");

→ если открываемый файл расположен во вложенную папку, то перед названием файла перечисляются названия вложенных папок через прямой или обратный слэш. Пример:
fp1 = std::fopen("folder1\file.txt", "w");
fp2 = std::fopen("folder1/file.txt", "w");
fp3 = std::fopen("folder1/folder2\file.txt", "w");
fp4 = std::fopen("folder1/folder2/file.txt", "w");

→ если папка с файлом расположена ниже уровнем, то перед названием файла указываются две точки и прямой или обратный слэш. Пример:
fp1 = std::fopen("../\file.txt", "w");
fp2 = std::fopen("..//file.txt", "w");
fp3 = std::fopen("../../\file.txt", "w");

Листинги на странице http://unicross.narod.ru/cpp/
fp4 = std::fopen("././file.txt", "w");

→ если в начале пути расположен слэш, то путь отсчитывается от корня диска. В этом
случае местоположение текущего рабочего каталога не имеет значения. Пример:
fp1 = std::fopen("\\book\\test\\file.txt", "w");
fp2 = std::fopen("/book/test/file.txt", "w");

Параметр mode в функции fopen() может принимать следующие значения внутри С-
строки:

→ r — только чтение. После открытия файла курсор устанавливается на начало
файла. Если файл не существует, то функция fopen() возвращает нулевой
указатель;

→ r+ — чтение и запись. После открытия файла курсор устанавливается на начало
файла. Если файл не существует, то функция fopen() возвращает нулевой
указатель;

→ w — запись. Если файл не существует, то он будет создан. Если файл существует, то
он будет перезаписан. После открытия файла курсор устанавливается на начало
файла;

→ w+ — чтение и запись. Если файл не существует, то он будет создан. Если файл
существует, то он будет перезаписан. После открытия файла курсор устанавливается
на начало файла;

→ a — запись. Если файл не существует, то он будет создан. Запись осуществляется в
конец файла. Содержимое файла не удаляется;

→ a+ — чтение и запись. Если файл не существует, то он будет создан. Чтение
производится с начала файла. Запись осуществляется в конец файла. Содержимое
файла не удаляется.

Примечание
В Microsoft Visual C++ имеются дополнительные режимы с, n, t, S, R, T и D, а также
параметр ccs, позволяющий указать кодировку файла. Эти режимы не входят в стандарт
языка C++. За дополнительной информацией обращайтесь к документации.

После режима может следовать модификатор:

→ b — файл будет открыт в бинарном режиме;

Листинги на странице http://unicross.narod.ru/cpp/
t — файл будет открыт в текстовом режиме (значение по умолчанию в Windows). В этом режиме при чтении символ \r будет уделен, а при записи наоборот добавлен.

В качестве примера откроем файл на запись в текстовом режиме и запишем в него две строки, разделенные символом \n (листинг 12.1). Затем откроем тот же файл на чтение в бинарном режиме и считаем первую строку. Чтобы продемонстрировать факт добавления символа \r при записи в текстовом режиме, выведем коды символов, которые расположены в конце считанной строки.

Листинг 12.1. Работа с файлами в текстовом и бинарном режимах

```cpp
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

int main() {
    char buf[200] = {0};
    std::FILE *fp = 0;
    fp = std::fopen("file.txt", "w"); // Открываем файл на запись в текстовом режиме

    if (!fp) {
        std::perror("Error"); // Проверяем успешность
        std::cin.get();
        std::exit(1); // Выходим при ошибке
    }

    std::puts("String1\nString2", fp); // Записываем строку
    std::fflush(fp); // Сбрасываем буфер
    std::fclose(fp); // Закрываем файл
    fp = std::fopen("file.txt", "rb"); // Открываем файл на чтение в бинарном режиме

    if (!fp) {
        std::perror("Error"); // Выводим сообщение
        std::cin.get();
        std::exit(1); // Выходим при ошибке
    }

    std::fgets(buf, 30, fp); // Читаем одну строку
    std::cout << (int)buf[7] << " " << // Результат:
```

Листинги на странице http://unicross.narod.ru/cpp/
(int)buf[8] << std::endl;  // 13 10 == \r \n
std::fclose(fp);          // Закрываем файл

std::cin.get();           
return 0;
}

Попробуйте во втором случае изменить режим с "rb" на "r". В результате вывод будет "10 0", а не "13 10". Это доказывает, что при чтении в текстовом режиме символ \r удаляется.

В этом примере мы воспользовались несколькими новыми функциями. Функция perror() выводит текст сообщения об ошибке. Перед этим сообщением добавляется текст, переданный в параметре, и символ двоеточия. Например, если в первом случае файл доступен только для чтения, то получим следующее сообщение в окне консоли: "Error: Permission denied". Если во втором случае файл не существует, то сообщение будет другим: "Error: No such file or directory". Прототип функции:

```cpp
#include <cstdio>
void perror(const char *ErrMsg);
```

Вместо функции perror() можно воспользоваться функцией strerror(), которой в параметре передается значение переменной errno. Пример:

```cpp
std::cout << std::strerror(errno) << std::endl;
```

Функция fputs() записывает C-строку в файл. Для ускорения работы производится буферизация записываемых данных. Информация из буфера записывается в файл полностью только в момент закрытия файла. Сбросить буфер в файл яким образом можно с помощью функции fflush(). Прототип функции:

```cpp
#include <cstdio>
int fflush(FILE *File);
```

Прочитать одну строку из файла позволяет функция fgets(). Считывание производится пока не будет достигнут символ перевода строки или из файла не будет прочитано указанное количество символов минус один символ (используется для вставки нулевого символа).

Закрыть файл позволяет функция fclose(). Прототип функции:

```cpp
#include <cstdio>
int fclose(FILE *File);
```

В единственном параметре передается указатель на открытый ранее файл. В качестве значения функция возвращает:

Листинги на странице http://unicross.narod.ru/cpp/
0 — если файл успешно закрыт;

EOF — если при закрытии файла произошла ошибка, например, дискета была вынута из дисковода или нет свободного места на диске. Макрос EOF определен следующим образом:

```c
#define EOF (-1)
```

При использовании функции fopen() в VC++ выводится предупреждающее сообщение "warning C4996: 'fopen': This function or variable may be unsafe". Чтобы избежать вывода сообщения вместо функции fopen() можно использовать функцию fopen_s(). Прототип функции:

```c
#include <stdio.h>
errno_t fopen_s(FILE **File, const char *Filename, const char *Mode);
```

В первом параметре функция принимает адрес указателя на файловую структуру. Остальные параметры аналогичны параметрам функции fopen(). В качестве результата функция возвращает 0 при отсутствии ошибки или код ошибки. Пример использования функции fopen_s() приведен в листинге 12.2.

### Листинг 12.2. Работа с файлами в текстовом и бинарном режимах

```c
#include <iostream>
#include <cstdio>
#include <cstdlib>

int main() {
    std::FILE *fp = 0;
    errno_t err;
    err = fopen_s(&fp, "file.txt", "w"); // Открываем файл на запись
    if (err != 0) { // Проверяем успешность
        std::cerr("Error"); // Выводим сообщение
        std::cin.get();
        std::exit(1); // Выходим при ошибке
    }
    std::fputs("String1\nString2", fp); // Записываем строку
    std::fflush(fp); // Сбрасываем буфер
    std::fclose(fp); // Закрываем файл
    std::cout << "OK" << std::endl;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
Запись в файл и чтение из файла

Для записи в файл предназначены следующие функции:

- `fputc()` и `putc()` — записывают символ в файл. Функции возвращают код записанного символа или значение макроса `EOF` в случае ошибки. Прототипы функций:
  ```
  #include <cstdio>
  int fputc(int Ch, FILE *File);
  int putc(int Ch, FILE *File);
  ```

  Пример:
  ```
  std::fputc('T', fp);
  std::putc('S', fp);
  ```

- `fputs()` — записывает C-строку в файл. Функция возвращает неотрицательное число, если ошибок нет, и значение макроса `EOF` при наличии ошибки. Прототип функции:
  ```
  #include <cstdio>
  int fputs(const char *Str, FILE *File);
  ```

  Пример:
  ```
  std::fputs("String1\nString2", fp);
  ```

- `fprintf()` — производит форматированный вывод в файл. При наличии ошибки функция возвращает значение макроса `EOF`. Прототип функции:
  ```
  #include <cstdio>
  int fprintf(FILE *File, const char *Format, ...);
  ```

  В параметре `Format` указывается строка специального формата. Внутри этой строки можно указать обычные символы и спецификаторы формата, начинающиеся с символа `%`. Эти спецификаторы мы рассматривали при изучении функции `printf()`. Пример использования функции `fprintf()`:
  ```
  int a = 10, b = 20;
  std::fprintf(fp, "%d %d", a, b);
  ```

  Листинги на странице http://unicross.narod.ru/cpp/
fwrite() — записывает объект в файл. Обратите внимание на то, что файл должен быть открыт в бинарном режиме. Функция возвращает количество записанных объектов. Прототип функции:

```c
#include <cstdio>
size_t fwrite(const void *Buf, size_t Size, size_t Count, FILE *File);
```

В первом параметре функция принимает указатель на объект. Во втором параметре указывается размер объекта в байтах, а в третьем параметре — количество записываемых объектов. Размер объекта следует определять с помощью оператора sizeof. В качестве примера запишем структуру в файл, открытый на запись в бинарном режиме:

```c
struct Point {
    int x, y;
} point;
point.x = 10;
point.y = 20;
int x = std::fwrite(&point, sizeof(Point), 1, fp);
std::cout << x << std::endl; // 1
```

Перечислим функции, предназначенные для чтения файла:

- fputs() и getc() — считывают один символ из файла. В качестве значений функции возвращают код прочитанного символа. Если в файле нет больше символов или произошла ошибка, то функции возвращают значение константы EOF. Прототипы функций:

```c
#include <cstdio>
int fputs(FILE *File);
int getc(FILE *File);
```

Пример использования функций:

```c
char ch1 = std::fputc(fp);
std::cout << ch1 << std::endl;
char ch2 = std::getc(fp);
std::cout << ch2 << std::endl;
```

- fgets() — читает одну строку из файла. Прототип функции:

```
Листинг на странице http://unicross.narod.ru/cpp/
```
#include <cstdio>

char *fgets(char *Buf, int MaxCount, FILE *File);

Считывание производится до первого символа перевода строки или до конца файла или пока не будет прочитано `MaxCount - 1` символов. Возвращаемая строка включает символ перевода строки. Исключением является последняя строка. Если она не завершается символом перевода строки, то символ добавлен не будет. Если произошла ошибка или достигнут конец файла, то функция возвращает нулевой указатель. Пример считывания одной строки из файла:
char buf[200] = {0}, *pch;
pch = std::fgets(buf, 200, fp);
if (pch) std::cout << buf << std::endl;

→ `fscanf()` — позволяет считать данные из файла и автоматически преобразовать в конкретный тип, например, в целое число. Прототип функции:
#include <cstdio>

int fscanf(FILE *File, const char *Format, ...);

В параметре `Format` указывается строка специального формата, внутри которой задаются спецификаторы, аналогичные применяемым в функции `scanf()`. В последующих параметрах передаются ссылки на переменные, в которых будет сохранено значение. В качестве результата функция возвращает количество преобразованных данных. Пример считывания из файла целого числа:
int x = 0, count = 0;
count = std::fscanf(fp, "%d", &x); // Символ & обязательно!!!
std::cout << count << std::endl;
std::cout << x << std::endl;

В VC++ при использовании функции `fscanf()` выводится предупреждающее сообщение "warning C4996: ‘fscanf‘: This function or variable may be unsafe“. Чтобы избежать этого сообщения следует применять функцию `fscanf_s()`. Прототип функции:
int fscanf_s(FILE *File, const char *Format, ...);

В этом случае при считывании строки вначале указывается адрес строки, а затем максимальное количество символов. Пример считывания строки:
char str[100] = {0};
// Символ & для строки не указывается!!!

Листинги на странице http://unicross.narod.ru/cpp/
fscanf_s(fp, "%s", str, 100);
std::cout << str << std::endl;

 fread() — считывает объект из файла. Обратите внимание на то, что файл должен быть открыт в бинарном режиме. Прототип функции:
#include <cstdio>
size_t fread(void *DstBuf, size_t ElementSize, size_t Count,
             FILE *File);

В первом параметре функция принимает указатель на объект. Во втором параметре указывается размер объекта в байтах, а в третьем параметре — количество считываемых объектов. Размер объекта следует определять с помощью оператора sizeof. Функция возвращает количество считанных объектов.

Запишем структуру в файл, открытый на запись в бинарном режиме, а затем считаем структуру из файла и выведем значения (листинг 12.3).

Листинг 12.3. Сохранение структуры в файл

#include <iostream>
#include <cstdio>
#include <stdlib>

struct Point {
  int x, y;
} point1, point2;

int main() {
  std::FILE *fp = 0;
  int x = 0;
  if (fopen_s(&fp, "file.txt", "wb") != 0) { // Бинарный режим!
    std::cerr("Error");
    std::cin.get();
    std::exit(1);
  }
  point1.x = 10;
  point1.y = 20;
  x = std::fwrite(&point1, sizeof(Point), 1, fp);

  Листинги на странице http://unicross.narod.ru/cpp/
std::cout << x << std::endl; // 1
std::fflush(fp);
std::fclose(fp);
if (fopen_s(&fp, "file.txt", "rb") != 0) { // Бинарный режим!
    std::perror("Error");
    std::cin.get();
    std::exit(1);
}
x = std::fread(&point2, sizeof(Point), 1, fp);
std::cout << point2.x << std::endl; // 10
std::cout << point2.y << std::endl; // 20
std::cout << x << std::endl; // 1
std::fclose(fp);
std::cin.get();
return 0;
}
В качестве еще одного примера запишем строку в файл, а затем произведем посимвольное считывание (листинг 12.4).

**Листинг 12.4. Посимвольное считывание из файла**

```cpp
#include <iostream>
#include <stdio>
#include <stdlib>

int main() {
    std::FILE *fp = 0;
    char ch = 0;
    if (fopen_s(&fp, "file.txt", "w+") != 0) { // Чтение и запись
        std::perror("Error");
        std::cin.get();
        std::exit(1);
    }
    std::fputs("String1\nString2", fp); // Записываем строку
    std::fflush(fp);
    std::rewind(fp); // Перемещаем курсор в начало файла
    Листинги на странице http://unicross.narod.ru/cpp/
ch = std::fgetc(fp);   // Считываем один символ
while (!std::feof(fp)) {
    if (std::ferror(fp)) {  // Проверяем наличие ошибок
        std::cerr("Error");
        std::cin.get();
        std::exit(1);
    }
    std::cout << ch << " (" << (int)ch
              << ")" << std::endl;
    ch = std::fgetc(fp);   // Считываем один символ
}
std::fclose(fp);
std::cin.get();
return 0;
}

В этом примере были использованы три новые функции:

- **rewind()** — устанавливает курсор на начало файла. Прототип функции:

```cpp
#include <cstdio>

void rewind(FILE *File);
```

- **feof()** — возвращает ненулевое значение, если достигнут конец файла. В противном случае возвращается значение 0. Прототип функции:

```cpp
#include <cstdio>

int feof(FILE *File);
```

- **ferror()** — возвращает ненулевое значение, если при работе с файлом возникла ошибка. В противном случае возвращается значение 0. Функцию следует вызывать сразу после выполнения операции. Прототип функции:

```cpp
#include <cstdio>

int ferror(FILE *File);
```

Удалить все ошибки, связанные с потоком, позволяет функция clearerr(). Прототип функции:

```cpp
#include <cstdio>

void clearerr(FILE *File);
```

В VC++ вместо функции clearerr() можно использовать функцию clearerr_s(). Если операція успешно произведена, то функция возвращает значение 0. Прототип функции:

```c
#include <stdio.h>
errno_t clearerr_s(FILE *File);
```

**Файлы произвольного доступа**

Файл можно читать не только с самого начала, но и с произвольной позиции. Как правило в этом случае файл открывают в бинарном режиме. Записывать в файл также можно с произвольной позиции. Для работы с файлами произвольного доступа предназначены следующие функции:

- `fgetpos()` — записывает позицию курсора относительно начала файла в переменную, адрес которой передается во втором параметре. В качестве результата функция возвращает значение 0, если ошибок не возникло, и ненулевое значение в противном случае. Прототип функции:

```c
#include <cstdio>
int fgetpos(FILE *File, fpos_t *Pos);
```

Тип `fpos_t` объявлен как целочисленный тип. Пример использования функции:

```c
std::fpos_t pos = 0;
fgetpos(fp, &pos);
std::cout << pos << std::endl;
```

- `ftell()` — возвращает позицию курсора относительно начала файла. При ошибке возвращается значение -1L. Прототип функции:

```c
#include <cstdio>
long ftell(FILE *File);
```

Пример использования функции:

```c
long pos = 0;
pos = std::ftell(fp);
std::cout << pos << std::endl;
```

В VC++ вместо функции `ftell()` можно воспользоваться функцией `_ftelli64()`. Прототип функции:

```c
#include <stdio.h>
```

Листинги на странице http://unicross.narod.ru/cpp/
__int64 _ftelli64(FILE *File);

Пример:
__int64 pos = 0;
pos = _ftelli64(fp);
std::cout << pos << std::endl;

rewind() — устанавливает курсор на начало файла. Прототип функции:
#include <cstdio>
void rewind(FILE *File);

fsetpos() — перемещает курсор относительно начала файла. Адрес переменной, в которой содержится значение, передается во втором параметре. В качестве результата функция возвращает значение 0, если ошибок не возникло, и ненулевое значение в противном случае. Прототип функции:
#include <cstdio>
int fsetpos(FILE *File, const fpos_t *Pos);

Пример:

fpos_t pos = 2 * sizeof(int);
std::fsetpos(fp, &pos);

fseek() — устанавливает курсор в позицию, имеющую смещение offset относительно позиции origin. В качестве результата функция возвращает значение 0, если ошибок не возникло, и ненулевое значение в противном случае. Прототип функции:
#include <cstdio>
int fseek(FILE *File, long Offset, int Origin);

В параметре origin могут быть указаны следующие макросы:
• SEEK_SET — начало файла;
• SEEK_CUR — текущая позиция курсора;
• SEEK_END — конец файла.
Определения макросов выглядят следующим образом:
define SEEK_SET 0
define SEEK_CUR 1

Листинги на странице http://unicross.narod.ru/cpp/
В VC++ вместо функции fseek можно воспользоваться функцией _fseeki64(). Прототип функции:
#include <stdio.h>
int _fseeki64(FILE *File, __int64 Offset, int Origin);

В качестве примера использования файлов произвольного доступа запишем массив в файл, а затем считаем из файла значения первого и предпоследнего элементов массива (листинг 12.5).

Листинг 12.5. Файлы произвольного доступа

#include <iostream>
#include <cstdio>
#include <cstdlib>

int main() {
    FILE *fp = 0;
    int arr[] = { 1, 2, 3, 4, 5 }, x = 0, size = 0;
    long pos = 0;
    if (fopen_s(&fp, "file.txt", "wb") != 0) { // Бинарный режим!!!
        std::cerr("Error");
        std::cin.get();
        std::exit(1);
    }
    size = sizeof(int);
    std::fwrite(arr, size, 5, fp); // Записываем массив
    std::fflush(fp);
    std::fseek(fp, 0L, SEEK_SET); // Перемещаем курсор в начало файла
    pos = std::ftell(fp);
    std::cout << pos << std::endl; // 0
    std::fread(&x, size, 1, fp);
    std::cout << x << std::endl; // 1
    // Перемещаем курсор на предпоследний элемент
    std::fseek(fp, -2L * size, SEEK_END);
    pos = std::ftell(fp);
    std::cout << pos << std::endl; // 12
}

Листинги на странице http://unicross.narod.ru/cpp/
std::fread(&x, size, 1, fp);
std::cout << x << std::endl; // 4
std::fclose(fp);
std::cin.get();
return 0;
}

Создание временных файлов

Для создания временного файла предназначена функция tmpfile(). Временный файл создается в корневой папке. Функция возвращает файловый указатель на поток, открытый в режиме w+b, или нулевой указатель в случае ошибки. После закрытия временный файл автоматически удаляется. Обратите внимание на то, что в некоторых операционных системах для создания временного файла могут потребоваться права администратора. Прототип функции:

```c++
#include <stdio.h>
FILE *tmpfile(void);
```

В VC++ при использовании функции tmpfile() выводится предупреждающее сообщение "warning C4996: 'tmpfile': This function or variable may be unsafe". Чтобы избежать этого сообщения следует применять функцию tmpfile_s(). Прототип функции:

```c++
#include <stdio.h>
errno_t tmpfile_s(FILE **File);
```

При успешном выполнении операции функция возвращает значение 0, а в противном случае код ошибки. Пример использования функции tmpfile_s() приведен в листинге 12.6.

### Листинг 12.6. Создание временного файла

```c++
#include <iostream>
#include <stdio.h>
#include <cstdlib>

int main() {
    std::FILE *fp = 0;
    char buf[200] = {0}, *pch = 0;
    if (tmpfile_s(&fp)) {
        // Проверяем успешность
    }
}
```

Листинги на странице http://unicross.narod.ru/cpp/
Вместо создания временного файла можно сгенерировать уникальное название с помощью функции tmpnam(), а затем создать файл обычным образом. Длина названия ограничена значением макроса L_tmpnam. Прототип функции:

```cpp
#include <cstdio>
char *tmpnam(char *Buffer);
```

Если произошла ошибка, то функция возвращает нулевой указатель. Сгенерированное название записывается в символьный массив buffer. Если в параметре передан нулевой указатель, то сгенерированное название размещается во внутреннем статическом буфере, а ссылка на него возвращается в качестве значения. Эти данные будут перезаписаны при следующем вызове функции tmpnam(). Пример использования функции:

```cpp
char buf[L_tmpnam] = {0}, *p = 0;
p = std::tmpnam(buf);
if (!p) std::cout << buf << std::endl; // Например, \s3g4.
std::cout << L_tmpnam << std::endl; // В VC++ 2010: 14
```

В VC++ при использовании функции tmpnam() выводится предупреждающее сообщение "warning C4996: 'tmpnam': This function or variable may be unsafe". Чтобы избежать этого сообщения следует применять функцию tmpnam_s(). Прототип функции:

```cpp
#include <stdio.h>
```
errno_t tmpnam_s(char *Buf, rsize_t Size);

При успешном выполнении операции функция возвращает значение 0, а в противном случае код ошибки. Сгенерированное название сохраняется в символьном массиве Buf, максимальный размер которого указывается в параметре Size. Длина названия ограничена значением макроса L_tmpnam_s. Пример:

```
char buf[L_tmpnam_s] = {0};
if (tmpnam_s(buf, L_tmpnam_s) == 0) {
    std::cout << buf << std::endl; // Например, \s3i4.
}
```

std::cout << L_tmpnam_s << std::endl; // В VC++ 2010: 18

Кроме перечисленных функций, в VC++ для генерации названия временного файла можно воспользоваться функцией _tempnam(). Прототип функции:

```
#include <stdio.h>
char * _tempnam(const char *DirName, const char *FilePrefix);
```

В первом параметре указывается путь к каталогу, который будет использоваться если переменная окружения TMP не определена или содержит некорректный путь. Если переменная TMP корректна, то будет использоваться ее значение. Во втором параметре указывается префикс. Функция динамически выделяет под название необходимый объем памяти с помощью функции malloc() и возвращает указатель на него. Обратите внимание на то, что ответственность за освобождение динамической памяти лежит на плечах программиста. Поэтому после использования памяти следует обязательно вызвать функцию free() и передать ей указатель. В случае ошибки функция возвращает нулевой указатель. Пример использования функции:

```
char *p = _tempnam("C:\\book", "tmp_”);
if (p) {
    std::cout << p << std::endl;
    // Например, C:\DOCUME~1\user\LOCALS~1\Temp\tmp_2
    std::free(p); // Освобождаем память!!!
}
```

**Перенаправление ввода/вывода**

Потоки ввода/вывода на консоль аналогичны потокам ввода/вывода в файл. При запуске программы автоматически открываются три потока:

- `stdin` — стандартный ввод;
- `stdout` — стандартный вывод;

Листинги на странице http://unicross.narod.ru/cpp/
stderr — стандартный вывод сообщений об ошибках.

Все эти идентификаторы являются файловыми указателями, связанными по умолчанию с окном консоли. Следовательно их можно указывать вместо обычных файловых указателей в функциях, предназначенных для работы с файлами. Например, вывести строку в окно консоли можно так:

```cpp
std::fputs("String1\nString2", stdout);
std::fflush(stdout);
```

Ввод символа выполняется следующим образом:

```cpp
char ch = std::fgetc(stdin);
```

Стандартные потоки можно перенаправить таким образом, чтобы данные записывались в файл или считывались из файла. Для этого существуют два способа:

- перенаправить из командной строки. В этом случае ничего в программе менять не нужно. Чтобы выполнить перенаправление вывода в файл следует в командной строке выполнить одну из команд:

```bash
test.exe > file.txt
test.exe >> file.txt
```

Первая строка записывает результат выполнения программы `test.exe` в файл `file.txt`. Если файл не существует, то он будет создан, а если существует, то он будет перезаписан. Вторая строка производит дозапись в конец файла.

Чтобы ввести в программу данные из файла следует выполнить команду:

```bash
test.exe < file.txt
```

- перенаправить с помощью функции `freopen()`. Прототип функции:

```cpp
#include <cstdio>
FILE *freopen(const char *Filename, const char *Mode, FILE *File);
```

Первые два параметра аналогичны параметрам функции `fopen()`. Пример перенаправления стандартного вывода в файл:

```cpp
std::FILE *fp = 0;
fp = std::freopen("file.txt", "w", stdout);
if (!fp) std::exit(1);
std::printf("%s\n", "Пишем в файл!");
```

Листинги на странице http://unicross.narod.ru/cpp/
В VC++ при использовании функции freopen() выводится предупреждающее сообщение "warning C4996: 'freopen': This function or variable may be unsafe". Чтобы избежать этого сообщения следует применять функцию freopen_s(). Прототип функции:

#include <stdio.h>

errno_t freopen_s(FILE **File, const char *Filename,
                   const char *Mode, FILE *OldFile);

Первые три параметра аналогичны параметрам функции fopen_s(). Пример использования функции freopen_s():

std::FILE *fp = 0;
if (freopen_s(&fp, "file.txt", "w", stdout) != 0) {
    std::exit(1);
}

std::printf("%s\n", "Пишем в файл!");

Ввод/вывод расширенных символов и строк

Функции для ввода/вывода расширенных символов и строк являются аналогами функций, предназначенных для ввода/вывода обычных символов и строк. В названии таких функций добавляется буква "w", например, расширенная функция fprintf() является аналогом функции fprintf(). Кроме того, изменяется значение, возвращаемое при ошибке. Вместо значения макроса EOF возвращается значение макроса WEOF. Определение макроса выглядит так:

#define WEOF (wint_t)(0xFFFF)

Прежде чем использовать расширенные функции необходимо настроить локаль и подключить заголовочные файлы stdio и wchar:

#include <cstdio>
#include <cwchar>

Некоторые обычные функции могут работать как с потоками обычных символов, так и с потоками расширенных символов, например, функции fclose(), fwrite(), fread(), fprintf(), flush(), rewind(), feof(), ferror(), clearerr(), а также функции для работы с файлами произвольного доступа. Названия обычных функций и прототипы соответствующих им расширенных функций приведены в табл. 12.1.

Листинги на странице http://unicross.narod.ru/cpp/
Таблица 12.1. Функции для ввода/вывода расширенных символов и строк

<table>
<thead>
<tr>
<th>Обычная функция</th>
<th>Прототип расширенной функции</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Работа с консолью</strong></td>
<td></td>
</tr>
<tr>
<td><code>putchar()</code></td>
<td><code>wint_t putwchar(wchar_t Ch);</code></td>
</tr>
<tr>
<td><code>puts()</code></td>
<td><code>int _putws(const wchar_t *Str);</code></td>
</tr>
<tr>
<td><code>printf()</code></td>
<td><code>int wprintf(const wchar_t *Format, ...);</code></td>
</tr>
<tr>
<td><code>getchar()</code></td>
<td><code>wint_t getwchar(void);</code></td>
</tr>
<tr>
<td><code>gets()</code></td>
<td><code>wchar_t * _getws(wchar_t *Buf);</code></td>
</tr>
<tr>
<td><code>scanf()</code></td>
<td><code>int wscanf(const wchar_t *Format, ...);</code></td>
</tr>
<tr>
<td><code>scanf_s()</code></td>
<td><code>int wscanf_s(const wchar_t *Format, ...);</code></td>
</tr>
<tr>
<td><strong>Открытие файла</strong></td>
<td></td>
</tr>
<tr>
<td><code>fopen()</code></td>
<td><code>FILE * _wfopen(const wchar_t *Filename, const wchar_t *Mode);</code></td>
</tr>
<tr>
<td><code>fopen_s()</code></td>
<td><code>errno_t _wfopen_s(FILE **File, const wchar_t *Filename, const wchar_t *Mode);</code></td>
</tr>
<tr>
<td><code>freopen()</code></td>
<td><code>FILE * _wfreopen(const wchar_t *Filename, const wchar_t *Mode, FILE *OldFile);</code></td>
</tr>
<tr>
<td><code>freopen_s()</code></td>
<td><code>errno_t _wfreopen_s(FILE **File, const wchar_t *Filename, const wchar_t *Mode, FILE *OldFile);</code></td>
</tr>
<tr>
<td><strong>Запись в файл и чтение из файла</strong></td>
<td></td>
</tr>
<tr>
<td><code>fputc()</code></td>
<td><code>wint_t fputwc(wchar_t Ch, FILE *File);</code></td>
</tr>
<tr>
<td><code>putc()</code></td>
<td><code>wint_t putwc(wchar_t Ch, FILE *File);</code></td>
</tr>
<tr>
<td><code>fputs()</code></td>
<td><code>int fputws(const wchar_t *Str, FILE *File);</code></td>
</tr>
<tr>
<td><code>fprintf()</code></td>
<td><code>int fwprintf(FILE *File, const wchar_t *Format, ...);</code></td>
</tr>
<tr>
<td><code>fgets()</code></td>
<td><code>wint_t fgetwc(FILE *File);</code></td>
</tr>
<tr>
<td><code>getc()</code></td>
<td><code>wint_t getwc(FILE *File);</code></td>
</tr>
<tr>
<td><code>fgets()</code></td>
<td><code>wchar_t *fgetws(wchar_t *Dst, int SizeInWords, FILE *File);</code></td>
</tr>
</tbody>
</table>

Листинги на странице http://unicross.narod.ru/cpp/
<table>
<thead>
<tr>
<th>функция</th>
<th>определение</th>
</tr>
</thead>
<tbody>
<tr>
<td>fscanf()</td>
<td>int fscanf(FILE *File, const wchar_t *Format, ...);</td>
</tr>
<tr>
<td>fscanf_s()</td>
<td>int fwscanf_s(FILE *File, const wchar_t *Format, ...);</td>
</tr>
<tr>
<td>perror()</td>
<td>void _wperror(const wchar_t *ErrMsg);</td>
</tr>
<tr>
<td>rename()</td>
<td>int _wrename(const wchar_t *OldFilename, const wchar_t *NewFilename);</td>
</tr>
<tr>
<td>remove()</td>
<td>int _wremove(const wchar_t *Filename);</td>
</tr>
<tr>
<td>tmpnam()</td>
<td>wchar_t *_wtmpnam(wchar_t *Buffer);</td>
</tr>
<tr>
<td>tmpnam_s()</td>
<td>errno_t _wtmpnam_s(wchar_t *Buf, size_t Size);</td>
</tr>
<tr>
<td>__tempnam()</td>
<td>wchar_t *__wtempnam(const wchar_t *Directory, const wchar_t *FilePrefix);</td>
</tr>
</tbody>
</table>

В VC++ в функциях _wfopen() и _wfopen_s() параметр mode может содержать опцию ccs, которая задает кодировку файла. Опция принимает значения UNICODE, UTF-8 или UTF-16LE. Если указана опция UTF-8 или UTF-16LE, то файл создается в соответствующей кодировке и в начало файла вставляются служебные байты, сокращенно называемые BOM (Byte Order Mark, метка порядка байтов). Если открываемый файл содержит BOM, то значение, указанное в опции ccs, игнорируется. При отсутствии опции ccs автоматическое определение кодировки файла по BOM не производится. В качестве примера создадим файл в кодировке UTF-16LE, запишем строку, а затем выведем каждый символ и его код (листинг 12.7). Обратите внимание на то, что первым символом является BOM.

### Листинг 12.7. Создание и чтение файла в кодировке UTF-16LE

```cpp
#include <iostream>
#include <stdio>
#include <stdlib>
#include <locale>
#include <cwchar>

int main() {
    std::setlocale(LC_ALL, "Russian_Russia.1251");
}
```

Листинги на странице http://unicross.narod.ru/cpp/
Ввод/вывод данных в языке C++

Система ввода/вывода языка C++ является объектно-ориентированной и по умолчанию синхронизируется с потоками языка C. Поэтому в одной программе можно одновременно использовать обе системы. Тем не менее стоит отдать предпочтение только системе ввода/вывода языка C++.

Система ввода/вывода языка C++ состоит из иерархии шаблонных классов, которые позволяют работать как с обычными символами, так и с расширенными. Иерархия наследования для ввода данных выглядит так:

Листинги на странице http://unicross.narod.ru/cpp/
ios_base - basic_ios - basic_istream - basic_ifstream

Иерархия наследования для вывода данных:
ios_base - basic_ios - basic_ostream - basic_ofstream

Иерархия наследования для ввода и вывода данных:
ios_base - basic_ios - (basic_istream, basic_ostream) - basic_iostream
   - basic_fstream

Классом верхнего уровня является шаблонный класс ios_base. Шаблонный класс basic_ios наследует класс ios_base. В свою очередь класс basic_ios является базовым для шаблонных классов basic_istream и basic_ostream, которые предназначены для работы с консолью. Для использования этих классов необходимо подключить заголовочный файл iostream:

```
#include <iostream>
```

Для удобства использования шаблонных классов в заголовочном файле определяются более короткие имена с помощью оператора typedef:

```
typedef basic_istream< char, char_traits<char> > istream;
typedef basic_ostream< char, char_traits<char> > ostream;
typedef basic_iostream< char, char_traits<char> > iostream;
typedef basic_istream< wchar_t, char_traits<wchar_t> > wistream;
typedef basic_ostream< wchar_t, char_traits<wchar_t> > wostream;
typedef basic_iostream< wchar_t, char_traits<wchar_t> > wiostream;
```

Объект cin, предназначенный для ввода обычных символов с клавиатуры, является экземпляром класса istream, а объекты cout, cerr и clog, позволяющие вывести обычные символы в окно консоли, являются экземплярами класса ostream. Объект wcin, предназначенный для ввода расширенных символов, является экземпляром класса wistream, а объекты wcout, wcerr и wclog, позволяющие вывести расширенные символы, являются экземплярами класса wostream. Объекты cin, cout и cerr мы уже рассматривали в главе 1, а объекты wcin и wcout в главе 6. В этой главе мы рассмотрим работу с файлами, а также форматированный вывод данных. Большинство операций с консолью в языке C++ выполняется аналогичным образом.

**Классы basic_ifstream, basic_ofstream и basic_fstream**

Классы basic_ifstream и basic_ofstream являются базовыми для шаблонных классов basic_ifstream и basic_ofstream соответственно. Класс basic_ifstream предназначен для чтения данных из файла, а класс basic_ofstream — для записи данных в файл. Кроме того, существует шаблонный класс basic_fstream, который

Листинг на странице http://unicross.narod.ru/cpp/
позволяет как читать из файла, так и записывать в файл. Прежде чем использовать классы basic_ifstream, basic_ofstream и basic_fstream необходимо подключить заголовочный файл fstream:

```cpp
#include <fstream>
```

Для удобства использования шаблонных классов в заголовочном файле определяются более короткие имена с помощью оператора typedef:

```cpp
typedef basic_ifstream< char, char_traits<char> > ifstream;
typedef basic_ofstream< char, char_traits<char> > ofstream;
typedef basic_fstream< char, char_traits<char> > fstream;
typedef basic_ifstream< wchar_t, char_traits<wchar_t> > wifstream;
typedef basic_ofstream< wchar_t, char_traits<wchar_t> > wofstream;
typedef basic_fstream< wchar_t, char_traits<wchar_t> > wfstream;
```

В последующих примерах мы будем рассматривать только классы для работы с обычными строками. Работа с расширенными строками выполняется аналогично.

При создании экземпляра класса чаще всего используются два формата:

```cpp
<Klass> <Переменная>;
<Klass> <Переменная>({<Путь к файлу> [, <Режим> [, <Блокировка>]}});
```

Название класса зависит от необходимой операции с файлом:

- ifstream — только чтение;
- ofstream — только запись;
- fstream — одновременно и чтение и запись.

При использовании первого формата вызывается конструктор по умолчанию. В этом случае создается объект потока не связанный ни с каким файлом. Чтобы связать объект с конкретным файлом следует вызвать метод open(). Пример:

```cpp
std::ofstream file; // Объект для записи в файл
```

Второй формат позволяет при создании экземпляра класса сразу связать поток с конкретным файлом. В первом параметре указывается путь к файлу. Путь может быть абсолютным или относительным. При использовании абсолютного пути следует учитывать, что слеш необходимо обязательно удавливать:

```cpp
std::ofstream file("C:\book\file.txt");
```

При указании относительного пути полный путь определяется с учетом местоположения текущего рабочего каталога. Обратите внимание на то, что текущий рабочий каталог не всегда совпадает с каталогом, в котором находится исполняемый
файл. При запуске из редактора VC++ текущим рабочим каталогом будет каталог проекта, а не каталог, в котором находится исполняемый файл. Если файл запускается с помощью двойного щелчка на значке, то каталоги будут совпадать. Если же файл запускается из командной строки, то текущим рабочим каталогом будет каталог, из которого запускается файл. Способы указания относительного пути мы рассматривали при изучении системы ввода/вывода в языке C. Примеры:

```cpp
std::ofstream file1("file.txt");
std::ofstream file2("..\file.txt");
std::ofstream file3("folder1\file.txt");
```

Путь может быть C-строкой, расширенной строкой или объектом классов `string` и `wstring`. Пример:

```cpp
std::ofstream file1("file.txt"); // C-строка
std::ofstream file2(L"file.txt"); // Расширенная строка
std::string path("file.txt");
std::ofstream file3(path); // Экземпляр класса string
std::wstring wpath(L"file.txt");
std::ofstream file4(wpath); // Экземпляр класса wstring
```

Параметр `<Режим>` задает режим открытия файла. Если параметр не указан, то используются следующие режимы по умолчанию:

- `ifstream — ios::in` — файл открывается только на чтение в текстовом режиме. После открытия файла курсор устанавливается на начало файла. Если файл существует, то содержимое файла не удаляется. Если файл не существует, то устанавливается флаг `ios::failbit`;

- `ofstream — ios::out` — файл открывается только на запись в текстовом режиме. Если файл не существует, то он будет создан. Если файл существует, то он будет перезаписан. После открытия файла курсор устанавливается на начало файла;

- `fstream — ios::in | ios::out` — файл открывается одновременно на чтение и запись в текстовом режиме. После открытия файла курсор устанавливается на начало файла. Если файл не существует, то устанавливается флаг `ios::failbit`. Если файл существует, то содержимое файла не удаляется.

В параметре `<Режим>` можно указать следующие значения (обычно их комбинацию через оператор `|`):

- `ios::in` — файл открывается для чтения. Если файл существует, то содержимое файла не удаляется;

> **ios::out** — файл открывается для записи. Если файл существует, то содержимое файла стирается;

> **ios::trunc** — если файл существует, то его содержимое стирается. Режим используется по умолчанию, если установлен режим *ios::out*, а режимы *ios::in*, *ios::app* или *ios::ate* не указаны;

> **ios::app** — запись производится в конец файла, даже если указана другая позиция с помощью метода *seekp*();

> **ios::ate** — при открытии файла курсор устанавливается на конец файла. Запись производится в текущую позицию курсора, а не обязательно в конец файла как в режиме *ios::app*;

> **ios::binary** — файл открывается в бинарном режиме. По умолчанию используется текстовый режим. В текстовом режиме при чтении символ \r будет удален, а при записи наоборот добавлен.

Рассмотрим несколько примеров. Вначале откроем существующий файл для чтения в текстовом и бинарном режимах:

```cpp
// Текстовый режим чтения
std::ifstream file1("file.txt", std::ios::in);
// Бинарный режим чтения
std::ifstream file2("file.txt", std::ios::in | std::ios::binary);
```

В этом случае если файл не существует, то устанавливается флаг *ios::failbit*. Теперь откроем файл на запись с удалением содержимого:

```cpp
// Текстовый режим записи
std::ofstream file1("file.txt", std::ios::out);
// Бинарный режим записи
std::ofstream file2("file.txt", std::ios::out | std::ios::binary);
```

В этом случае если файл не существует, то он будет автоматически создан, а если существует, то его содержимое удаляется. Пример дозаписи в конец файла:

```cpp
// Текстовый режим дозаписи
std::ofstream file1("file.txt", std::ios::out | std::ios::app);
// Бинарный режим дозаписи
std::ofstream file2("file.txt",
    std::ios::out | std::ios::app | std::ios::binary);
```

Листинги на странице http://unicross.narod.ru/cpp/
При использовании режима дозаписи, если файл не существует, то он автоматически создается. Содержимое существующего файла не удаляется. Запись производится в конец файла. Теперь откроем файл на ввод и вывод одновременно:

```cpp
// Текстовый режим ввода и вывода
std::fstream file1("file.txt", std::ios::in | std::ios::out);
// Бинарный режим ввода и вывода
std::fstream file2("file.txt",
                  std::ios::in | std::ios::out | std::ios::binary);
```

В этом случае при отсутствии файла устанавливается флаг `ios::failbit`. Содержимое существующего файла не удаляется. После открытия файла курсор устанавливается на начало файла. Запись производится в текущую позицию курсора. Обратите внимание на то, что после чтения из файла необходимо переместить курсор прежде чем записывать в файл, иначе данные записаны не будут.

Необязательный параметр `<Блокировка>` задает режим доступа, который будет использоваться после открытия файла. Блокировка снимается после закрытия файла. В параметре могут быть указаны следующие макросы (определенны в заголовочном файле `share.h`) или их значения:

- `_SH_DENYRW` — 0x10 — другие потоки не могут одновременно читать из файла и записывать в файл, пока файл не будет закрыт;
- `_SH_DENYWR` — 0x20 — другие потоки могут одновременно читать из файла, но не могут записывать в файл;
- `_SH_DENYRD` — 0x30 — другие потоки не могут одновременно читать из файла, но могут записывать в файл, если файл был открыт ранее;
- `_SH_DENYNO` — 0x40 — другие потоки могут одновременно читать из файла и записывать в файл (значение по умолчанию).

Пример открытия файла на запись в текстовом режиме с запретом ввода и вывода для других потоков:

```cpp
#include <fstream>
#include <share.h>
// ... Фрагмент опущен ...
std::ofstream file("file.txt", std::ios::out, _SH_DENYRW);
```

Листинги на странице http://unicross.narod.ru/cpp/
Открытие и закрытие файла

Открыть файл можно сразу при создании объекта, передав конструктору путь к файлу. Этот способ мы рассматривали в предыдущем разделе. Если при создании объекта использовался конструктор по умолчанию, то открыть файл можно с помощью метода `open()`. Прототип метода:

```c++
void open( <Путь к файлу>[, <Режим>[, <Блокировка>]])
```

Предназначение всех параметров совпадает с предназначением параметров конструктора. Пример открытия файла на запись:

```c++
std::ofstream file;
file.open("file.txt");
```

Этот код эквивалентен следующему коду:

```c++
std::ofstream file("file.txt");
```

Если файл открыт не удалось, то устанавливается флаг ошибки для потока и глобальной переменной `errno` присваивается код ошибки. Проверить успешность открытия файла можно следующим образом:

```c++
std::ofstream file("file.txt");
if (!file) std::cout << "Не удалось открыть файл" << std::endl;
```

Проверить открыт файл или нет позволяет также метод `is_open()`. Если файл открыт, то метод возвращает значение `true`. Прототип метода:

```c++
bool is_open() const;
```

Пример проверки:

```c++
if (file.is_open()) std::cout << "Файл открыт" << std::endl;
```

Для закрытия файла предназначен метод `close()`. Если файл закрыть не удалось, то устанавливается флаг ошибки для потока и глобальной переменной `errno` присваивается код ошибки. Прототип метода:

```c++
void close();
```

В качестве примера откроем файл на запись в текстовом режиме и запишем в него две строки, разделенные символом \n (листинг 12.8). Файл блокируем на чтение и запись.

Листинг 12.8. Обработка ошибок при открытии файла

```c++
#include <fstream>
#include <iostream>
#include <cstdlib>
```

Листинги на странице http://unicross.narod.ru/cpp/
```cpp
#include <share.h>
#include <cstring>

void print_error(int err);

int main() {
    std::ofstream file;
    file.open("file.txt", std::ios::out, _SH_DENYRW);
    if (!file) print_error(errno);
    file << "String1\nString2";
    file.flush();
    if (file.is_open()) std::cout << "Open" << std::endl;    // Open
    else std::cout << "Close" << std::endl;
    file.close();
    if (file.is_open()) std::cout << "Open" << std::endl;
    else std::cout << "Close" << std::endl;         // Close
    std::cin.get();
    return 0;
}
void print_error(int err) {
    char str[200] = {0};
    strerror_s(str, 200, err);
    std::cout << "Error: " << str << std::endl;
    std::cin.get();
    std::exit(1);
}

Как видно из примера, для записи в файл можно использовать оператор <<. Для ускорения работы производится буферизация записываемых данных. Информация из буфера записывается в файл полностью только в момент закрытия файла. Сбросить буфер в файл явным образом можно с помощью метода flush(). Прототип метода:

```cpp
ostream &flush();
```

**Запись в файл и чтение из файла**

Для записи в файл предназначены следующие методы:

- оператор `<<` — позволяет записать данные любого типа. Пример:

int x = 250;
double y = 12.5;
char ch = 'w';
char str[] = "String1\nString2";
file << x << " " << y << " " << ch << " " << str;

put() — записывает символ. Прототип метода:

ostream &put(char ch);

Пример:
char ch = 'w';
file.put(ch);

write() — записывает объект в файл. Обратите внимание на то, что файл должен быть открыт в бинарном режиме. Прототип метода:

ostream &write(const char *Str, streamsize Count);

В первом параметре метод принимает символьный указатель. Чтобы передать объект следует выполнить приведение типа к char *. Во втором параметре указывается размер объекта в байтах. Размер объекта следует определять с помощью оператора sizeof. В качестве примера запишем структуру в файл, открытый на запись в бинарном режиме:

std::ofstream file("file.txt",
    std::ios::out | std::ios::binary);

if (!file) {
    std::cout << "Error" << std::endl;
    std::exit(1);
}

struct Point {
    int x, y;
} point;
point.x = 10;
point.y = 20;
file.write((char *)&point, sizeof(Point));
file.flush();

Листинги на странице http://unicross.narod.ru/cpp/
file.close();

Перечислим методы, предназначенные для чтения файла:

→ оператор >> — позволяет считать данные любого типа. Пробельные символы перед данными игнорируются. Пример считывания одного символа:

```cpp
// Содержимое файла: " \\t\\nString"
char ch;
file >> ch;
std::cout << ch << std::endl; // Результат: 'S'
```

Обратите внимание на то, что считывание строки производится до первого пробельного символа (например, пробела, табуляции, символа перевода строки и т. д.). Пробельный символ удаляется из потока, но в строку не записывается. Чтобы считать строку определенной длины следует использовать методы get() или getline();

→ get() — считывает один символ или строку определенной длины. В отличие от оператора >> при считывании символа пробельные символы не игнорируются. Метод имеет несколько прототипов. Первый прототип:

```cpp
int get();
```

Считывает один символ и возвращает его код. Если достигнут конец файла, то метод возвращает значение макроса EOF. Пример считывания символа:

```cpp
char ch = file.get();
std::cout << ch << std::endl;
```

Второй прототип:

```cpp
istream &get(char &Ch);
```

Считывает один символ и записывает его в переменную, адрес которой передан в параметре. Если достигнут конец файла, то в переменную записывается значение макроса EOF и устанавливается флаг eofbit. Кроме того, поток, связанный с файлом, принимает значение false. Проверить установку флага позволяет метод eof(). Прототип метода:

```cpp
bool eof() const;
```

Метод возвращает значение true, если достигнут конец файла. В противном случае метод возвращает значение false. Пример считывания одного символа с проверкой корректности:

Листинги на странице http://unicross.narod.ru/cpp/
char ch;
file.get(ch);
// Использование метода eof()
if (!file.eof()) std::cout << ch << std::endl;
else std::cout << "EOF" << std::endl;
// Проверка потока
if (file) std::cout << ch << std::endl;
else std::cout << "EOF" << std::endl;

Третий прототип:
istream &get(char *Str, streamsize Count);
Считывает из файла строку длиной count-1 символов. Если раньше встретится символ перевода строки или будет достигнут конец файла, то считывание прекращается досрочно. Символ перевода строки остается в потоке. В конец строки записывается нулевой символ. Пример:
char str[100] = {0};
file.get(str, 50);
std::cout << str << std::endl;

Четвертый прототип:
istream &get(char *Str, streamsize Count, char Delim);
Считывает из файла строку длиной count-1 символов. Если раньше встретится символ Delim или будет достигнут конец файла, то считывание прекращается досрочно. Символ Delim остается в потоке и доступен при следующей операции считывания. В конец строки записывается нулевой символ. Пример:
char str[100] = {0};
file.get(str, 50, 'g');
std::cout << str << std::endl;

peek() — читает один символ из файла и возвращает его код. Символ остается в потоке и доступен при следующей операции считывания. Прототип метода:
int_type peek();
Пример:
// Содержимое файла: "String"

Листинги на странице http://unicross.narod.ru/cpp/
char ch1, ch2;
ch1 = file.peek();
std::cout << ch1 << std::endl; // S
ch2 = file.get();
std::cout << ch2 << std::endl; // S

getline() — читает строку из файла. В отличие от третьего и четвертого форматов метода get() метод getline() считывает символ-ограничитель, а не оставляет его в потоке. Символ-ограничитель не записывается в строку. Прототипы метода:

```cpp
istream &getline(char *Str, streamsize Count);
istream &getline(char *Str, streamsize Count, char Delim);
```

Метод, соответствующий первому прототипу, считывает из файла строку длиной Count-1 символов. Если раньше встретится символ перевода строки или будет достигнут конец файла, то считывание прекращается досрочно. Символ перевода строки считывается из потока, но не записывается в строку str. В конец строки записывается нулевой символ. Пример:

```cpp
char str[100] = {0};
file.getline(str, 50);
std::cout << str << std::endl;
```

Метод, соответствующий второму прототипу, считывает из файла строку длиной Count-1 символов. Если раньше встретится символ Delim или будет достигнут конец файла, то считывание прекращается досрочно. Символ Delim считывается из потока, но не записывается в строку str. В конец строки записывается нулевой символ. Пример:

```cpp
char str[100] = {0};
file.getline(str, 50, 'g');
std::cout << str << std::endl;
```

ignore() — считывает указанное количество символов, но нигде их не сохраняет. Прототип метода:

```cpp
istream &ignore(streamsize Count=1, int_type Metadelim=EOF);
```

Первый параметр задает максимальное количество считываемых символов (по умолчанию один символ). Второй параметр задает символ-разделитель. Если символ-разделитель встретится раньше, чем будет почитано указанное количество символов, считывание прекращается досрочно.

Листинги на странице http://unicross.narod.ru/cpp/
символов, то считывание прекращается. В этом случае символ-разделитель также считывается из потока. Пример:

```c
file.ignore(9, '
');
```

**read()** — считывает объект из файла. Обратите внимание на то, что файл должен быть открыт в бинарном режиме. Прототип метода:

```c
istream &read(char *Str, streamsize Count);
```

В первом параметре метод принимает символьный указатель. Чтобы передать объект следует выполнить приведение типа к char *.

Во втором параметре указывается размер объекта в байтах. Размер объекта следует определять с помощью оператора sizeof. Нулевой символ в конце str автоматически не вставляется. Если в файле меньше байтов, чем указано в параметре count, то устанавливается флаг failbit. Выполнить проверку установки флага можно с помощью метода fail(), а получить реальное количество считанных байтов позволяет метод gcount(). Прототип метода gcount():

```c
streamsize gcount() const;
```

Запишем структуру в файл, открытый на запись в бинарном режиме, а затем считаем структуру из файла и выведем значения (листинг 12.9).

```
Листинг 12.9. Сохранение структуры в файл
```

```c
#include <fstream>
#include <iostream>
#include <stdlib.h>
#include <cstring>

struct Point {
  int x, y;
} point;

void print_error(int err);

int main() {
  std::fstream file("file.txt", std::ios::in | std::ios::out |
                      std::ios::trunc | std::ios::binary);
  if (!file) print_error(errno);

```

Листинги на странице http://unicross.narod.ru/cpp/
point.x = 10;
point.y = 20;
file.write((char *)&point, sizeof(Point)); // Записываем структуру
file.flush();
file.seekg(0, std::ios::beg); // Перемещаем указатель в начало
point.x = 0;
point.y = 0;
file.read((char *)&point, sizeof(Point)); // Считываем структуру
std::cout << point.x << std::endl; // 10
std::cout << point.y << std::endl; // 20
file.close();
std::cin.get();
return 0;
}

void print_error(int err) {
    char str[200] = {0};
    strerror_s(str, 200, err);
    std::cout << "Error: " << str << std::endl;
    std::cin.get();
    std::exit(1);
}

В качестве еще одного примера запишем строку в файл, а затем произведем посимвольное считывание (листинг 12.10).

**Листинг 12.10. Посимвольное считывание из файла**

```cpp
#include <fstream>
#include <iostream>

int main() {
    char ch = 0;
    std::fstream file("file.txt", std::ios::in | std::ios::out |
                        std::ios::trunc);
    if (!file) {
        std::cout << "Error" << std::endl;
        std::cin.get();
    }

    Листинги на странице http://unicross.narod.ru/cpp/```
return 1;
}
file << "String1\nString2";
file.flush();
file.seekg(0, std::ios::beg);
while (!file.get(ch).eof()) {
    std::cout << ch << " (" << (int)ch << ")" << std::endl;
}
file.close();
std::cin.get();
return 0;

Обратите внимание на этот фрагмент:
while (!file.get(ch).eof()) {
    std::cout << ch << " (" << (int)ch << ")" << std::endl;
}

Символы из файла считаются до тех пор, пока не будет достигнут конец файла. В этом случае метод eof() вернет значение true. Так как метод get() возвращает ссылку на поток, можно составить цепочку из вызовов методов и сразу после считывания символа проверить достигнут ли конец файла. При достижении конца файла поток принимает значение false. Следовательно предыдущий фрагмент можно записать еще проще:
while (file.get(ch)) {
    std::cout << ch << " (" << (int)ch << ")" << std::endl;
}

Некоторые считанные символы можно вернуть в поток. Обратите внимание, именно вернуть в поток, а не записать в файл. Этот символ будет доступен при следующей операции чтения. Вернуть последний считанный символ позволяет метод unget(). Прототип метода:

```
istream &unget();
```

Для примера считаем первый символ, выводим его, а затем вернем его в поток. Далее прочитаем строку из файла и выводим ее:

```cpp
// Содержимое файла: "String1\nString2"
char str[100] = {0}, ch = 0;
ch = file.get();
```

Листинг на странице http://unicross.narod.ru/cpp/
std::cout << ch << std::endl; // S
file.unget();
file.getline(str, 50);
std::cout << str << std::endl; // String1
Вернуть произвольный символ в поток можно с помощью метода putback(). Прототип метода:

`istream &putback(char Ch);`

Пример:

// Содержимое файла: "String1
String2"
char str[100] = {0}, ch = 0;
ch = file.get();
std::cout << ch << std::endl; // S
file.putback('s');
file.getline(str, 50);
std::cout << str << std::endl; // string1

Файлы произвольного доступа

Файл можно читать не только с самого начала, но и с произвольной позиции. Как правило в этом случае файл открывают в бинарном режиме. Записывать в файл также можно с произвольной позиции. Для работы с файлами произвольного доступа предназначены следующие методы:

- `tellg()` и `tellp()` — возвращают позицию соответствующего курсора относительно начала файла. При ошибке возвращается значение -1. Метод `tellg()` возвращает позицию курсора чтения, а `tellp()` — позицию курсора записи.

Прототипы методов:

```cpp
pos_type tellg();
pos_type tellp();
```

Пример:

```cpp
std::ifstream file("file.txt",
    std::ios::in | std::ios::binary);
if (!file) return 1;
std::cout << file.tellg() << std::endl; // 0
char ch = file.get();
std::cout << ch << std::endl;
```

Листинги на странице http://unicross.narod.ru/cpp/
std::cout << file.tellg() << std::endl; // 1
file.close();

seekg() и seekp() — устанавливают соответствующие курсоры в позицию, имеющую смещение Offset относительно позиции Origin, или в позицию Pos относительно начала файла. Метод seekg() устанавливает позицию курсора чтения, а seekp() — позицию курсора записи. Прототипы методов:

```cpp
istream &seekg(pos_type Pos);
istream &seekg(off_type Offset, ios::seekdir Origin);
ostream &seekp(pos_type Pos);
ostream &seekp(off_type Offset, ios::seekdir Origin);
```

В параметре Origin могут быть указаны следующие значения:

- `ios::beg` — начало файла;
- `ios::cur` — текущая позиция курсора;
- `ios::end` — конец файла.

Считаем из файла вначале пятый символ, далее первый символ, а затем последний символ и выведем их в окно консоли (листинг 12.11).

**Листинг 12.11. Файлы произвольного доступа**

```cpp
#include <fstream>
#include <iostream>

int main() {
    char ch = 0;
    std::ifstream file("file.txt", std::ios::in | std::ios::binary);
    if (!file) {
        std::cout << "Error" << std::endl;
        std::cin.get();
        return 1;
    }
    file.seekg(4, std::ios::cur); // Перемещаем курсор относительно текущей позиции
    ch = file.get();
}
```

std::cout << ch << std::endl;
file.seekg(0, std::ios::beg); // Перемещаем курсор в начало
ch = file.get();
std::cout << ch << std::endl;
file.seekg(-1, std::ios::end); // Перемещаем курсор в конец
ch = file.get();
std::cout << ch << std::endl;
file.close();
std::cin.get();
return 0;

**Проверка состояния потока**

После каждой операции с потоком производится установка одного из флагов:

- `ios::goodbit` — ошибок нет;
- `ios::eofbit` — достигнут конец файла;
- `ios::failbit` — последняя операция окончилась неудачей. После сброса флага поток готов к дальнейшим операциям;
- `ios::badbit` — критическая ошибка в потоке.

Проверить состояние флагов позволяют следующие методы:

- `rdstate()` — возвращает текущее состояние потока. Вместо метода `rdstate()` удобнее использовать методы `good()`, `eof()`, `fail()` и `bad()`. Прототип метода:

  ```cpp
  iostate rdstate() const;
  ```

  Проверить успешность выполнения операции можно следующим образом:

  ```cpp
  if (file.rdstate() == std::ios::goodbit) {
      std::cout << "goodbit" << std::endl;
  }
  ```

  Проверка остальных флагов производится с использованием побитовых операторов. Например, проверить, достигнут ли конец файла, можно так:

  ```cpp
  if ((int)file.rdstate() & (int)std::ios::eofbit) {
      std::cout << "eofbit" << std::endl;
  }
  ```

→ good() — возвращает значение true, если установлен флаг ios::goodbit, и false — в противном случае. Прототип метода:
bool good() const;

→ eof() — возвращает значение true, если достигнут конец файла, и false — в противном случае. Прототип метода:
bool eof() const;

→ fail() — возвращает значение true, если установлен флаг ios::failbit, и false — в противном случае. Прототип метода:
bool fail() const;

Проверить установку флага ios::failbit можно также указав поток внутри логического выражения в операторе ветвления. Если флаг установлен, то поток будет иметь значение false. Пример проверки:
if (!file) {
    std::cout << "failbit" << std::endl;
}

Обратите внимание на то, что в VC++ флаг ios::failbit устанавливается также при достижении конца файла вместе с флагом ios::eofbit. Поэтому поток будет иметь значение false не только в случае ошибки;

→ bad() — возвращает значение true, если установлен флаг ios::badbit, и false — в противном случае. Прототип метода:
bool bad() const;

Установить флаг из программы позволяет метод setstate(). Прототип метода:
void setstate(iostate State, bool Reraise = false);

Пример проверки и установки флага ios::failbit:
if (file.fail()) std::cout << "failbit" << std::endl;
else std::cout << "No failbit" << std::endl;
file.setstate(std::ios::failbit);
if (file.fail()) std::cout << "failbit" << std::endl;
else std::cout << "No failbit" << std::endl;

При установке флагов старые флаги не сбрасываются. Чтобы сбросить все флаги ошибок следует вызвать метод clear() без параметров. Прототип метода:
void clear(iostate State = goodbit, bool Reraise = false);

Листинги на странице http://unicross.narod.ru/cpp/
Если в параметре `State` указан флаг, отличный от флага `ios::goodbit`, то вместо сбрасывания флага производится его установка. Пример:

```cpp
file.clear();  // Сбрасываем все флаги ошибок
file.clear(std::ios::failbit |
           std::ios::eofbit);  // Устанавливаем флаги failbit и eofbit
std::cout << file.good() << std::endl;  // 0
std::cout << file.eof() << std::endl;   // 1
std::cout << file.fail() << std::endl;  // 1
std::cout << file.bad() << std::endl;   // 0
```

По умолчанию установка флагов происходит без генерации исключения. Метод `exceptions()` позволяет включить генерацию исключения для заданных флагов. Прототипы метода:

```cpp
iostream exceptions() const;
void exceptions(iostate Newexcept);
```

Первый прототип возвращает флаги, для которых включен режим генерации исключения. Второй прототип устанавливает флаги. При установке нескольких флагов используется побитовый оператор `|`. Объектом исключения является экземпляр класса `ios::failure`. Этот класс является наследником класса `exception`, поэтому получить описание ошибки можно с помощью метода `what()`. В качестве примера включим генерацию исключения для флага `ios::failbit` и обработаем исключение (листинг 12.12).

### Листинг 12.12. Использование метода `exceptions()`

```cpp
#include <fstream>
#include <iostream>
#include <locale>

int main() {
    std::setlocale(LC_ALL, ".1251");
    char ch = 0;
    std::ifstream file;
    std::cout << file.exceptions() << std::endl;  // 0
    file.exceptions(std::ios::failbit);
    std::cout << file.exceptions() << std::endl;  // 2
    try {

    }
```

Листинги на странице http://unicross.narod.ru/cpp/
```cpp
file.open("file871.txt", std::ios::in | std::ios::binary);
}
catch (std::ios::failure &err) {
    std::cout << "Не удалось открыть файл" << std::endl;
    std::cout << err.what() << std::endl;
    std::cin.get();
    return 1;
}
file.close();
std::cin.get();
return 0;
}
```
Если файла не существует или его нельзя открыть на чтение, то результат будет следующим:
0
2
Не удалось открыть файл
ios_base::failbit set

**Классы basic_istream, basic_ostringstream и basic_stringstream**

Классы basic_istream, basic_ostringstream и basic_stringstream позволяют работать со строкой, как с потоком. Класс basic_istream предназначен для чтения данных из строки, класс basic_ostringstream — для записи данных в строку, а класс basic_stringstream позволяет как читать из строки, так и записывать в строку. Прежде чем использовать классы basic_istream, basic_ostringstream и basic_stringstream необходимо подключить заголовочный файл sstream:

```cpp
#include <sstream>
```

Иерархия наследования для ввода данных выглядит так:

ios_base - basic_ios - basic_istream - basic_istream

Иерархия наследования для вывода данных:

ios_base - basic_ios - basic_ostringstream - basic_ostringstream

Иерархия наследования для ввода и вывода данных:

ios_base - basic_ios - (basic_istream, basic_ostringstream) - basic_iostream

- basic_stringstream

Листинги на странице http://unicross.narod.ru/cpp/
Для удобства использования шаблонных классов в заголовочном файле определяются более короткие имена с помощью оператора typedef:

typedef basic_istreamstream<char, char_traits<char>, allocator<char> > istreamstream;
typedef basic_ostringstream<char, char_traits<char>, allocator<char> > ostringstream;
typedef basic_stringstream< char, char_traits<char>, allocator<char> > stringstream;
typedef basic_istreamstream<wchar_t, char_traits<wchar_t>,
                             allocator<wchar_t> > wistreamstream;
typedef basic_ostringstream<wchar_t, char_traits<wchar_t>,
                             allocator<wchar_t> > wostreamstream;
typedef basic_stringstream< wchar_t, char_traits<wchar_t>,
                             allocator<wchar_t> > wostreamstream;

В последующих примерах мы будем рассматривать только классы для работы с обычными строками. Работа с расширенными строками выполняется аналогично.

При создании экземпляра класса чаще всего используются четыре формата:

<Класс> <Переменная>;<
<Класс> <Переменная>{{<Режим>}};
<Класс> <Переменная>{{<С-строика}[, <Режим>]}};
<Класс> <Переменная>{{<Объект класса basic_string>[, <Режим>]}};

Название класса зависит от необходимой операции со строкой:

- istreamstream — только чтение;
- ostringstream — только запись;
- stringstream — одновременно чтение и запись.

Параметр <Режим> задает режим открытия. Используются те же режимы, что и при работе с файлами. Если параметр не указан, то используются следующие режимы по умолчанию:

- istreamstream — ios::in;
- ostringstream — ios::out;
- stringstream — ios::in | ios::out.

При использовании первого формата создается объект с режимом по умолчанию:

Листинги на странице http://unicross.narod.ru/cpp/
std::ostringstream str;
str << "String"; // Записываем строку
std::cout << str.str() << std::endl; // String

Второй формат позволяет указать режим открытия:
std::ostringstream str(std::ios::out);
str << "String"; // Записываем строку
std::cout << str.str() << std::endl; // String

Третий формат копирует C-строку в объект:
std::ostringstream str("String", std::ios::out | std::ios::ate);
str << "123";
std::cout << str.str() << std::endl; // String123

Четвертый формат копирует строку из экземпляра класса basic_string:
std::string s("String");
std::ostringstream str(s, std::ios::out | std::ios::ate);
str << "123";
std::cout << str.str() << std::endl; // String123
std::cout << s << std::endl; // String

Записать строку из экземпляра класса basic_string после создания объекта, а также
получить строковое представление объекта позволяет метод str(). Прототипы метода:
void str(const basic_string &NewStr);
basic_string str() const;

Первый прототип очищает содержимое объекта и записывает в него строку из
экземпляра класса basic_string. Второй прототип возвращает экземпляр класса
basic_string. Пример:
std::string s1, s2("Hello");
std::ostringstream str;
str << "String";
std::cout << str.str() << std::endl; // String
s1 = str.str();
std::cout << s1 << std::endl; // String
str.str(s2);
std::cout << str.str() << std::endl; // Hello

Для записи в поток используется оператор <<, а также методы put() и write().
Чтение производится с помощью оператора >>, а также методов get(), getline() и

Листинги на странице http://unicross.narod.ru/cpp/
read(). Кроме того, можно перемещаться внутри потока и проверять состояние потока. Все эти операторы и методы мы уже рассматривали при изучении работы с файлами, поэтому повторно их описывать не будем. Основные операции показаны в листинге 12.13.

Листинг 12.13. Основные операции при работе со строкой, как с потоком

```cpp
#include <iostream>
#include <string>
#include <sstream>

int main() {  
    char s[50] = {0}, ch = 0;
    int x = 0;
    std::string s2(""');
    std::stringstream str;
    str << "String " << 250; // Записываем данные
    str.put('\.'); // Записываем символ
    str.write("abc", 4); // Записываем строку
    std::cout << str.str() << std::endl; // String 250. abc
    str.seekg(0, std::ios::beg);
    str >> s >> x; // Считываем данные
    std::cout << s << std::endl; // String
    std::cout << x << std::endl; // 250
    ch = str.get(); // Считываем символ
    str.getline(s, 5, '\b'); // Считываем строку
    std::cout << ch << std::endl; // .
    std::cout << s << std::endl; // a
    str.read(s, 1); // Считываем один символ
    s[1] = '\0'; // Вставляем нулевой символ
    std::cout << s << std::endl; // c
    str.seekg(0, std::ios::beg); // Ставим курсор в начало
    while (!str.get(ch).eof()) { // Посимвольное считывание
        std::cout << ch << std::endl;
    }
    str.clear(); // Очищаем от ошибок
}
```

Листинг на странице http://unicross.narod.ru/cpp/
s2 = str.str();          // Преобразуем в объект
    // класса string
std::cout << s2 << std::endl;   // String 250. abc
std::cin.get();
return 0;
}

Считывание данных из буфера

Для ускорения работы данные вначале помещаются в буфер. Когда буфер заполнится
производится вывод данных на консоль или в файл. При вводе данных из консоли
нажатие клавиши <Enter> приводит к считыванию данных в буфер. Если количество
введенных данных больше требуемого количества, то лишние данные остаются в
буфере и доступны для последующих операций чтения. Например, считаем два
символа:
char ch1 = 0, ch2 = 0;
std::cout << "ch1 = " << std::flush;
std::cin >> ch1;          // Вводим: ab
std::cout << ch1 << std::endl;  // a
std::cout << "ch2 = " << std::flush;
std::cin >> ch2;          // Считывается из буфера
std::cout << ch2 << std::endl;  // b

Если при вводе первого символа вместо одного символа ввести сразу два и нажать
клавишу <Enter>, то в буфер запишутся три символа 'a', 'b' и '\n'. Символ 'a'
записывается в переменную ch1, символ 'b', не дожидаясь ввода пользователя,
считывается из буфера и записывается в переменную ch2, а символ '\n' остается в
буфере. Результат выполнения примера в этом случае выглядит так:
ch1 = ab
a
ch2 = b

Если вводить символы отдельно, то результат будет другим:
ch1 = a
a
ch2 = b
b

Листинги на странице http://unicross.narod.ru/cpp/
Как видно из примера результаты отличаются. Это может запутать пользователя, если в первой операции ввода второй символ был введен случайно. Кроме того, символ 'n' может стать причиной закрытия окна консоли, если задержка выполнялась с помощью инструкции:

```cpp
std::cin.get();
```

Проигнорировать определенное количество символов, находящихся в буфере, позволяет метод `ignore()`. Прототип метода:

```cpp
istream &ignore(streamsize Count=1, int_type Metadelim=EOF);
```

Первый параметр задает максимальное количество считываемых символов (по умолчанию один символ). Второй параметр задает символ-разделитель. Если символ-разделитель встретится ранее, чем будет почитано указанное количество символов, то считывание прекращается. В этом случае символ-разделитель также считывается из потока. Переделаем предыдущий пример и сделаем так, чтобы каждый раз считывалось только по одному символу:

```cpp
char ch1 = 0, ch2 = 0;
std::cout << "ch1 = " << std::flush;
std::cin >> ch1; // Вводим: ab
std::cout << ch1 << std::endl; // a
std::cin.ignore(200, '\n');
std::cout << "ch2 = " << std::flush;
std::cin >> ch2; // Вводим: b
std::cout << ch2 << std::endl; // b
std::cin.ignore(200, '\n');
```

Результат выполнения примера в этом случае выглядит так:

```cpp
ch1 = ab
a
ch2 = b
b
```

Число 200 в первом параметре метода `ignore()` является произвольным значением. Чтобы считать все символы, содержащиеся в буфере, то можно воспользоваться методом `readsome()`. Прототип метода:

```cpp
streamsize readsome(char *Str, streamsize Count);
```

В первом параметре передается указатель на символьный массив, в котором будут сохранены считанные символы, а во втором параметре задается количество символов. Нулевой символ в конец символьного массива автоматически не вставляется. Метод
возвращает количество считанных символов или значение 0, если символов в буфере больше нет. Переделаем наш пример и используем метод `readsome()` вместо метода `ignore()`:

```cpp
char ch1 = 0, ch2 = 0, buf[5] = {0};
std::cout << "ch1 = " << std::flush;
std::cin >> ch1;     // Вводим: abcd
std::cout << ch1 << std::endl;   // a
while (std::cin.readsome(buf, 5)) {}
std::cout << "ch2 = " << std::flush;
std::cin >> ch2;     // Вводим: b
std::cout << ch2 << std::endl;   // b
while (std::cin.readsome(buf, 5)) {}
```

Обратите внимание на инструкции, которые выводят подсказки пользователю. После вывода подсказки вызывается манипулятор `flush`, который сбрасывает содержимое буфера. При использовании некоторых компиляторов буфер сбрасывается только после вывода символа `\n`. Поэтому можно не дождаться вывода подсказки вовсе. Чтобы избежать этого следует сбрасывать буфер явным образом с помощью манипулятора `flush`:

```cpp
std::cout << "ch1 = " << std::flush;
```

или метода `flush()`:

```cpp
std::cout << "ch1 = ";
std::cout.flush();
```

Вместо вызова манипулятора `flush` или метода `flush()` можно синхронизировать потоки `cin` и `cout` с помощью метода `tie()`. В этом случае прежде чем получать данные от пользователя будет автоматически произведен сброс буфера объекта `cout`. По умолчанию в VC++ потоки `cin` и `cout` синхронизированы. Прототип метода:

```cpp
ostream *tie(ostream *Newtie);
```

Пример:

```cpp
char ch = 0;
std::cin.tie(&std::cout);    // Синхронизируем потоки
std::cout << "ch = ";
std::cin >> ch;              // Вводим: a
std::cout << ch << std::endl;  // a
```

Листинги на странице http://unicross.narod.ru/cpp/
Настройка локали для потока

Для каждого потока можно настроить свою локаль. По умолчанию для потоков настроена локаль с. Чтобы настроить другую локаль для конкретного потока следует передать объект класса locale в метод imbue(). Прототип метода:

```cpp
locale imbue(const locale &Loc);
```
Метод возвращает объект с предыдущими настройками локали. Для получения текущих настроек локали предназначен метод getloc(). Прототип метода:

```cpp
locale getloc() const;
```
Пример настройки локали для объекта cout приведен в листинге 12.14.

Листинг 12.14. Настройка локали для потока

```cpp
#include <iostream>
#include <locale>

int main() {
    std::locale loc("Russian_Russia.1251");
    std::locale::global(loc); // Глобальная настройка локали
    std::cout.imbue(loc);      // Настройка локали для потока
    std::cout << std::cout.getloc().c_str() << std::endl;
    // Russian_Russia.1251
    std::cout << "Этот текст корректно отображается в окне консоли"
               << std::endl;
    std::cin.get();
    return 0;
}
```

Форматированный ввод/вывод в языке C++

При выводе (в некоторых случаях и при вводе) данных можно произвести форматирование, например, указать ширину поля, выравнивание внутри поля, количество цифр после десятичной точки и т.д. Для этого предварительно следует установить соответствующие флаги с помощью определенных методов или специальных функций-манипуляторов. В этом разделе мы рассмотрим форматированный ввод/вывод на примере объектов cout и cin. Форматированный вывод в файл осуществляется аналогичным образом.

Листинги на странице http://unicross.narod.ru/cpp/
Методы `fill()`, `precision()` и `width()`

Для указания минимальной ширины поля предназначен метод `width()`. Ширина устанавливается только для последующей операции вывода. После выполнения операции значение сбрасывается. Если указанная ширина поля не достаточна для отображения данных, то значение игнорируется и данные выводятся полностью. Прототипы метода:

```cpp
streamsize width() const;
streamsize width(streamsize NewWidth);
```

Первый прототип метода позволяет определить текущее значение. Пример:

```cpp
std::cout << std::cout.width() << std::endl; // 0
std::cout.width(10);
std::cout << std::cout.width() << std::endl; // 10
std::cout << std::cout.width() << std::endl; // 0
```

Второй прототип предназначен для указания минимальной ширины. Метод возвращает предыдущее значение. Пример:

```cpp
std::cout << '\';
std::cout.width(10);
std::cout << "abc";
std::cout << '\' << std::endl; // Результат: ' abc'
```

Внутри поля данные выравниваются по правому краю, а перед данными по умолчанию добавляются пробелы. Метод `fill()` позволяет задать другой символ. Прототипы метода:

```cpp
char fill() const;
char fill(char Newfill);
```

Первый прототип возвращает текущий символ-заполнитель. Второй прототип предназначен для указания другого символа. Метод возвращает предыдущий символ-заполнитель. Пример:

```cpp
std::cout << '\';
std::cout.width(10);
std::cout.fill('*');
std::cout << "abc";
std::cout << '\' << std::endl; // Результат: '******abc'
```

Метод `precision()` предназначен для указания точность для чисел. Если количество цифр в числе больше указанного значения, то производится округление. Обратите

Листинги на странице http://unicross.narod.ru/cpp/
внимание, чтобы указать количество цифр после десятичной точки для вещественных чисел необходимо дополнительно установить флаг fixed, например, с помощью одноименного манипулятора. Прототипы метода:
streamsize precision() const;
streamsize precision(streamsize Newprecision);
Первый прототип возвращает текущее значение. Второй прототип предназначен для указания нового значения. Метод возвращает предыдущее значение. Пример:
std::cout << std::cout.precision()
   << std::endl; // 6
std::cout << 3.14159265359 << std::endl; // 3.14159
std::cout.precision(1);
std::cout << 123.1234567 << std::endl; // 1e+002
std::cout.precision(3);
std::cout << 123.1234567 << std::endl; // 123
std::cout.precision(6);
std::cout << 123.12367 << std::endl; // 123.124
std::cout << std::fixed << 123.1234567
   << std::endl; // 123.123457

Установка и сброс флагов формата
В предыдущем разделе мы рассмотрели возможность указания ширины поля, символа-заполнителя и количества цифр в числе. Остальные параметры форматирования задаются с помощью установки соответствующих флагов. Для установки и сброса флагов предназначены следующие методы:

flags() — устанавливает указанные в параметре флаги, а все остальные сбрасывает. Прототипы метода:
fmtflags flags() const;
fmtflags flags(fmtflags Newfmtflags);
Первый прототип возвращает текущее значение. Второй прототип устанавливает указанные флаги. Чтобы указать сразу несколько флагов следует между ними вставить побитовый оператор |. Пример:
std::cout << std::cout.flags() << std::endl; // 513
std::cout.flags(std::ios_base::hex | std::ios_base::left);
std::cout.width(10);

Листинги на странице http://unicross.narod.ru/cpp/
std::cout.fill('*');
std::cout << 14 << std::endl; // e**********
std::cout << std::cout.flags() << std::endl; // 840

setf() — устанавливает флаги. Прототипы метода:

fmtflags setf(fmtflags Newfmtflags);
fmtflags setf(fmtflags Newfmtflags, fmtflags Mask);

Первый прототип устанавливает указанные флаги. Флаги, установленные ранее, не сбрасываются. Метод возвращает предыдущее значение. Пример:

std::cout << std::cout.flags() << std::endl; // 513
std::cout.setf(std::ios_base::left);
std::cout.width(10);
std::cout.fill('*');
std::cout << 14 << std::endl; // 14**********
std::cout << std::cout.flags() << std::endl; // 577

Второй прототип производит вычисление значения следующим образом:

(<Текущее значение> & ~Mask) | (Newfmtflags & Mask & 0xffff)

Пример установки флагов hex и left:

std::cout << std::cout.flags() << std::endl; // 513
std::cout.setf(std::ios_base::hex | std::ios_base::left,
               std::ios_base::basefield | std::ios_base::adjustfield);
std::cout.width(10);
std::cout.fill('*');
std::cout << 14 << std::endl; // e**********
std::cout << std::cout.flags() << std::endl; // 841

unsetf() — сбрасывает указанные флаги. Прототип метода:

void unsetf(fmtflags mask);

В методах flags(), setf() и unsetf() можно указать следующие флаги (или их комбинацию через побитовые операторы):

boolalpha — позволяет вводить и выводить логические значения true и false.

По умолчанию выводятся числа 1 и 0 соответственно. Пример:

Листинг на странице http://unicross.narod.ru/cpp/
bool is = true;
std::cout << is << std::endl; // 1
std::cout.flags(std::ios_base::boolalpha);
std::cout << is << std::endl; // true

dec — десятичное отображение числа:
std::cout.flags(std::ios_base::dec);
std::cout << 10 << std::endl; // 10

oc — восьмеричное отображение числа:
std::cout.flags(std::ios_base::oct);
std::cout << 10 << std::endl; // 12

hex — шестнадцатеричное отображение числа:
std::cout.flags(std::ios_base::hex);
std::cout << 10 << std::endl; // a

basefield — комбинация флагов dec | hex | oct. Обычно указывается во втором параметре метода setf(). Пример:
std::cout.setf(std::ios_base::hex,
               std::ios_base::basefield);
std::cout << 14 << std::endl; // e

showbase — для восьмеричных значений добавляет в начало символ "0", для шестнадцатеричных значений добавляет комбинацию символов "0x":
std::cout.flags(std::ios_base::oct | std::ios_base::showbase);
std::cout << 10 << std::endl; // 012
std::cout.flags(std::ios_base::hex | std::ios_base::showbase);
std::cout << 10 << std::endl; // 0xa

showpos — выводит знак перед положительным числом:
std::cout.flags(std::ios_base::showpos);
std::cout << 10 << std::endl; // +10

showpoint — выводит десятичную точку и нули для вещественных чисел:
double x = 10.;
std::cout << x << std::endl; // 10

Листинги на странице http://unicross.narod.ru/cpp/
std::cout.flags(std::ios_base::showpoint);
std::cout << x << std::endl; // 10.0000

- **scientific** — вещественное число в экспоненциальной форме:
  ```c++
double x = 100.;
std::cout.flags(std::ios_base::scientific);
std::cout << x << std::endl; // 1.000000e+002
```

- **fixed** — при указании вместе с методом `precision()` позволяет задать количество цифр после десятичной точки. Если флаг не указан, то метод `precision()` задает количество цифр в числе. Пример:
  ```c++
std::cout.precision(6);
std::cout << 123.1234567 << std::endl; // 123.123
std::cout.flags(std::ios_base::fixed);
std::cout.precision(2);
std::cout << 123.1234567 << std::endl; // 123.12
std::cout.precision(6);
std::cout << 123.1234567 << std::endl; // 123.123457
```

- **floatfield** — комбинация флагов `fixed` | `scientific`. Обычно указывается во втором параметре метода `setf()`. Пример:
  ```c++
double x = 100.;
std::cout.setf(std::ios_base::scientific,
               std::ios_base::floatfield);
std::cout << x << std::endl; // 1.000000e+002
```

- **uppercase** — выводит буквы в шестнадцатеричном числе в верхнем регистре. Кроме того, в верхнем регистре выводится "x" при отображении основания счисления и буква "e" при выводе вещественного числа в экспоненциальной форме. Пример:
  ```c++
std::cout.flags(std::ios_base::hex | std::ios_base::showbase
                | std::ios_base::uppercase);
std::cout << 10 << std::endl; // 0XA
std::cout.flags(std::ios_base::scientific |
                std::ios_base::uppercase);
```

Листинги на странице [http://unicross.narod.ru/cpp/]
std::cout << 100. << std::endl; // 1.000000E+002

right — выравнивание по правому краю поля (по умолчанию):
std::cout << '\'
std::cout.width(10);
std::cout.flags(std::ios_base::right);
std::cout << "abc"
std::cout << '\' << std::endl; // Результат: ' abc'

left — выравнивание по левому краю поля:
std::cout << '\'
std::cout.width(10);
std::cout.flags(std::ios_base::left);
std::cout << "abc"
std::cout << '\' << std::endl; // Результат: 'abc'

internal — знак числа выравнивается по левому краю, а само число выравнивается по правому краю. Между знаком и числом вставляются символы-заполнители (по умолчанию пробелы). С помощью метода setf() можно задать другой символ-заполнитель. Пример:
std::cout << '\'
std::cout.width(10);
std::cout.flags(std::ios_base::internal);
std::cout << -10;
std::cout << '\' << std::endl; // Результат: '- 10'

adjustfield — комбинация флагов internal | left | right. Обычно указывается во втором параметре метода setf(). Пример:
std::cout << '\'
std::cout.width(10);
std::cout.fill('*');
std::cout.setf(std::ios_base::internal,
            std::ios_base::adjustfield);
std::cout << -10;
std::cout << '\' << std::endl; // Результат: '-*********10'

Листинги на странице http://unicross.narod.ru/cpp/
→ unitbuf — сбрасывает содержимое буфера после каждой операции вывода;

→ skipws — если флаг установлен, то при вводе данных пробельные символы игнорируются (по умолчанию флаг установлен). Введем один символ, а затем сбросим флаг skipws и запросим ввод еще одного символа. После ввода первого символа в буфере остается символ перевода строки. Именно этот символ будет считан при сброшенном флаге skipws во второй операции ввода. Пример:

```cpp
char ch = 0;
std::cin >> ch;    // Вводим a
std::cout << ch << std::endl;    // a
std::cin.unsetf(std::ios_base::skipws);
std::cin >> ch;    // Символ \n считывается из буфера
std::cout << (int)ch << std::endl;    // 10 (символ \n)
```

### Манипуляторы формата

Установить и сбросить флаги формата можно не только с помощью методов flags(), setf() и unsetf(), но с помощью манипуляторов. Манипуляторы формата являются функциями, принимающими и возвращающими ссылку на поток, поэтому их можно указывать после операторов << и >>. Вначале рассмотрим манипуляторы, принимающие дополнительный параметр:

→ resetiosflags(fmtflags Flags) — сбрасывает указанные флаги;

→ setiosflags(fmtflags Flags) — устанавливает указанные флаги. Пример:

```cpp
#include <iomanip>

// ... фрагмент опущен ...
std::cout << std::resetiosflags(std::ios_base::dec)
    << std::setiosflags(std::ios_base::hex)
    << 14 << std::endl;    // e
```

→ setw(streamsize Newwidth) — задает минимальную ширину поля;

→ setfill(char Ch) — задает символ-заполнитель. Пример:

```cpp
std::cout << std::setw(10) << std::setfill('*')
    << "abc" << std::endl;    // ******abc
```

Листинги на странице http://unicross.narod.ru/cpp/
setprecision(streamsize Newprecision) — указывает точность для чисел. Если количество цифр в числе больше указанного значения, то производится округление. Обратите внимание, чтобы указать количество цифр после десятичной точки для вещественных чисел необходимо дополнительно установить флаг fixed.
Пример:

```cpp
std::cout << std::setprecision(6)
   << 123.1234567 << std::endl; // 123.123
std::cout << std::setprecision(6) << std::fixed
   << 123.1234567 << std::endl; // 123.123457
```

setbase(int Base) — задает систему счисления для чисел:

```cpp
std::cout << std::setbase(10) << 10 << std::endl; // 10
std::cout << std::setbase(8) << 10 << std::endl; // 12
std::cout << std::setbase(16) << 10 << std::endl; // a
```

**Обратите внимание**

Для использования манипуляторов, принимающих дополнительный параметр, необходимо подключить заголовочный файл iomanip.

Перечислим остальные манипуляторы:

- endl — выводит символ перевода строки и сбрасывает буфер:
  ```cpp
  std::cout << 10 << std::endl;
  ```

- ends — выводит нулевой символ;

- flush — сбрасывает буфер:
  ```cpp
  std::cout << 10 << std::flush;
  ```

- boolalpha — устанавливает флаг boolalpha:
  ```cpp
  bool is = true;
  std::cout << std::boolalpha << is << std::endl; // true
  ```

- noboolalpha — сбрасывает флаг boolalpha;

- dec — устанавливает флаг dec:
  ```cpp
  std::cout << std::dec << 10 << std::endl; // 10
  ```

- oct — устанавливает флаг oct:
  ```cpp
  std::cout << std::oct << 10 << std::endl; // 12
  ```

Листинги на странице http://unicross.narod.ru/cpp/
hex — устанавливает флаг hex:
    std::cout << std::hex << 10 << std::endl; // a

decimal — устанавливает флаг showbase:
    std::cout << std::oct << std::showbase
                   << 10 << std::endl; // 012
    std::cout << std::hex << std::showbase
                   << 10 << std::endl; // 0xA

noshowbase — сбрасывает флаг showbase;
noshowpos — сбрасывает флаг showpos;
showpoint — устанавливает флаг showpoint:
    double x = 10.;
    std::cout << std::showpoint << x << std::endl; // 10.00000

noshowpoint — сбрасывает флаг showpoint;
scientific — устанавливает флаг scientific:
    double x = 100.;
    std::cout << std::scientific << x; // 1.000000e+02

fixed — устанавливает флаг fixed:
    std::cout << std::fixed << std::setprecision(2)
                   << 123.1234567 << std::endl // 123.12
                   << std::setprecision(6)
                   << 123.1234567 << std::endl; // 123.123457

defaultfloat — сбрасывает флаги fixed и scientific;
uppercase — устанавливает флаг uppercase:
    std::cout << std::hex << std::showbase << std::uppercase
                   << 10 << std::endl; // 0xA
    std::cout << std::scientific << std::uppercase
                   << 100. << std::endl; // 1.000000E+002

Листинги на странице http://unicross.narod.ru/cpp/
→ nouppercase — сбрасывает flag uppercase;
→ right — устанавливает flag right;
→ left — устанавливает flag left;
→ internal — устанавливает flag internal;
→ unitbuf — устанавливает flag unitbuf;
→ nounitbuf — сбрасывает flag unitbuf;
→ skipws — устанавливает flag skipws;
→ noskipws — сбрасывает flag skipws.

Создание пользовательских манипуляторов

Создание манипулятора вывода осуществляется следующим образом:

```cpp
std::ostream &<Название>(std::ostream &stream) {
    // ...
    return stream; // Возвращаем ссылку на поток
}
```

Создание манипулятора ввода выглядит так:

```cpp
std::istream &<Название>(std::istream &stream) {
    // ...
    return stream; // Возвращаем ссылку на поток
}
```

В качестве параметров функции принимают ссылку на соответствующий поток. Внутри функции необходимо вернуть ссылку на поток, чтобы иметь возможность составлять цепочки из вызовов. Пример создания пользовательского манипулятора вывода показан в листинге 12.15.

### Листинг 12.15. Создание манипулятора вывода

```cpp
#include <iostream>

std::ostream &my_manip(std::ostream &stream) {
    stream << "Hello, C++";
    return stream;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
int main() {
    std::cout << my_manip << std::endl; // Hello, C++
    std::cin.get();
    return 0;
}

Реализация манипулятора, принимающего один параметр, зависит от компилятора. В компиляторе, который входит в VC++ 2010, такие манипуляторы реализуются с помощью шаблонной структуры _Smanip. Пример создания манипуляторов с одним параметром показан в листинге 12.16.

### Листинг 12.16. Создание манипулятора с одним параметром

```cpp
#include <iostream>
#include <iomanip>

void func1(std::ios_base &stream, int x) {
    std::ostream *p = dynamic_cast<std::ostream *>(&stream);
    if (p) {
        (*p) << "x = " << x;
    }
}

void func2(std::ios_base &stream, char *str) {
    std::ostream *p = dynamic_cast<std::ostream *>(&stream);
    if (p) {
        (*p) << str;
    }
}

std::_Smanip<int>
__cdecl my_manip1(int x) {
    return (std::_Smanip<int>(&func1, x));
}

std::_Smanip<char *>
__cdecl my_manip2(char *str) {
    return (std::_Smanip<char *>(&func2, str));
}
```

Листинги на странице http://unicross.narod.ru/cpp/
int main() {
    std::cout << my_manip1(100) << std::endl; // x = 100
    std::cout << my_manip2("Hello") << std::endl; // Hello
    std::cin.get();
    return 0;
}

Работа с файловой системой

В предыдущих разделах были описаны способы работы с файлами в языках C и C++. Теперь мы переходим к рассмотрению вопросов переименования и удаления файлов, получения информации о файлах, а также созданию, удалению и чтению каталогов. Следует сразу заметить, что большинство функций, описанных в этом разделе, не входят в стандарт языка C, ни в стандарт языка C++. Реализация этих функций зависит от операционной системы и компилятора. Описанные в этом разделе функции применимы для операционной системы Windows и используются в компиляторе, входящем в состав Microsoft Visual C++ 2010.

Преобразование пути к файлу или каталогу

Преобразовать путь к файлу или каталогу в VC++ позволяют следующие функции:

➔ _fullpath() и _wfullpath() — преобразуют относительный путь в абсолютный путь, учитывая местоположение текущего рабочего каталога. Прототипы функций:

```c
#include <stdlib.h>
char *fullpath(char *FullPath, const char *Path,
               size_t SizeInBytes);
wchar_t *wfullpath(wchar_t *FullPath, const wchar_t *Path,
                   size_t SizeInWords);
```

В первом параметре передается указатель на буфер, в который будет записан абсолютный путь. Если в втором параметре передать нулевой указатель, то память под абсолютный путь будет выделена динамически с помощью функции malloc(). В этом случае после использования памяти ее следует освободить с помощью функции free(). Во втором параметре указывается относительный путь, а в третьем параметре — максимальный размер буфера. В качестве максимального размера буфера обычно указывается макрос _MAX_PATH. Определение макроса выглядит так:

Листинги на странице http://unicross.narod.ru/cpp/
Пример преобразования относительного пути в абсолютный путь:
char full_path[_MAX_PATH] = {0}, *p = 0;
p = _fullpath(full_path, "test.txt", _MAX_PATH);
if (p) {
    std::cout << full_path << std::endl;
} // Результат: C:\book\test\test\test.txt
p = _fullpath(0, "..\..\..\test.txt", _MAX_PATH);
if (p) {
    std::cout << p << std::endl; // C:\book\test.txt
    std::free(p);
}

_splitpath_s() и _wsplitpath_s() — разбивают абсолютный путь на составляющие: имя диска, путь, имя файла и расширение. Прототипы функций:
#include <stdlib.h>
errno_t _splitpath_s(const char *FullPath, char *Drive,
    size_t DriveSize, char *Dir, size_t DirSize,
    char *Filename, size_t FilenameSize,
    char *Ext, size_t ExtSize);
errno_t _wsplitpath_s(const wchar_t *FullPath, wchar_t *Drive,
    size_t DriveSize, wchar_t *Dir, size_t DirSize,
    wchar_t *Filename, size_t FilenameSize,
    wchar_t *Ext, size_t ExtSize);

В параметре FullPath задается абсолютный путь. В параметре Drive передается указатель на буфер, в котором будет сохранено имя диска и двоеточие, а в параметре DriveSize — максимальный размер этого буфера. В параметре Dir передается указатель на буфер, в котором будет сохранен путь, а в параметре DirSize — максимальный размер этого буфера. В параметре Filename передается указатель на буфер, в котором будет сохранено имя файла без расширения, а в параметре FilenameSize — максимальный размер этого буфера. В параметре Ext передается указатель на буфер, в котором будет сохранено расширение файла с предваряющей точкой, а в параметре ExtSize — максимальный размер этого буфера. Функция
возвращает значение 0, если ошибок нет, и код ошибки при неудачном выполнении.
В качестве размеров обычно указываются следующие макросы:

```c
#define _MAX_DRIVE 3
#define _MAX_DIR 256
#define _MAX_FNAME 256
#define _MAX_EXT 256
```

Пример разбиения абсолютного пути на составляющие:

```c
errno_t err = 0;
char full_path[] = "C:\\book\\test.txt";
char drive[_MAX_DRIVE] = {0}, dir[_MAX_DIR] = {0};
char name[_MAX_FNAME] = {0}, ext[_MAX_EXT] = {0};
err = _splitpath_s(full_path, drive, _MAX_DRIVE, dir,
                   _MAX_DIR, name, _MAX_FNAME, ext, _MAX_EXT);
if (err == 0) {
    std::cout << "drive: " << drive << std::endl; // drive: C:
    std::cout << "dir: " << dir << std::endl; // dir: \book\n    std::cout << "name: " << name << std::endl; // name: test
    std::cout << "ext: " << ext << std::endl; // ext: .txt
}
```

Если какая-либо из составляющих не нужна, то в соответствующем параметре следует передать нулевой указатель и в качестве размера указать значение 0. Пример получения только имени файла:

```c
errno_t err = 0;
char full_path[] = "C:\\book\\test.txt";
char name[_MAX_FNAME] = {0};
err = _splitpath_s(full_path, 0, 0, 0,
                   0, name, _MAX_FNAME, 0, 0);
if (err == 0) {
    std::cout << "name: " << name << std::endl; // name: test
}
```

Листинги на странице http://unicross.narod.ru/cpp/
_makepath_s() и _wmakepath_s() — соединяют элементы пути (имя диска, путь, имя файла и расширение) в абсолютный путь. Прототипы функций:

```c
#include <stdlib.h>

errno_t _makepath_s(char *PathResult, size_t SizeInWords,
const char *Drive, const char *Dir,
const char *Filename, const char *Ext);

erro_t _wmakepath_s(wchar_t *PathResult, size_t SIZE,
const wchar_t *Drive, const wchar_t *Dir,
const wchar_t *Filename, const wchar_t *Ext);
```

В первом параметре передается указатель на буфер, в который будет записан абсолютный путь, а во втором параметре указывается максимальный размер буфера. В параметре Drive задается имя диска, в параметре Dir — путь, в параметре Filename — имя файла, а в параметре Ext — расширение файла. Некоторые из этих четырех параметров можно не указывать, в этом случае следует передать нулевой указатель или пустую строку. Функция возвращает значение 0, если ошибок не было, и код ошибки при неудачном выполнении. Пример:

```c
errno_t err = 0;
char full_path[_MAX_PATH] = {0};
err = _makepath_s(full_path, _MAX_PATH, "C", "\book\", 
"test", "txt");
if (err == 0) {
    std::cout << full_path << std::endl; // C:\book\test.txt
}
```

### Переименование и удаление файла

Для переименования и удаление файла предназначены следующие функции:

- rename() — переименовывает файл. В первом параметре указывается старое название файла, а во втором параметре — новое название. Если операция успешно произведена, то функция возвращает значение 0. В противном случае функция возвращает ненулевое значение и глобальной переменной errno присваивается код ошибки. Прототип функции:

```c
#include <cstdio>

int rename(const char *OldFilename, const char *NewFilename);
```

Пример переименования файла:
if (std::rename("file.txt", "file2.txt")) {
    std::cout << "Error" << std::endl;
}
else {
    std::cout << "OK" << std::endl;
}

remove() — удаляет файл, название которого передано в параметре. Если операция успешно произведена, то функция возвращает значение 0. В противном случае функция возвращает значение -1 и глобальной переменной errno присваивается код ошибки. Прототип функции:
#include <cstdio>
int remove(const char *Filename);

Пример удаления файла:
if (std::remove("file2.txt") != 0) {
    std::cout << "Error" << std::endl;
}
else {
    std::cout << "OK" << std::endl;
}

Проверка прав доступа к файлу и каталогу
Для проверки существования файла и каталога, а также возможности чтения и записи файла в VC++ предназначены функции _access() и _waccess(). Прототипы функций:
#include <io.h>
int _access(const char *Filename, int AccessMode);
#include <wchar.h>
int _waccess(const wchar_t *Filename, int AccessMode);
В первом параметре указывается путь к файлу или каталогу, а во втором параметре задается одно из следующих значений:

0 — проверка существования файла или каталога;

Листинги на странице http://unicross.narod.ru/cpp/
2 — проверка возможности записи в файл;
4 — проверка возможности чтения файла;
6 — проверка возможности чтения и записи файла.

В случае успешности проверки функции возвращают значение 0. В противном случае функции возвращают значение -1 и глобальной переменной errno присваивается номер ошибки. Пример проверки доступа к файлу приведен в листинге 12.17.

Листинг 12.17. Проверка доступа к файлу

```cpp
#include <iostream>
#include <io.h>
#include <cstdio>
#include <locale>

int main() {  
    char path[] = "C:\\book\\test.txt";
    std::setlocale(LC_ALL, "");
    if (_access(path, 0) != -1) {
        std::cout << "Файл существует" << std::endl;
        if (_access(path, 2) != -1) {
            std::cout << "Файл доступен на запись" << std::endl;
        }
    } else std::perror("Error");
    if (_access(path, 4) != -1) {
        std::cout << "Файл доступен для чтения" << std::endl;
    } else std::perror("Error");
    if (_access(path, 6) != -1) {
        std::cout << "Файл доступен для чтения и записи"
                   << std::endl;
    } else std::perror("Error");
}
```

Листинги на странице http://unicross.narod.ru/cpp/
Изменение прав доступа к файлу

Для изменения прав доступа к файлу в VC++ предназначены функции _chmod() и _wchmod(). Прототипы функций:

```c
#include <io.h>
int _chmod(const char *Filename, int Mode);
#include <wchar.h>
int _wchmod(const wchar_t *Filename, int Mode);
```

В первом параметре указывается путь к файлу, а во втором параметре задается один из следующих макросов (макросы определены в заголовочном файле sys/stat.h):

- `_S_IREAD` — разрешает только чтение файла;
- `_S_IWRITE` — разрешает запись в файл.

В случае успешности установки прав доступа функции возвращают значение 0. В противном случае функции возвращают значение -1 и глобальной переменной errno присваивается номер ошибки. Пример установки прав доступа только для чтения приведен в листинге 12.18.

```
Листинг 12.18. Установка прав доступа только для чтения

#include <iostream>
#include <io.h>
#include <sys/stat.h>
#include <cstdio>

int main() {
    char path[] = "C:\book\test.txt";
    if (_chmod(path, _S_IREAD) != -1) {
        std::cout << "OK" << std::endl;
    } else std::perror("Error");
    std::cin.get();
    return 0;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
Получение информации о файле

Получить размер файла и время создания, изменения и доступа к файлу, а также значения других метаданных позволяют в VC++ функции _stat(), _stat32(), _stat64(), _stati64(), _stat32i64(), _stat64i32(), _wstat(), _wstat32(), _wstat64(), _wstati64(), _wstat32i64() и _wstat64i32(). По умолчанию в VC++ 2010 функция _stat() эквивалентна функции _stat64i32(). Число 64 в этой функции означает количество бит, выделяемых под дату, а число 32 означает количество бит, выделяемых под размер файла. Прототип функции _stat64i32():

```c
#include <sys/types.h>
#include <sys/stat.h>

int _stat64i32(const char *Name, struct _stat64i32 *Stat);
```

В первом параметре указывается путь к файлу. Результат записывается в структуру stat. Функция возвращает значение 0, если ошибок не было, и значение -1 в случае ошибки. Код ошибки сохраняется в глобальной переменной errno.

Структура _stat64i32 объявлена следующим образом:

```c
define _S_IFMT 0xF000 /* Маска для типа файла */
define _S_IFDIR 0x4000 /* Каталог */
define _S_IFCHR 0x2000 /* Символьное устройство */
define _S_IFIFO 0x1000 /* Канал */
```

Листинги на странице http://unicross.narod.ru/cpp/
Пример использования функции _stat() приведен в листинге 12.19.

Листинг 12.19. Пример использования функции _stat()

```cpp
#include <iostream>
#include <ctime>
#include <sys/types.h>
#include <sys/stat.h>
#include <cstdio>

void print_time(__time64_t &t);

int main() {
    struct _stat info;
    char ch = 0;
    int err = _stat("C:\\book\\test.txt", &info);
    if (err == 0) {
        std::cout << info.st_size << std::endl; // Размер файла
        ch = info.st_dev + 'A';
        std::cout << ch << std::endl; // Имя диска
        print_time(info.st_ctime); // Создание файла
        print_time(info.st_atime); // Последний доступ
        print_time(info.st_mtime); // Изменение файла
    } else std::cerr("Error");
    std::cin.get();
    return 0;
}

void print_time(__time64_t &t) {
    std::tm ptm;
    const short SIZE = 100;
    char str[SIZE] = {0};

    Листинги на странице http://unicross.narod.ru/cpp/
size_t count = 0;
errno_t err = localtime_s(&ptm, &t);
if (!err) {
    count = std::strftime(str, SIZE, "%d.%m.%Y %H:%M:%S", &ptm);
    if (count) {
        std::cout << str << std::endl;
    }
}

Обновить время последнего доступа и время изменения файла в VC++ позволяют функции _utime(), _utime32(), _utime64(), _wutime(), _wutime32() и _wutime64(). Прототип функции _utime():
#include <sys/utime.h>
int _utime(const char *Filename, struct _utimbuf *Utimbuf);

В первом параметре функция принимает указатель на строку, в которой находится путь к файлу. Во втором параметре передается указатель на структуру _utimbuf.
Объявление структуры выглядит так:

struct _utimbuf {
    time_t actime;       /* Дата последнего доступа */
    time_t modtime;      /* Дата изменения */
};

Если во втором параметре передать нулевой указатель, то дата будет текущей. Функция возвращает значение 0, если ошибок не было, и значение -1 в противном случае. Код ошибки сохраняется в глобальной переменной errno. Пример использования функции _utime() приведен в листинге 12.20.

Листинг 12.20. Пример использования функции _utime()

#include <iostream>
#include <ctime>
#include <sys/utime.h>
#include <cstdio>

int main() {
    struct _utimbuf new_time;
    new_time.actime = std::time(0) - 3600;
    
    Листинги на странице http://unicross.narod.ru/cpp/
new_time.modtime = std::time(0) - 7200;
int err = _utime("C:\book\test.txt", &new_time);
if (err == 0) {
    std::cout << "OK" << std::endl;
}
else std::perror("Error");

// Текущее время
err = _utime("C:\book\test.txt", 0);
if (err == 0) {
    std::cout << "OK" << std::endl;
}
else std::perror("Error");
std::cin.get();
return 0;


Функции для работы с дисками и каталогами

Для работы с дисками и каталогами в VC++ используются следующие функции:

➔ `_getdrive()` — возвращает номер текущего диска (1 — диск А, 2 — диск В и т. д.).
Прототип функции:
```
#include <direct.h>
int _getdrive(void);
```
Пример:
```
int ch = 0, drive = 0;
drive = _getdrive();
ch = drive - 1 + 'A';
std::cout << (char)ch << std::endl; // C
```

➔ `_chdrive()` — делает указанный диск текущим. Прототип функции:
```
#include <direct.h>
int _chdrive(int Drive);
```
В параметре Drive указывается номер диска (1 — диск А, 2 — диск В и т. д.).
Функция возвращает значение 0, если ошибок нет, и значение -1 в противном случае. Код ошибки сохраняется в глобальной переменной errno. Пример:

Листинги на странице http://unicross.narod.ru/cpp/


if (_chdrive(4) == 0) {
    std::cout << _getdrive() << std::endl; // 4
}

_getcwd() и _wgetcwd() — позволяют получить строковое представление текущего рабочего каталога. От этого значения зависит преобразование относительного пути в абсолютный. Кроме того, важно помнить, что текущим рабочим каталогом будет каталог, из которого запускается файл, а не каталог с исполняемым файлом. Прототипы функций:

```cpp
#include <direct.h>
char *_getcwd(char *DstBuf, int SizeInBytes);
#include <wchar.h>
wchar_t *_wgetcwd(wchar_t *DstBuf, int SizeInWords);
```

В первом параметре функции принимают указатель на буфер, а во втором параметре — максимальный размер буфера. В качестве максимального размера буфера обычно указывается макрос _MAX_PATH. В случае ошибки функции возвращают нулевой указатель. Пример:

```cpp
char buf[_MAX_PATH] = {0};
_getcwd(buf, _MAX_PATH);
std::cout << buf << std::endl; // C:\book\test\test
```

Если в качестве параметров указаны нулевые значения, то строка создается динамически с помощью функции malloc(). В этом случае функция _getcwd() возвращает указатель на эту строку или нулевой указатель в случае ошибки. После окончания работы со строкой следует освободить память с помощью функции free(). Пример:

```cpp
char *buf = 0;
buf = _getcwd(0, 0);
if (buf) {
    std::cout << buf << std::endl; // C:\book\test\test
    std::free(buf);
    buf = 0;
}
```

Листинги на странице http://unicross.narod.ru/cpp/
-_getdcwd() и _wgetdcwd() — функции аналогичны функциям _getcwd() и _wgetcwd(), но позволяют получить строковое представление текущего рабочего каталога не для текущего диска, а для указанного в первом параметре. Прототипы функций:
#include <direct.h>
char * _getdcwd(int Drive, char *DstBuf, int SizeInBytes);

#include <wchar.h>
wchar_t * _wgetdcwd(int Drive, wchar_t *DstBuf, int SizeInWords);

В первом параметре указывается номер диска (0 — диск по умолчанию, 1 — диск A, 2 — диск B и т. д.). Пример:
char buf[_MAX_PATH] = {0};
_getdcwd(3, buf, _MAX_PATH);
std::cout << buf << std::endl; // C:\book\test\test

-_chdir() и _wchdir() — делают указанный каталог текущим. Функции возвращают значение 0, если ошибок не было, и значение −1 в противном случае. Код ошибки сохраняется в глобальной переменной errno. Прототипы функций:
#include <direct.h>
int _chdir(const char *Path);

#include <wchar.h>
int _wchdir(const wchar_t *Path);

Пример:
char buf[_MAX_PATH] = {0};
if (_chdir("C:\book\") == 0) {
    _getcwd(buf, _MAX_PATH);
    std::cout << buf << std::endl; // C:\book
}

-_mkdir() и _wmkdir() — создают новый каталог. Функции возвращают значение 0, если каталог создан, и значение −1 в противном случае. Код ошибки сохраняется в глобальной переменной errno. Прототипы функций:
#include <direct.h>
int _mkdir(const char *Path);

Листинги на странице http://unicross.narod.ru/cpp/
```c
#include <wchar.h>

int _wmkdir(const wchar_t *Path);

Пример:
if (_mkdir("C:\book\folder1") == 0) {
    std::cout << "OK" << std::endl;
}

_rmdir() и _wrmdir() — удаляют каталог. Обратите внимание на то, что удалить можно только пустой каталог. Функции возвращают значение 0, если каталог удален, и значение -1 в противном случае. Код ошибки сохраняется в глобальной переменной errno. Прототипы функций:
#include <direct.h>
int _rmdir(const char *Path);
#include <wchar.h>
int _wrmdir(const wchar_t *Path);

Пример удаления пустого каталога:
if (_rmdir("C:\book\folder1") == 0) {
    std::cout << "OK" << std::endl;
}
```

Перебор объектов, расположенных в каталоге

Перебрать все (или только некоторые) объекты в указанном каталоге позволяют функции _findfirst(), _findnext() и _findclose(). Поиск осуществляется следующим образом:

1. вызывается функция _findfirst(). Функция возвращает дескриптор, с помощью которого производится дальнейший поиск,

2. в цикле вызывается функция _findnext(). Этой функции необходимо передать дескриптор, возвращаемый функцией _findfirst(). Если объектов больше нет, то функция возвращает значение -1;

3. с помощью функции _findclose() завершается поиск.

По умолчанию в VC++ 2010 функция _findfirst() соответствует функции _findfirst64i32(), а функция _findnext() — функции _findnext64i32(). Число

Листинги на странице http://unicross.narod.ru/cpp/
64 в этих функциях означает количество бит, выделяемых под дату, а число 32 означает количество бит, выделяемых под размер файла. Прототипы функций:

```
#include <io.h>

intptr_t _findfirst64i32(const char *Filename,
                        struct _finddata64i32_t *FindData);

int _findnext64i32(intptr_t FindHandle,
                    struct _finddata64i32_t *FindData);

int _findclose(intptr_t FindHandle);
```

В первом параметре функции _findfirst64i32 () указывается путь к папке и маска поиска. Если путь не указан, то поиск производится в текущем рабочем каталоге. В маске можно использовать следующие специальные символы:

- `?` — любой одиночный символ;
- `*` — любое количество символов.

Рассмотрим несколько примеров масок:

- `*` — все файлы и подкаталоги;
- `*.txt` — все файлы с расширением txt;
- `t???.c??` — файлы с названием из четырех букв, начинающиеся с буквы "т", имеющие расширение из трех букв, начинающегося с буквы "с";
- `t*` — файлы и подкаталоги, начинающиеся с буквы "т".

В параметре FindData передается указатель на структуру _finddata64i32_t, в которую будет записаны атрибуты найденного объекта. Структура объявлена так:

```
struct _finddata64i32_t {
    unsigned attrib;    /* Флаги */
    __time64_t time_create; /* Дата создания */
    __time64_t time_access; /* Дата последнего доступа */
    __time64_t time_write; /* Дата изменения */
    _fsize_t size;       /* Размер файла */
    char name[260];      /* Название объекта */
};
```

Поле `attrib` может содержать комбинацию следующих флагов:

```
#define _A_NORMAL 0x00       /* Обычный файл */
#define _A_RDONLY 0x01       /* Файл только для чтения */
```

Листинг на странице http://unicross.narod.ru/cpp/
Если объект, соответствующий параметру `Filename`, найден, то функция `_findfirst64i32()` записывает атрибуты объекта в структуру `finddata64i32_t` и возвращает дескриптор. Если поиск окончился неудачей, то функция возвращает значение `−1` и присваивает глобальной переменной `errno` код ошибки.

Поиск остальных объектов производится в цикле с помощью функции `_findnext64i32()`. В первом параметре функция принимает дескриптор, который был получен с помощью функции `findfirst64i32()`, и указатель на структуру `finddata64i32_t`. Если объект найден, то его атрибуты записываются в структуру и возвращается значение 0. Если поиск окончился неудачей, то функция возвращает значение `−1` и присваивает глобальной переменной `errno` код ошибки.

Завершение поиска осуществляется с помощью функции `findclose()`, которая в качестве параметра принимает дескриптор. Если ошибок не произошло, то функция возвращает значение 0. В противном случае функция возвращает значение `−1` и присваивает глобальной переменной `errno` код ошибки.

Пример поиска всех файлов и подкаталогов показан в листинге 12.21.

### Листинг 12.21. Перебор всех объектов, расположенных в каталоге

```
#include <iostream>
#include <io.h>
#include <cstdio>

int main() {
    struct _finddata_t info;
    intptr_t h;
    h = _findfirst("C:\\book\\*", &info);
    if (h == -1) std::cerr << "Error";
    else {
        do {
            std::cout.width(30);
            std::cout << std::left << info.name;
            std::cout << std::endl;
        } while (_findnext64i32(h, &info));
    }
}
```

Листинги на странице http://unicross.narod.ru/cpp/
if (info.attrib & _A_SUBDIR)
    std::cout << " _A_SUBDIR";
if (info.attrib & _A_RDONLY)
    std::cout << " _A_RDONLY";
if (info.attrib & _A_HIDDEN)
    std::cout << " _A_HIDDEN";
if (info.attrib & _A_SYSTEM)
    std::cout << " _A_SYSTEM";
if (info.attrib & _A_ARCH)
    std::cout << " _A_ARCH";
    std::cout << std::endl;
} while (_findnext(h, &info) == 0);
}
_findclose(h);
std::cin.get();
return 0;
}
Примерный результат выполнения:
.
.. 
    _A_SUBDIR
converts.cpp    _A_ARCH
converts.h     _A_ARCH
helloworld     _A_SUBDIR
helloworld.exe  _A_ARCH
test.txt       _A_RDONLY _A_ARCH

Обратите внимание на первые две строки. Одна точка обозначает текущий каталог. Две точки обозначают каталог выше уровнем, например, если поиск осуществляется в каталоге C:\book, то две точки указывают на корень диска C:. Если поиск производится в корневом каталоге диска, то этих двух строк не будет.

Листинги на странице http://unicross.narod.ru/cpp/
Программирование на C++ в Visual Studio® 2010 Express

Глава 13. STL

© Прохоренок Н.А., 2010 г., unicross@mail.ru

Этот DjVu-файл предоставляется КАК ЕСТЬ. Автор не несет никакой ответственности за прямые или косвенные проблемы, связанные с использованием данного файла. ВЫ ИСПОЛЬЗУЕТЕ ЕГО НА СВОЙ СТРАХ И РИСК.

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Запрещено:
- перепечатывать всю книгу или отдельные главы без письменного разрешения Автора;
- декомпилировать данный файл и преобразовывать его в любой другой формат.
Стандартная библиотека шаблонов (STL)

Итераторы

**Iterator** — это объект, выполняющий в контейнере роль указателя. С помощью итератора можно перемещаться внутри контейнера и получать доступ к отдельным элементам. В классах `deque`, `list`, `vector`, `map`, `multimap`, `set` и `multiset` определены следующие типы итераторов:

- **iterator** — итератор. При увеличении значения итератор перемещается к концу контейнера. Пример объявления переменной:
  ```cpp
  std::vector<int>::iterator it;
  ```

- **const_iterator** — константный итератор. Изменить значение, на которое ссылается итератор, нельзя. Пример объявления переменной:
  ```cpp
  std::vector<int>::const_iterator it;
  ```

- **reverse_iterator** — обратный итератор. При увеличении значения итератор перемещается к началу контейнера. Пример объявления переменной:
  ```cpp
  std::vector<int>::reverse_iterator it;
  ```

- **const_reverse_iterator** — константный обратный итератор. Изменить значение, на которое ссылается итератор, нельзя. Пример объявления переменной:
  ```cpp
  std::vector<int>::const_reverse_iterator it;
  ```

Присвоить значения переменным позволяют следующие методы:

- **begin()** — возвращает итератор, установленный на первый элемент контейнера. Прототипы метода:
  ```cpp
  iterator begin();
  const_iterator begin() const;
  ```

Выведем первый элемент вектора:
```cpp
std::vector<int> v;
std::vector<int>::iterator it;
v.push_back(1); v.push_back(2); v.push_back(3);
it = v.begin();
std::cout << *it << std::endl; // 1
```

cbegin() — возвращает константный итератор, установленный на первый элемент контейнера. Прототип метода:
const_iterator cbegin() const;
rbegin() — возвращает обратный итератор, установленный на последний элемент контейнера. Прототипы метода:
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;

Выведем последний элемент вектора:
std::vector<int> v;
std::vector<int>::reverse_iterator it;
v.push_back(1); v.push_back(2); v.push_back(3);
it = v.rbegin();
std::cout << *it << std::endl; // 3

crbegin() — возвращает константный обратный итератор, установленный на последний элемент контейнера. Прототип метода:
const_reverse_iterator crbegin() const;
end() — возвращает итератор, установленный на позицию после последнего элемента контейнера Прототипы метода:
iterator end();
const_iterator end() const;

Выведем последний элемент вектора:
std::vector<int> v;
std::vector<int>::iterator it;
v.push_back(1); v.push_back(2); v.push_back(3);
it = v.end();
std::cout << *(--it) << std::endl; // 3

cend() — возвращает константный итератор, установленный на позицию после последнего элемента контейнера. Прототип метода:
const_iterator cend() const;
rend() — возвращает обратный итератор, установленный на позицию перед первым элементом контейнера. Прототипы метода:

Листинги на странице http://unicross.narod.ru/cpp/
reverse_iterator rend();
const_reverse_iterator rend() const;

Выведем первый элемент вектора:
std::vector<int> v;
std::vector<int>::reverse_iterator it;
v.push_back(1); v.push_back(2); v.push_back(3);
it = v.rend();
std::cout << *(--it) << std::endl; // 1

rend() — возвращает константный обратный итератор, установленный на позицию перед первым элементом контейнера. Прототип метода:
const_reverse_iterator rend() const;

Над итераторами можно производить такие же операции, как и с указателями. Чтобы получить или изменить значение, на которое ссылается итератор, перед названием переменной указывается оператор * (*it). Перемещение итератора осуществляется с помощью операторов ++ и --. Кроме того, итераторы можно сравнивать с помощью операторов сравнения. В качестве примера изменим значение первого элемента, а затем выведем все элементы вектора в прямом и обратном порядке с помощью цикла for (листинг 13.1).

<table>
<thead>
<tr>
<th>Листинг 13.1. Перебор элементов вектора с помощью итераторов</th>
</tr>
</thead>
</table>

```c++
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v;
    std::vector<int>::iterator it1;
    std::vector<int>::reverse_iterator it2;
    for (int i = 1; i <= 10; ++i) v.push_back(i);
    it1 = v.begin();
    *it1 = 800;            // Изменение значения

    // Перебор элементов в прямом направлении
    for (it1 = v.begin(); it1 != v.end(); ++it1) {
        std::cout << *it1 << std::endl;
    }
}
```

Листинги на странице http://unicross.narod.ru/cpp/
std::cout << "--------------------" << std::endl;
// Перебор элементов в обратном порядке
for (it2=v.rbegin(); it2!=v.rend(); ++it2) {
    std::cout << *it2 << std::endl;
}
std::cin.get();
return 0;

Функциры

Функциор — это класс, в котором определен "операторный" метод operator()(). В этом случае можно обработать вызов экземпляра класса как вызов функции. Пример использования пользовательского функциора приведен в листинге 13.2.

Листинг 13.2. Пример использования функциора

#include <iostream>

class MyClass {
    int x_
;
public:
    MyClass() { x_ = 0; }
    void operator()(int x) { x_ = x; }
    int operator()() { return x_; }
};

int main() {
    MyClass obj;
    obj(10);
    std::cout << obj() << std::endl; // 10
    std::cout << MyClass()() << std::endl; // 0
    std::cin.get();
    return 0;
}
В библиотеке STL имеется множество встроенных функций, определенных в заголовочном файле `functional`. Пример подключения файла:

```cpp
#include <functional>
```

Встроенные функции делятся на унарные и бинарные. Унарные функции принимают один параметр. Перечислим унарные функции:

- `negate` — унарный минус:
  ```cpp
  std::cout << std::negate<int>{}(10); // -10
  ```

- `logical_not` — логическое отрицание:
  ```cpp
  std::cout << std::boolalpha
             << std::logical_not<bool>{}(false); // true
  ```

Бинарные функции принимают два параметра. Перечислим бинарные функции:

- `plus` — оператор сложения `+`:
  ```cpp
  std::cout << std::plus<int>{}(5, 3); // 8
  ```

- `minus` — оператор вычитания `-`:
  ```cpp
  std::cout << std::minus<int>{}(5, 3); // 2
  ```

- `multiplies` — оператор умножения `*`:
  ```cpp
  std::cout << std::multiplies<int>{}(5, 3); // 15
  ```

- `divides` — оператор деления `/`:
  ```cpp
  std::cout << std::divides<int>{}(15, 3); // 5
  ```

- `modulus` — остаток от деления `%`:
  ```cpp
  std::cout << std::modulus<int>{}(10, 3); // 1
  ```

- `bit_and` — двоичное `И &`:
  ```cpp
  std::cout << std::bit_and<int>{}(100, 75); // 64
  ```

- `bit_or` — двоичное `ИЛИ |`:
  ```cpp
  std::cout << std::bit_or<int>{}(100, 75); // 111
  ```

- `bit_xor` — двоичное исключающее `ИЛИ ^`:
  ```cpp
  std::cout << std::bit_xor<int>{}(100, 250); // 158
  ```

- `equal_to` — оператор равно `==`:
  ```cpp
  std::cout << std::boolalpha
             << std::equal_to{};
  ```

not_equal_to — оператор !=:
std::cout << std::boolalpha
   << std::not_equal_to<int>()(4, 5);  // true

less — оператор меньше <
std::cout << std::boolalpha
   << std::less<int>()(4, 5);        // true

less_equal — оператор меньше или равно <=:
std::cout << std::boolalpha
   << std::less_equal<int>()(5, 5); // true

greater — оператор больше >:
std::cout << std::boolalpha
   << std::greater<int>()(4, 5);    // false

greater_equal — оператор больше или равно >=:
std::cout << std::boolalpha
   << std::greater_equal<int>()(5, 5); // true

logical_and — логический оператор &&:
std::cout << std::boolalpha
   << std::logical_and<bool>()(false, true); // false

logical_or — логический оператор ||:
std::cout << std::boolalpha
   << std::logical_or<bool>()(false, true); // true

Как видно из примеров, создание объекта производится по следующей схеме:
Название_функтора<Tип>()
Тип данных должен совпадать с типом данных контейнера. В качестве типа может выступать объект пользовательного класса. В этом случае внутри класса необходимо перегрузить оператор, соответствующий функтору. Например, при использовании функтора less внутри класса должен быть метод operator<(), принимающий в качестве параметра ссылку на экземпляр того же класса, или соответствующая дружественная функция.

Листинги на странице http://unicross.narod.ru/cpp/
Чтобы вызвать функцию необходимо добавить круглые скобки, внутри которых передать один или два параметра:

Название_функтора<Tип>()(Параметры)

При создании пользовательских шаблонных функторов обычно наследуют класс unary_function (для унарных функторов) или binary_function (для бинарных функторов). Объявление классов:

```cpp
template<class Arg, class Result>
struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
};

template<class Arg1, class Arg2, class Result>
struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
};
```

Например, объявление функтора less выглядит следующим образом:

```cpp
template<class T>
struct less : public std::binary_function<T, T, bool> {
    bool operator()(const T &Left, const T &Right) const {
        return (Left < Right);
    }
};
```

**Инверторы**

**Инвертор** предназначен для изменения логического значения, возвращаемого функтором, на противоположное. Существуют два инвертора:

- **not1** — инвертирует значение унарного функтора. Пример:

  ```cpp
  std::cout << std::boolalpha
             << std::logical_not<bool>()(false); // true
  std::cout << std::boolalpha
             << std::not1( std::logical_not<bool>() )(false); // false
  ```

  Листинги на странице http://unicross.narod.ru/cpp/
not2 — инвертирует значение бинарного функтора. Пример:

```cpp
std::cout << std::boolalpha
  << std::less<int>()(4, 5); // true
std::cout << std::boolalpha
  << std::not2(std::less<int>(){4, 5}); // false
```

Для использования инверторов необходимо добавить заголовок:

```cpp
#include <functional>
```

Редакторы связей

Редакторы связей позволяют указать конкретное значение в одном из параметров бинарного функтора, создавая таким образом унарный функтор. Существуют два редактора связей:

- **bind1st** — передает значение1 в первый параметр функтора функтор, а значение2 во второй параметр функтора функтор:

```cpp
bind1st(Функтор, значение1)(значение2)
```

Эквивалентно:

Функтор(значение1, значение2)

Пример:

```cpp
std::cout << std::boolalpha
  << std::less<int>(){}, 4}(5);
// true (4 < 5)
```

- **bind2nd** — передает значение1 во второй параметр функтора функтор, а значение2 в первый параметр функтора функтор:

```cpp
bind2nd(Функтор, значение1)(значение2)
```

Эквивалентно:

Функтор(значение2, значение1)

Пример:

```cpp
std::cout << std::boolalpha
  << std::less<int>(){}, 4}(5);
// false (5 < 4)
```

Листинги на странице http://unicross.narod.ru/cpp/
Для использования редакторов связей необходимо добавить заголовок:
#include <functional>

Наиболее часто редакторы связей применяются совместно с алгоритмами. В качестве примера найдем первый элемент вектора, который имеет значение больше 4, с помощью алгоритма find_if (листинг 13.3). Более подробно алгоритмы будут рассматриваться далее в этой главе.

Листинг 13.3. Пример использования редакторов связей

```cpp
#include <iostream>
#include <vector>
#include <functional>
#include <algorithm>

int main() {
    std::vector<int> v;
    std::vector<int>::iterator res;
    for (int i = 1; i <= 10; ++i) v.push_back(i);
    // v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    res = std::find_if(v.begin(), v.end(),
                        std::bind2nd(std::greater<int>(), 4));
    if (res == v.end())
        std::cout << "No" << std::endl;
    else
        std::cout << *res << std::endl; // 5
    std::cin.get();
    return 0;
}
```

Адаптеры

Адаптеры преобразуют указатели на функции и методы класса таким образом, чтобы обычные функции и методы класса можно было передавать вместо функторов. Перечислим адаптеры, доступные в STL:

- ptr_fun — преобразует указатель на функцию. Синтаксис:

  `ptr_fun(Название_функции)`

Листинги на странице http://unicross.narod.ru/cpp/
В параметре название_функции можно передать название функции, принимающей один или два параметра. Адаптер возвращает экземпляры классов pointer_to_unary_function или pointer_to_binary_function в зависимости от количества параметров функции. Пример передачи адреса функции с двумя параметрами:

```cpp
bool func(int x, int y) {
    return x < y;
}
// ... фрагмент опущен ...
std::cout << std::boolalpha
    << std::bind2nd(std::ptr_fun(func), 4)(5);
// false ( вызов func(5, 4) )
```

- mem_fun — позволяет использовать указатель на метод класса при хранении в контейнере указателей на экземпляры класса. Пример передачи адреса метода print класса MyClass в алгоритм for_each, который осуществляет перебор элементов контейнера:

```cpp
void MyClass::print() {
    std::cout << x_ << " ";
}
// ... фрагмент опущен ...
std::vector<MyClass *> v2;
// ... фрагмент опущен ...
std::for_each( v2.begin(), v2.end(),
    std::mem_fun(&MyClass::print) );
```

- mem_fun_ref — позволяет использовать указатель на метод класса при хранении в контейнере объектов класса. Пример:

```cpp
std::vector<MyClass> v1;
// ... фрагмент опущен ...
std::for_each( v1.begin(), v1.end(),
    std::mem_fun_ref(&MyClass::print) );
```

Для использования адаптеров необходимо добавить заголовок:

```cpp
#include <functional>
```

Пример использования адаптеров `mem_fun` и `mem_fun_ref` показан в листинге 13.4.

### Листинг 13.4. Пример использования адаптеров `mem_fun` и `mem_fun_ref`

```cpp
#include <iostream>
#include <vector>
#include <functional>
#include <algorithm>

class MyClass {
    int x_;  
public:
    MyClass() { x_ = 0; }
    MyClass(int x) { x_ = x; }
    MyClass(const MyClass &obj) { x_ = obj.x_; }
    void print() {
        std::cout << x_ << " ";
    }
    void plus(int x) {
        x_ += x;
    }
}

void MyClass::print() {
    std::cout << x_ << " ";
}

void MyClass::plus(int x) {
    x_ += x;
}

int main() {
    std::vector<MyClass> v1;    // Хранит объекты класса MyClass
    std::vector<MyClass *> v2;   // Хранит указатели на объекты
    MyClass obj1(40);
    MyClass obj2(50);
    MyClass obj3(60);

    // Использование адаптера `mem_fun_ref`
    v1.push_back(MyClass(10));
    v1.push_back(MyClass(20));

    Листинги на странице http://unicross.narod.ru/cpp/
```
v1.push_back(MyClass(30));
std::for_each( v1.begin(), v1.end(),
            std::mem_fun_ref(&MyClass::print) );
std::cout << std::endl; // 10 20 30
// Изменяем значение каждого объекта
std::for_each( v1.begin(), v1.end(),
            std::bind2nd(std::mem_fun_ref(&MyClass::plus), 5) );
std::for_each( v1.begin(), v1.end(),
            std::mem_fun_ref(&MyClass::print) );
std::cout << std::endl; // 15 25 35

// Использование адаптера mem_fun
v2.push_back(&obj1);
v2.push_back(&obj2);
v2.push_back(&obj3);
std::for_each( v2.begin(), v2.end(),
            std::mem_fun(&MyClass::print) );
std::cout << std::endl; // 40 50 60
std::cin.get();}
return 0;
}

Обзор контейнеров

Итак, вспомогательные средства (итераторы, функции, инверторы, редакторы связей и адаптеры) библиотеки STL рассмотрены и теперь можно переходить к изучению контейнеров и алгоритмов. Начнем с контейнеров, а если быть точнее, то продолжим изучение контейнеров, так как ранее мы рассматривали классы string и wstring. Эти классы являются спецификацией шаблонного класса basic_string, который хотя и не является частью STL, но полностью совместим с этой библиотекой. Таким образом к классам string и wstring можно применять алгоритмы, имеющиеся в библиотеке STL. Кроме того, объекты этих классов можно сохранить в других контейнерах.

В библиотеке STL контейнеры делятся на три типа:

→ последовательные контейнеры. К этому типу относятся классы deque (двусторонняя очередь), list (список) и vector (динамический массив);

Листинги на странице http://unicross.narod.ru/cpp/
ассоциативные контейнеры. К этому типу относятся классы map (ассоциативный массив, в котором каждому ключу соответствует одно значение), multimap (ассоциативный массив, в котором ключу может соответствовать несколько значений), set (множество, состоящее из уникальных элементов) и multiset (множество, в котором элементы могут повторяться);

контейнеры-адаптеры. К этому типу относятся классы priority_queue (очередь с приоритетами), queue (очередь) и stack (стек). Обратите внимание на то, что контейнеры-адаптеры не позволяют использовать итераторы.

Все контейнеры являются динамическими. Это означает, что при добавлении нового элемента следить за размерами контейнера нет необходимости. Управление динамической памятью осуществляется автоматически с помощью распределителя памяти, который реализуется классом allocator. При создании объектов контейнера можно указать экземпляр класса, который реализует пользовательский распределитель памяти. Получить доступ к распределителю памяти через объект контейнера позволяет метод get_allocator(). Так как в большинстве случаев достаточно возможностей встроенного распределителя памяти мы не будем создавать собственные классы, а также рассматривать класс allocator. Чтобы получить подробные сведения по этим вопросам обращайтесь к документации.

Класс deque. Двусторонняя очередь

Класс deque реализует двустороннюю очередь. Добавлять элементы можно в любую позицию очереди. Прежде чем использовать класс необходимо добавить заголовок:
#include <deque>

Создание объекта

Создать экземпляр класса deque можно следующими способами:

объявить экземпляр класса без инициализации. Для этого перед названием переменной указывается название класса. После названия класса внутри угловых скобок задается тип данных. В этом случае объект не содержит элементов. Пример объявления без инициализации:

std::deque<int> d;

указать внутри круглых скобок количество элементов. Пример:

std::deque<int> d(5);

Листинги на странице http://unicross.narod.ru/cpp/
Все элементы будут иметь значения по умолчанию для типа. Например, для типа int все элементы будут содержать значение 0. Указать другое значение можно во втором параметре. Пример создания объекта из 5 элементов со значением 1:

```cpp
std::deque<int> d(5, 1);
```

Заполнить объект класса deque внутри круглых скобок или после оператора =:

```cpp
std::deque<int> d1(5, 1);
std::deque<int> d2(d1);
std::deque<int> d3 = d1;
```

Указать диапазон внутри контейнера с помощью итераторов. В первом параметре передается итератор, указывающий на начало диапазона, а во втором параметре — итератор, указывающий на конец диапазона. Пример:

```cpp
std::deque<int> d1(5, 1);
std::deque<int> d2(d1.begin(), d1.end());
```

Над двумя объектами класса deque определены операции ==, !=, <, <=, > и >=. Пример сравнения двух объектов:

```cpp
std::deque<int> d1(5, 1);
std::deque<int> d2(d1.begin(), d1.end());
if (d1 == d2) {
    std::cout << "d1 == d2" << std::endl;
}
```

Кроме того, один объект можно присвоить другому объекту. В этом случае выполняется позиционное копирование. Пример:

```cpp
std::deque<int> d1(3, 1), d2;
std::deque<int>::const_iterator i;
d2 = d1;
for (i=d2.begin(); i!=d2.end(); ++i) {
    std::cout << *i << " ";
} // 1 1 1
```

Вместо оператора = для присваивания значения можно воспользоваться методом assign(). Прототип метода:

```cpp
void assign(size_type Count, const T &Val);
template<class It>
void assign(It First, It Last);
```

Листинги на странице http://unicross.narod.ru/cpp/
Первый прототип удаляет существующие элементы, а затем вставляет `val` элементов `Val`. Пример:
```cpp
std::deque<int> d1(2, 1);
std::deque<int>::const_iterator i;
d1.assign(3, 5);
for (i=d1.begin(); i!=d1.end(); ++i) {
    std::cout << *i << " ";
} // 5 5 5
```
Второй прототип удаляет существующие элементы, а затем вставляет элементы из диапазона, ограниченного итераторами `First` и `Last`. Пример:
```cpp
std::deque<int> d1(2, 1);
std::deque<int> d2(3, 5);
std::deque<int>::const_iterator i;
d1.assign(d2.begin(), d2.end());
for (i=d1.begin(); i!=d1.end(); ++i) {
    std::cout << *i << " ";
} // 5 5 5
```

**Вставка элементов**

Вставлять элементы позволяют следующие методы:

- `push_back()` — добавляет элемент в конец очереди. Прототип метода:
  ```cpp
  void push_back(const T &Val);
  ```
  Пример:
  ```cpp
  std::deque<int> d(2, 1);
  std::deque<int>::const_iterator i;
  d.push_back(5);
  for (i=d.begin(); i!=d.end(); ++i) {
      std::cout << *i << " ";
  } // 1 1 5
  ```

- `push_front()` — добавляет элемент в начало очереди. Прототип метода:
  ```cpp
  void push_front(const T &Val);
  ```
  Пример:
  ```cpp
  ```

Листинги на странице http://unicross.narod.ru/cpp/
std::deque<int> d(2, 1);
std::deque<int>::const_iterator i;
d.push_front(5);
for (i=d.begin(); i!=d.end(); ++i) {
    std::cout << *i << " ";
} // 5 1 1

insert() — вставляет элемент в определенную позицию. Прототипы метода:

iterator insert(const_iterator Where, const T &Val);
void insert(const_iterator Where, size_type Count,
                       const T &Val);

template<class It>
void insert(const_iterator Where, It First, It Last);

Первый прототип вставляет элемент в позицию, на которую указывает итератор Where. Остальные элементы сдвигаются в конец очереди. Метод возвращает итератор, указывающий на вставленный элемент. Пример:

std::deque<int> d(2, 1);
std::deque<int>::const_iterator i;
std::deque<int>::iterator it = d.begin();
++it;
d.insert(it, 5);
for (i=d.begin(); i!=d.end(); ++i) {
    std::cout << *i << " ";
} // 1 5 1

Второй прототип вставляет Count элементов Val в позицию, на которую указывает итератор Where. Остальные элементы сдвигаются в конец очереди. Пример:

std::deque<int> d(2, 1);
std::deque<int>::const_iterator i;
std::deque<int>::iterator it = d.begin();
++it;
d.insert(it, 3, 5);
for (i=d.begin(); i!=d.end(); ++i) {

Листинги на странице http://unicross.narod.ru/cpp/
std::cout << *i << " ";
} // 1 5 5 5 1

Третий прототип вставляет элементы из диапазона, ограниченного итераторами first и last, в позицию, на которую указывает итератор where. Остальные элементы сдвигаются в конец очереди. Пример:
std::deque<int> d1(2, 1);
std::deque<int> d2(3, 5);
std::deque<int>::const_iterator i;
std::deque<int>::iterator it = d1.begin();
++it;
d1.insert(it, d2.begin(), d2.end());
for (i=d1.begin(); i!=d1.end(); ++i) {
    std::cout << *i << " ";
} // 1 5 5 5 1

swap() — меняет элементы двух контейнеров местами. Прототип метода:
void swap(deque &Right);

Пример:
std::deque<int> d1(2, 1), d2(3, 5);
std::deque<int>::const_iterator i;
d1.swap(d2);
for (i=d1.begin(); i!=d1.end(); ++i) {
    std::cout << *i << " ";
} // 5 5 5
std::cout << std::endl;
for (i=d2.begin(); i!=d2.end(); ++i) {
    std::cout << *i << " ";
} // 1 1

Удаление элементов

Для удаления элементов предназначены следующие методы:

pop_back() — удаляет последний элемент. Прототип метода:

Листинги на странице http://unicross.narod.ru/cpp/
void pop_back();

Пример:
std::deque<int> d;
std::deque<int>::const_iterator i;
d.push_back(1); d.push_back(2);
d.pop_back();
for (i=d.begin(); i!=d.end(); ++i) {
    std::cout << *i << " ";
} // 1
	pop_front() — удаляет первый элемент. Прототип метода:
void pop_front();

Пример:
std::deque<int> d;
std::deque<int>::const_iterator i;
d.push_back(1); d.push_back(2);
d.pop_front();
for (i=d.begin(); i!=d.end(); ++i) {
    std::cout << *i << " ";
} // 2
	erease() — удаляет один элемент или элементы из диапазона. Прототипы метода:
iterator erase(const_iterator Where);
iterator erase(const_iterator First, const_iterator Last);

Первый прототип удаляет элемент на который указывает итератор. Пример:
std::deque<int> d;
std::deque<int>::const_iterator i, it;
d.push_back(1); d.push_back(2);
it = --d.end();
d.erase(it);
for (i=d.begin(); i!=d.end(); ++i) {
    std::cout << *i << " ";
}

Листинги на странице http://unicross.narod.ru/cpp/
} // 1

Второй прототип удаляет элементы из диапазона, ограниченного итераторами First и Last. Пример:

```cpp
std::deque<int> d;
std::deque<int>::const_iterator i;
d.push_back(1); d.push_back(2); d.push_back(3);
d.erase(d.begin(), --d.end());
for (i=d.begin(); i!=d.end(); ++i) {
    std::cout << *i << " ";
} // 3
```

- clear() — удаляет все элементы. Прототип метода:
  ```cpp
  void clear();
  ```

- empty() — возвращает значение true, если контейнер не содержит элементов, и false — в противном случае. Прототип метода:
  ```cpp
  bool empty() const;
  ```

Пример проверки:

```cpp
std::deque<int> d;
d.push_back(1); d.push_back(2); d.push_back(3);
d.clear();
if (d.empty()) {
    std::cout << "Нет элементов" << std::endl;
}
```

- size() — возвращает количество элементов в контейнере. Прототип метода:
  ```cpp
  size_type size() const;
  ```

Пример:

```cpp
std::deque<int> d;
d.push_back(1); d.push_back(2); d.push_back(3);
std::cout << d.size() << std::endl; // 3
```

- max_size() — возвращает максимальное количество элементов, которое может содержаться в контейнере. Прототип метода:
size_type max_size() const;

Пример:
std::deque<int> d1;
std::cout << d1.max_size() << std::endl; // 1073741823
std::deque<char> d2;
std::cout << d2.max_size() << std::endl; // 4294967295

resize() — задает количество элементов, равное числу Newsize. Если указанное количество элементов меньше текущего количества, то лишние элементы будут удалены. Если количество элементов необходимо увеличить, то в параметре Val можно указать значение, которое заполнит новое пространство. Прототипы метода:
void resize(size_type Newsize);
void resize(size_type Newsize, const T &Val);

Пример:
std::deque<int> d;
std::deque<int>::const_iterator i;
d.push_back(1); d.push_back(2); d.push_back(3);
d.resize(2);
for (i=d.begin(); i!=d.end(); ++i) {
    std::cout << *i << " ";
} // 1 2
std::cout << std::endl;
d.resize(5, 0);
for (i=d.begin(); i!=d.end(); ++i) {
    std::cout << *i << " ";
} // 1 2 0 0 0

Доступ к элементам

К любому элементу можно обратиться как к элементу массива. Достаточно указать его индекс в квадратных скобках. Нумерация начинается с нуля. Можно как получить значение, так и изменить его. Если индекс выходит за границы диапазона, то возвращаемое значение не определено. Пример доступа к элементу по индексу:
std::deque<int> d(2, 1);

Листинги на странице http://unicross.narod.ru/cpp/
std::cout << d[0] << " " << d[1] << std::endl; // 1 1

// Использование метода at()

d[1] = 2;
std::cout << d[0] << " " << d[1] << std::endl; // 1 2
Для доступа к элементам предназначены следующие методы:

→ at() — возвращает ссылку на элемент, расположенный по индексу pos. Метод позволяет как получить значение, так и изменить его. Если индекс выходит за границы диапазона, то метод генерирует исключение. Прототипы метода:

reference at(size_type Pos);
const_reference at(size_type Pos) const;

Пример:

std::deque<int> d(2, 1);
std::cout << d.at(0) << " " << d.at(1) << std::endl; // 1 1

// Использование метода at()

d.at(1) = 2;
std::cout << d.at(0) << " " << d.at(1) << std::endl; // 1 2

→ front() — возвращает ссылку на первый элемент. Метод позволяет как получить значение, так и изменить его. Прототипы метода:

reference front();
const_reference front() const;

Пример:

std::deque<int> d(2, 1);
d.front() = 5;

std::cout << d.front() << std::endl; // 5

// Использование метода front()

std::cout << d[0] << " " << d[1] << std::endl; // 5 1

→ back() — возвращает ссылку на последний элемент. Метод позволяет как получить значение, так и изменить его. Прототипы метода:

reference back();
const_reference back() const;

Пример:

std::deque<int> d(2, 1);
d.back() = 5;

std::cout << d.back() << std::endl; // 5

Листинги на странице http://unicross.narod.ru/cpp/
std::cout << d[0] << " " << d[1] << std::endl; // 1 5

- begin() — возвращает итератор, установленный на первый элемент контейнера. Прототипы метода:
  iterator begin();
  const_iterator begin() const;

- cbegin() — возвращает константный итератор, установленный на первый элемент контейнера. Прототип метода:
  const_iterator cbegin() const;

- rbegin() — возвращает обратный итератор, установленный на последний элемент контейнера. Прототипы метода:
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;

- crbegin() — возвращает константный обратный итератор, установленный на последний элемент контейнера. Прототип метода:
  const_reverse_iterator crbegin() const;

- end() — возвращает итератор, установленный на позицию после последнего элемента контейнера. Прототипы метода:
  iterator end();
  const_iterator end() const;

- cend() — возвращает константный итератор, установленный на позицию после последнего элемента контейнера. Прототип метода:
  const_iterator cend() const;

- rend() — возвращает обратный итератор, установленный на позицию перед первым элементом контейнера. Прототипы метода:
  reverse_iterator rend();
  const_reverse_iterator rend() const;

- crend() — возвращает константный обратный итератор, установленный на позицию перед первым элементом контейнера. Прототип метода:
  const_reverse_iterator crend() const;

В качестве примера изменим значение первого элемента, а затем выведем все элементы в прямом и обратном порядке с помощью цикла for (листинг 13.5).

Листинги на странице http://unicross.narod.ru/cpp/
Листинг 13.5. Перебор элементов с помощью итераторов

```cpp
#include <iostream>
#include <deque>

int main() {
    std::deque<int> d;
    std::deque<int>::iterator it1;
    std::deque<int>::reverse_iterator it2;
    for (int i = 1; i <= 10; ++i) d.push_back(i);
    it1 = d.begin();
    *it1 = 800; // Изменение значения
    // Перебор элементов в прямом направлении
    for (it1=d.begin(); it1!=d.end(); ++it1) {
        std::cout << *it1 << std::endl;
    }
    std::cout << "-------------------" << std::endl;
    // Перебор элементов в обратном порядке
    for (it2=d.rbegin(); it2!=d.rend(); ++it2) {
        std::cout << *it2 << std::endl;
    }
    std::cin.get();
    return 0;
}
```

Класс list. Список

Класс list реализует список. Добавить элементы можно в любое место списка. В отличие от класса deque получить доступ к элементу списка по индексу нельзя. Для списков определены дополнительные операции: сортировка, объединение списков, изменение порядка следования элементов на противоположный и др. Прежде чем использовать класс list необходимо добавить заголовок:
```
#include <list>
```

Создание объекта

Создать экземпляр класса list можно следующими способами:

Листинги на странице http://unicross.narod.ru/cpp/
объявить экземпляр класса без инициализации. Для этого перед названием переменной указывается название класса. После названия класса внутри угловых скобок задается тип данных. В этом случае объект не содержит элементов. Пример объявления без инициализации:

```cpp
std::list<int> L;
```

указать внутри круглых скобок количество элементов. Пример:

```cpp
std::list<int> L(5);
```

Все элементы будут иметь значения по умолчанию для типа int. Например, для типа int все элементы будут содержать значение 0. Указать другое значение можно во втором параметре. Пример создания объекта из 5 элементов со значением 1:

```cpp
std::list<int> L(5, 1);
```

указать объект класса list внутри круглых скобок или после оператора =:

```cpp
std::list<int> L1(5, 1);
std::list<int> L2(L1);
std::list<int> L3 = L1;
```

указать диапазон внутри контейнера с помощью итераторов. В первом параметре передается итератор, указывающий на начало диапазона, а во втором параметре — итератор, указывающий на конец диапазона. Пример:

```cpp
std::list<int> L1(5, 1);
std::list<int> L2(L1.begin(), L1.end());
```

Над двумя объектами класса list определены операции ==, !=, <, <=, > и >=. Пример сравнения двух объектов:

```cpp
std::list<int> L1(5, 1);
std::list<int> L2(L1.begin(), L1.end());
if (L1 == L2) {
    std::cout << "L1 == L2" << std::endl;
}
```

Кроме того, один объект можно присвоить другому объекту. В этом случае выполняется поэлементное копирование. Пример:

```cpp
std::list<int> L1(3, 1), L2;
std::list<int>::const_iterator i;
L2 = L1;
for (i=L2.begin(); i!=L2.end(); ++i) {
    // действия с элементами L2
}
```

Листинги на странице http://unicross.narod.ru/cpp/
Вместо оператора = для присваивания значения можно воспользоваться методом assign(). Прототипы метода:

```cpp
void assign(size_type Count, const T &Val);
```

```cpp
template<class It>
void assign(It First, It Last);
```

Первый прототип удаляет существующие элементы, а затем вставляет `Count` элементов `Val`. Второй прототип удаляет существующие элементы, а затем вставляет элементы из диапазона, ограниченного итераторами `First` и `Last`.

### Вставка элементов

Вставить элементы позволяют следующие методы:

- **push_back()** — добавляет элемент в конец контейнера. Прототип метода:
  ```cpp
  void push_back(const T &Val);
  ```

- **push_front()** — добавляет элемент в начало контейнера. Прототип метода:
  ```cpp
  void push_front(const T &Val);
  ```

- **insert()** — вставляет элемент в определенную позицию. Прототипы метода:
  ```cpp
  iterator insert(const_iterator Where, const T &Val);
  void insert(const_iterator Where, size_type Count,
              const T &Val);
  
  template<class It>
  void insert(const_iterator Where, It First, It Last);
  ```

Первый прототип вставляет элемент в позицию, на которую указывает итератор `Where`. Остальные элементы сдвигаются в конец списка. Метод возвращает итератор, указывающий на вставленный элемент.

Второй прототип вставляет `Count` элементов `Val` в позицию, на которую указывает итератор `Where`. Остальные элементы сдвигаются в конец списка.

Третий прототип вставляет элементы из диапазона, ограниченного итераторами `First` и `Last`, в позицию, на которую указывает итератор `Where`. Остальные элементы сдвигаются в конец списка;

- **swap()** — меняет элементы двух контейнеров местами. Прототип метода:

  Листинги на странице http://unicross.narod.ru/cpp/
void swap(list &Right);

Удаление элементов
Для удаления элементов предназначены следующие методы:

- pop_back() — удаляет последний элемент. Прототип метода:
  void pop_back();

- pop_front() — удаляет первый элемент. Прототип метода:
  void pop_front();

- erase() — удаляет один элемент или элементы из диапазона. Прототипы метода:
  iterator erase(const_iterator Where);
  iterator erase(const_iterator First, const_iterator Last);
Первый прототип удаляет элемент на который указывает итератор. Второй прототип удаляет элементы из диапазона, ограниченного итераторами First и Last;

- clear() — удаляет все элементы. Прототип метода:
  void clear();

- empty() — возвращает значение true, если контейнер не содержит элементов, и false — в противном случае. Прототип метода:
  bool empty() const;

- size() — возвращает количество элементов в контейнере. Прототип метода:
  size_type size() const;

- max_size() — возвращает максимальное количество элементов, которое может содержаться в контейнере. Прототип метода:
  size_type max_size() const;

- resize() — задает количество элементов, равное числу Newsize. Если указанное количество элементов меньше текущего количества, то лишние элементы будут удалены. Если количество элементов необходимо увеличить, то в параметре Val можно указать значение, которое заполнит новое пространство. Прототипы метода:
  void resize(size_type Newsize);
  void resize(size_type Newsize, const T &Val);

Помимо рассмотренных методов, имеющихся также в классе deque, в классе list присутствуют дополнительные методы:

Листинги на странице http://unicross.narod.ru/cpp/
remove() — удаляет все элементы, имеющие значение Val. Прототип метода:

```cpp
void remove(const T &Val);
```

Пример:

```cpp
class list<int> L(3, 1);
list<int>::const_iterator i;
L.push_front(0); L.push_back(2);
for (i=L.begin(); i!=L.end(); ++i) {
    std::cout << *i << " ";
} // 0 1 1 1 2
std::cout << std::endl;
L.remove(1);
for (i=L.begin(); i!=L.end(); ++i) {
    std::cout << *i << " ";
} // 0 2
```

remove_if() — удаляет элементы, для которых унарный функтор Pred вернул значение true. Функтор, возвращающий логическое значение, называется предикатом. Прототип метода:

```cpp
template<class Pr> void remove_if(Pr Pred);
```

Удалим все элементы, значения которых меньше числа 2:

```cpp
class list<int> L(3, 1);
list<int>::const_iterator i;
L.push_front(0); L.push_back(2); L.push_back(-1);
for (i=L.begin(); i!=L.end(); ++i) {
    std::cout << *i << " ";
} // 0 1 1 1 2 -1
std::cout << std::endl;
L.remove_if(std::bind2nd(std::less<int>(), 2));
for (i=L.begin(); i!=L.end(); ++i) {
    std::cout << *i << " ";
} // 2
```

Листинги на странице http://unicross.narod.ru/cpp/
unique() — удаляет повторяющиеся элементы. Прототипы метода:

```cpp
void unique();

template<class Pr> void unique(Pr Pred);
```

Первый прототип удаляет повторяющиеся элементы, расположенные рядом. Чтобы список состоял только из уникальных элементов необходимо предварительно отсортировать список. Второй прототип передает два элемента бинарному функтору. Если функтор вернет значение `true`, то повторяющийся элемент будет удален. Пример:

```cpp
std::list<int> L1(3, 1), L2;
std::list<int>::const_iterator i;
L1.push_front(0); L1.push_back(2); L1.push_back(1);
L2 = L1;
L1.unique();
for (i=L1.begin(); i!=L1.end(); ++i) {
    std::cout << *i << " ";
} // 0 1 2 1
std::cout << std::endl;
L2.unique(std::equal_to<int>());
for (i=L2.begin(); i!=L2.end(); ++i) {
    std::cout << *i << " ";
} // 0 1 2 1
```

**Доступ к элементам**

Для доступа к элементам предназначены следующие методы:

- front() — возвращает ссылку на первый элемент. Метод позволяет как получить значение, так и изменить его. Прототипы метода:

  ```cpp
  reference front();
  const_reference front() const;
  ```

- back() — возвращает ссылку на последний элемент. Метод позволяет как получить значение, так и изменить его. Прототипы метода:

  ```cpp
  reference back();
  const_reference back() const;
  ```

- begin() — возвращает итератор, установленный на первый элемент контейнера. Прототипы метода:
  iterator begin();
  const_iterator begin() const;

- cbegin() — возвращает константный итератор, установленный на первый элемент контейнера. Прототип метода:
  const_iterator cbegin() const;

- rbegin() — возвращает обратный итератор, установленный на последний элемент контейнера. Прототипы метода:
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;

- crbegin() — возвращает константный обратный итератор, установленный на последний элемент контейнера. Прототип метода:
  const_reverse_iterator crbegin() const;

- end() — возвращает итератор, установленный на позицию после последнего элемента контейнера. Прототипы метода:
  iterator end();
  const_iterator end() const;

- cend() — возвращает константный итератор, установленный на позицию после последнего элемента контейнера. Прототип метода:
  const_iterator cend() const;

- rend() — возвращает обратный итератор, установленный на позицию перед первым элементом контейнера. Прототипы метода:
  reverse_iterator rend();
  const_reverse_iterator rend() const;

- crend() — возвращает константный обратный итератор, установленный на позицию перед первым элементом контейнера. Прототип метода:
  const_reverse_iterator crend() const;

Листинги на странице http://unicross.narod.ru/cpp/
Сортировка, объединение и переворачивание списков

Помимо методов, рассмотренных в предыдущих разделах, в классе list присутствуют дополнительные методы, позволяющие отсортировать список, объединить два списка, а также изменить порядок следования элементов на противоположный.

Для сортировки списка предназначен метод sort(). Прототипы метода:

```cpp
void sort();
template<class Pr> void sort(Pr Pred);
```

Первый прототип производит сортировку по умолчанию, а второй прототип позволяет указать пользовательскую функцию сортировки. Отсортируем список в прямом и обратном порядке (листинг 13.6).

```
Листинг 13.6. Сортировка списка

```cpp
#include <iostream>
#include <list>
#include <functional>

int main() {
    std::list<int> L;
    std::list<int>::const_iterator i;
    L.push_back(5); L.push_back(2); L.push_back(4);

    // Сортировка в прямом порядке
    L.sort();
    for (i=L.begin(); i!=L.end(); ++i) {
        std::cout << *i << " ";
    } // 2 4 5
    std::cout << std::endl;

    // Сортировка в обратном порядке
    L.sort(std::greater<int>());
    for (i=L.begin(); i!=L.end(); ++i) {
        std::cout << *i << " ";
    } // 5 4 2
    std::cout << std::endl;
```

Листинги на странице http://unicross.narod.ru/cpp/
std::cin.get();
return 0;
}

Для объединения списков предназначены методы splice() и merge(). Прототипы метода splice():

void splice(const_iterator Where, list &Right);
void splice(const_iterator Where, list &Right, const_iterator First);
void splice(const_iterator Where, list &Right, const_iterator First,
            const_iterator Last);

Первый прототип вставляет все элементы списка Right, в позицию, на которую указывает итератор where, и очищает список right. Если вставка происходит не в конец списка, то существующие элементы сдвигаются в конец списка. Пример добавления элементов перед последним элементом списка:

```cpp
std::list<int> L1(3, 1), L2(3, 2);
std::list<int>::const_iterator i;
L1.splice(--L1.end(), L2);
for (i=L1.begin(); i!=L1.end(); ++i) {
    std::cout << *i << " ";
} // 1 1 2 2 2 1
std::cout << std::endl << L2.size() << std::endl; // 0
```

Второй прототип удаляет элемент, на который указывает итератор first, из списка right и вставляет его в позицию, на которую указывает итератор where. Существующие элементы сдвигаются в конец списка. Пример:

```cpp
std::list<int> L1(3, 1), L2;
std::list<int>::const_iterator i;
L2.push_back(1); L2.push_back(2); L2.push_back(3);
L1.splice(L1.end(), L2, ++L2.begin());
for (i=L1.begin(); i!=L1.end(); ++i) {
    std::cout << *i << " ";
} // 1 1 1 2
std::cout << std::endl;
for (i=L2.begin(); i!=L2.end(); ++i) {
    std::cout << *i << " ";
} // 1 3
```

Листинги на странице http://unicross.narod.ru/cpp/
Третий прототип удаляет элементы, входящие в диапазон, ограниченный итераторами first и last, из списка right и вставляет их в позицию, на которую указывает итератор where. Существующие элементы сдвигаются в конец списка. Пример:

```cpp
std::list<int> L1(3, 1), L2;
std::list<int>::const_iterator i;
L2.push_back(1); L2.push_back(2); L2.push_back(3);
L1.splice(--L1.end(), L2, ++L2.begin(), L2.end());
for (i=L1.begin(); i!=L1.end(); ++i) {
    std::cout << *i << " ";
} // 1 1 2 3 1
std::cout << std::endl;
for (i=L2.begin(); i!=L2.end(); ++i) {
    std::cout << *i << " ";
} // 1
```

Метод merge() предназначен для объединения упорядоченных списков. Результатом выполнения метода является упорядоченный список. Прототипы метода:

```cpp
void merge(list &Right);
template<class Pr> void merge(list &Right, Pr Pred);
```

Первый прототип добавляет все элементы списка right и очищает его. Пример:

```cpp
std::list<int> L1, L2;
std::list<int>::const_iterator i;
L1.push_back(3); L1.push_back(6); L1.push_back(1);
L2.push_back(5); L2.push_back(2); L2.push_back(4);
L1.sort(); L2.sort();
L1.merge(L2);
for (i=L1.begin(); i!=L1.end(); ++i) {
    std::cout << *i << " ";
} // 1 2 3 4 5 6
std::cout << std::endl << L2.size() << std::endl; // 0
```

Второй прототип позволяет дополнительно указать пользовательскую функцию сравнения. Произведем объединение списков, отсортированных в обратном порядке:

```cpp
std::list<int> L1, L2;
std::list<int>::const_iterator i;
L1.push_back(3); L1.push_back(6); L1.push_back(1);
L2.push_back(5); L2.push_back(2); L2.push_back(4);
```

Листинг на странице http://unicross.narod.ru/cpp/
L1.sort(std::greater<int>());
L2.sort(std::greater<int>());
L1.merge(L2, std::greater<int>());
for (i=L1.begin(); i!=L1.end(); ++i) {
    std::cout << *i << " ";
} // 6 5 4 3 2 1
std::cout << std::endl << L2.size() << std::endl; // 0
Для изменения порядка следования элементов на противоположный предназначен метод reverse(). Обратите внимание на то, что метод переворачивает список, а не сортирует его. Элементы будут следовать в обратном порядке относительно исходного списка. Прототип метода:
void reverse();
Пример:
std::list<int> L;
std::list<int>::const_iterator i;
L.push_back(3); L.push_back(6); L.push_back(1);
L.push_back(5); L.push_back(2); L.push_back(4);
for (i=L.begin(); i!=L.end(); ++i) {
    std::cout << *i << " ";
} // 3 6 1 5 2 4
std::cout << std::endl;
L.reverse();
for (i=L.begin(); i!=L.end(); ++i) {
    std::cout << *i << " ";
} // 4 2 5 1 6 3

Класс vector. Динамический массив

Класс vector реализует динамический массив. Он во многом похож на класс deque, но имеет важное отличие — все элементы вектора размещаются в памяти последовательно, а двусторонняя очередь может располагаться в памяти фрагментами. Благодаря последовательному размещению в памяти доступ к элементам вектора выполняется быстрее, но вставка новых элементов приводит к выделению объема памяти, достаточного для размещения всей последовательности элементов, и переносу всех элементов в новую область. Кроме того, вектор не имеет методов push_front() и pop_front(), предназначенных в классе deque для вставки элемента в начало и

Листинги на странице http://unicross.narod.ru/cpp/
удаления первого элемента соответственно. Прежде чем использовать класс необходимо добавить заголовок:
#include <vector>

Создание объекта
Создать экземпляр класса vector можно следующими способами:

→ объявить экземпляр класса без инициализации. Для этого перед названием переменной указывается название класса. После названия класса внутри угловых скобок задается тип данных. В этом случае объект не содержит элементов. Пример объявления без инициализации:
std::vector<int> v;

→ указать внутри круглых скобок количество элементов. Пример:
std::vector<int> v(5);
Все элементы будут иметь значения по умолчанию для типа. Например, для типа int все элементы будут содержать значение 0. Указать другое значение можно во втором параметре. Пример создания объекта из 5 элементов со значением 1:
std::vector<int> v(5, 1);

→ указать объект класса vector внутри круглых скобок или после оператора =:
std::vector<int> v1(5, 1);
std::vector<int> v2(v1);
std::vector<int> v3 = v1;

→ указать диапазон внутри контейнера с помощью итераторов. В первом параметре передается итератор, указывающий на начало диапазона, а во втором параметре — итератор, указывающий на конец диапазона. Пример:
std::vector<int> v1(5, 1);
std::vector<int> v2(v1.begin(), v1.end());

Над двумя объектами класса vector определены операции ==, !=, <, <=, > и >=. Пример сравнения двух объектов:
std::vector<int> v1(5, 1);
std::vector<int> v2(v1.begin(), v1.end());
if (v1 == v2) {
    std::cout << "v1 == v2" << std::endl;
}

Листинги на странице http://unicross.narod.ru/cpp/
} Кроме того, один объект можно присвоить другому объекту. В этом случае выполняется поэлементное копирование. Пример:
std::vector<int> v1(3, 1), v2;
std::vector<int>::const_iterator i;
v2 = v1;
for (i=v2.begin(); i!=v2.end(); ++i) {
  std::cout << *i << " ";
} // 1 1 1
Вместо оператора = для присваивания значения можно воспользоваться методом assign(). Прототипы метода:
void assign(size_type Count, const T &Val);
template<class It>
void assign(It First, It Last);
Первый прототип удаляет существующие элементы, а затем вставляет Count элементов Val. Второй прототип удаляет существующие элементы, а затем вставляет элементы из диапазона, ограниченного итераторами First и Last.

Вставка элементов
Вставить элементы позволяют следующие методы:
push_back() — добавляет элемент в конец вектора. Прототип метода:
void push_back(const T &Val);
insert() — вставляет элемент в определенную позицию. Прототипы метода:
iterator insert(const_iterator Where, const T &Val);
void insert(const_iterator Where, size_type Count,
            const T &Val);
template<class It>
void insert(const_iterator Where, It First, It Last);
Первый прототип вставляет элемент в позицию, на которую указывает итератор Where. Остальные элементы сдвигаются в конец вектора. Метод возвращает итератор, указывающий на вставленный элемент. Второй прототип вставляет Count элементов Val в позицию, на которую указывает итератор Where. Остальные элементы сдвигаются в конец вектора. Третий прототип вставляет элементы из Листинги на странице http://unicross.narod.ru/cpp/
диапазона, ограниченного итераторами First и Last, в позицию, на которую указывает итератор Where. Остальные элементы сдvigаются в конец вектора,

- swap () — меняет элементы двух векторов местами. Прототип метода:
  
  ```cpp
  void swap(vector &Right);
  ```

Удаление элементов

Для удаления элементов предназначены следующие методы:

- pop_back () — удаляет последний элемент. Прототип метода:
  
  ```cpp
  void pop_back();
  ```

- erase () — удаляет один элемент или элементы из диапазона. Прототипы метода:
  
  ```cpp
  iterator erase(const_iterator Where);
  iterator erase(const_iterator First, const_iterator Last);
  ```

Первый прототип удаляет элемент на который указывает итератор. Второй прототип удаляет элементы из диапазона, ограниченного итераторами First и Last;

- clear () — удаляет все элементы. Прототип метода:
  
  ```cpp
  void clear();
  ```

- empty () — возвращает значение true, если вектор не содержит элементов, и false — в противном случае. Прототип метода:
  
  ```cpp
  bool empty() const;
  ```

- size () — возвращает количество элементов в контейнере. Прототип метода:
  
  ```cpp
  size_type size() const;
  ```

- max_size () — возвращает максимальное количество элементов, которое может содержаться в контейнере. Прототип метода:
  
  ```cpp
  size_type max_size() const;
  ```

- resize () — задает количество элементов, равное числу Newsize. Если указанное количество элементов меньше текущего количества, то лишние элементы будут удалены. Если количество элементов необходимо увеличить, то в параметре Val можно указать значение, которое заполнит новое пространство. Прототипы метода:
  
  ```cpp
  void resize(size_type Newsize);
  void resize(size_type Newsize, T Val);
  ```

Листинги на странице http://unicross.narod.ru/cpp/
Доступ к элементам
К любому элементу вектора можно обратиться как к элементу массива. Достаточно указать его индекс в квадратных скобках. Нумерация начинается с нуля. Можно как получить значение, так и изменить его. Если индекс выходит за границы диапазона, то возвращаемое значение не определено. Пример доступа к элементу по индексу:

```cpp
std::vector<int> v(2, 1);
std::cout << v[0] << " " << v[1] << std::endl; // 1 1
v[1] = 2;
std::cout << v[0] << " " << v[1] << std::endl; // 1 2
```

Для доступа к элементам предназначены следующие методы:

- **at()** — возвращает ссылку на элемент, расположенный по индексу `Pos`. Метод позволяет как получить значение, так и изменить его. Если индекс выходит за границы диапазона, то метод генерирует исключение. Прототипы метода:
  ```cpp
  reference at(size_type Pos);
  const_reference at(size_type Pos) const;
  ```

- **front()** — возвращает ссылку на первый элемент. Метод позволяет как получить значение, так и изменить его. Прототипы метода:
  ```cpp
  reference front();
  const_reference front() const;
  ```

- **back()** — возвращает ссылку на последний элемент. Метод позволяет как получить значение, так и изменить его. Прототипы метода:
  ```cpp
  reference back();
  const_reference back() const;
  ```

- **begin()** — возвращает итератор, установленный на первый элемент контейнера. Прототипы метода:
  ```cpp
  iterator begin();
  const_iterator begin() const;
  ```

- **cbegin()** — возвращает константный итератор, установленный на первый элемент контейнера. Прототип метода:
  ```cpp
  const_iterator cbegin() const;
  ```

- **rbegin()** — возвращает обратный итератор, установленный на последний элемент контейнера. Прототипы метода:

  Листинги на странице http://unicross.narod.ru/cpp/
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;

→ crbegin() — возвращает константный обратный итератор, установленный на последний элемент контейнера. Прототип метода:
const_reverse_iterator crbegin() const;

→ end() — возвращает итератор, установленный на позицию после последнего элемента контейнера. Прототипы метода:
iterator end();
const_iterator end() const;

→ cend() — возвращает константный итератор, установленный на позицию после последнего элемента контейнера. Прототип метода:
const_iterator cend() const;

→ rend() — возвращает обратный итератор, установленный на позицию перед первым элементом контейнера. Прототипы метода:
reverse_iterator rend();
const_reverse_iterator rend() const;

→ crend() — возвращает константный обратный итератор, установленный на позицию перед первым элементом контейнера. Прототип метода:
const_reverse_iterator crend() const;

Все элементы вектора размещаются в памяти последовательно. Поэтому вектор поддерживает возможность перебора элементов с помощью указателей. Для указателей имеются специальные типы:

→ pointer — обычный указатель. Можно как получить значение, так и изменить;
→ const_pointer — константный указатель. Изменить значение нельзя.

Присвоить указателю адрес первого элемента вектора позволяет метод data(). Прототипы метода:
pointer data();
const_pointer data() const;

Пример перебора элементов с помощью указателей показан в листинге 13.7.

Листинги на странице http://unicross.narod.ru/cpp/
Листинг 13.7. Доступ к элементам вектора с помощью указателей

```cpp
#include <iostream>
#include <vector>

int main() {  
    std::vector<int> v;
    std::vector<int>::pointer p;
    std::vector<int>::const_pointer cp;
    v.reserve(10); // Резервируем минимальный размер
    for (int i=1; i<=10; ++i) v.push_back(i);
    p = v.data(); // Сохраняем обычный указатель
    *p = 800;     // Изменяем значение первого элемента
    cp = v.data(); // Сохраняем константный указатель
    for (int j=0, c=v.size(); j<c; ++j, ++cp) {
        std::cout << *cp << " ";
    } // 800 2 3 4 5 6 7 8 9 10
    std::cout << std::endl;
    std::cin.get();
    return 0;
}
```

Получение и изменение размера вектора

Как вы уже знаете, вектор является динамическим массивом, который может автоматически увеличивать объем зарезервированной памяти. Вставка новых элементов приводит к выделению объема памяти, достаточного для размещения всей последовательности элементов, и переносу всех элементов в новую область. Если зарезервированной памяти достаточно, то перераспределение памяти не производится. Получить количество элементов, для которых зарезервирована память, позволяет метод capacity(). Прототип метода:

```cpp
size_type capacity() const;
```

Пример:

```cpp
std::vector<int> v(3, 1);
std::cout << v.size() << std::endl; // 3
std::cout << v.capacity() << std::endl; // 3
v.pop_back(); // Удаляем элемент
```

Листинг на странице http://unicross.narod.ru/cpp/
std::cout << v.size() << std::endl; // 2
std::cout << v.capacity() << std::endl; // 3

Если добавление элементов производится часто, то это может снизить эффективность программы, так как перераспределение памяти будет выполнено несколько раз. Поэтому, если минимальное количество элементов заранее известно, то следует указать его с помощью метода reserve(). Прототип метода:

```cpp
void reserve(size_type Count);
```

Пример указания минимального размера вектора:

```cpp
std::vector<int> v;
v.reserve(10);
```

std::cout << v.size() << std::endl; // 0
std::cout << v.capacity() << std::endl; // 10
v.push_back(1); v.push_back(2); v.push_back(3);
std::cout << v.size() << std::endl; // 3
std::cout << v.capacity() << std::endl; // 10

Уменьшить размер вектора до минимального значения позволяет метод shrink_to_fit(). Прототип метода:

```cpp
void shrink_to_fit();
```

Пример:

```cpp
std::vector<int> v;
v.reserve(10);
```

std::cout << v.size() << std::endl; // 0
std::cout << v.capacity() << std::endl; // 10
v.push_back(1); v.push_back(2); v.push_back(3);
v.shrink_to_fit();
std::cout << v.size() << std::endl; // 3
std::cout << v.capacity() << std::endl; // 3

Специализация vector<bool>

Для специализации vector<bool> определены дополнительные методы:

- flip() — инвертирует значения всех элементов. Прототип метода:

  ```cpp
  void flip();
  ```

Пример:

```cpp
std::vector<bool> v;
```

Листинги на странице http://unicross.narod.ru/cpp/
std::vector<bool>::const_iterator i;
v.push_back(true); v.push_back(false);
v.flip();
for (i=v.begin(); i!=v.end(); ++i) {
    std::cout << std::boolalpha << *i << " " ;
} // false true

swap() — меняет два элемента вектора местами. Прототип метода:
static void swap(reference Left, reference Right);

Пример:
std::vector<bool> v;
std::vector<bool>::const_iterator i;
v.push_back(true); v.push_back(false); v.push_back(false);
v.swap(v[0], v[1]);
for (i=v.begin(); i!=v.end(); ++i) {
    std::cout << std::boolalpha << *i << " " ;
} // false true false

Класс map. Ассоциативный массив с уникальными ключами
Класс map реализует ассоциативный массив, в котором каждому ключу соответствует одно значение. Основным отличием ассоциативных массивов от обычных является возможность обращения к элементу массива не только по числовому индексу, но и, например, по индексу, состоящему из строки. Индексы ассоциативного массива называются ключами. Прежде чем использовать класс необходимо добавить заголовок:
#include <map>

Класс pair
Класс pair, объявленный в заголовочном файле utility, реализует пару ключ/значение. Ключ доступен через атрибут first, а значение — через атрибут second. Создать экземпляр класса можно так:

объявить экземпляр класса без инициализации. Для этого перед названием переменной указывается название класса pair. После названия класса внутри

Листинги на странице http://unicross.narod.ru/cpp/
угловых скобок через запятую задаются типы данных ключа и значения. В этом случае атрибуты будут иметь значения по умолчанию для конкретного типа. Например, для типа int атрибут будет содержать значение 0. Пример объявления без инициализации:

```cpp
std::pair<int, int> pr;
std::cout << pr.first << std::endl;  // 0
std::cout << pr.second << std::endl; // 0
```

→ указав после названия переменной внутри круглых скобок два значения через запятую. Пример:

```cpp
typedef struct std::pair<std::string, int> PAIR;
PAIR pr("Key", 10);
std::cout << pr.first << std::endl;  // Key
std::cout << pr.second << std::endl; // 10
```

→ указав после названия переменной внутри круглых скобок другой объект класса pair. Пример:

```cpp
typedef struct std::pair<std::string, int> PAIR;
PAIR pr1("Key", 10);
PAIR pr2(pr1);
std::cout << pr2.first << std::endl;  // Key
std::cout << pr2.second << std::endl; // 10
```

Кроме того, можно воспользоваться шаблонной функцией make_pair(). Пример:

```cpp
typedef struct std::pair<int, int> PAIR;
PAIR pr = std::make_pair(5, 10);
std::cout << pr.first << std::endl;  // 5
std::cout << pr.second << std::endl; // 10
```

Над двумя объектами класса pair определены операции ==, !=, <, <=, > и >=. Кроме того, один объект можно присвоить другому объекту. В этом случае выполняется копирование значений атрибутов.

**Создание объекта**

Создать экземпляр класса map можно следующими способами:

Листинги на странице http://unicross.narod.ru/cpp/
объявить экземпляр класса без инициализации. Для этого перед названием переменной указывается название класса. После названия класса внутри угловых скобок через запятую задаются типы данных ключа и значения. В этом случае объект не содержит элементов. Пример объявления без инициализации:

```cpp
std::map<std::string, int> m;
```

Внутри угловых скобок после типов данных ключа и значения можно дополнительно указать функцию сравнения для ключа:

```cpp
std::map<int, int, std::less<int> > m;
```

**Внимание**

Между закрывающими угловыми скобками пробел добавлен специально. Если пробел не добавить, то некоторые компиляторы считают это ошибкой.

указать объект класса `map` внутри круглых скобок или после оператора `=:`

```cpp
std::map<int, int> m1;
std::map<int, int> m2 (m1);
std::map<int, int> m3 = m1;
```

указать диапазон внутри контейнера с помощью итераторов. В первом параметре передается итератор, указывающий на начало диапазона, а во втором параметре — итератор, указывающий на конец диапазона. Пример:

```cpp
typedef struct std::pair<int, int> PAIR;
std::map<int, int> m1;
m1.insert(PAIR(0, 10));
m1.insert(PAIR(1, 20));
std::map<int, int> m2 (m1.begin(), m1.end());
```

Над двумя объектами класса `map` определены операции `==`, `!=`, `<`, `<=`, `>` и `>=`. Кроме того, один объект можно присвоить другому объекту. Пример:

```cpp
typedef struct std::pair<int, int> PAIR;
std::map<int, int> m1, m2;
std::map<int, int>::const_iterator i;
m1.insert(PAIR(0, 10));
m1.insert(PAIR(1, 20));
m2 = m1;
for (i=m2.begin(); i!=m2.end(); ++i) {
...
```

Листинги на странице http://unicross.narod.ru/cpp/
std::cout << i->first << " -> "
    << i->second << "; ";
} // 0 -> 10; 1 -> 20;

Вставка элементов
Вставить элемент можно указав ключ внутри квадратных скобок. Значение элемента задается после оператора присваивания. Если ключ уже существует, то вместо вставки элемента производится изменение значения. Пример:
std::map<std::string, int> m;
std::map<std::string, int>::const_iterator i;
m["one"] = 1;
m["two"] = 2;
for (i=m.begin(); i!=m.end(); ++i) {
    std::cout << i->first << " -> "
        << i->second << "; ";
} // one -> 1; two -> 2;

Вставить элементы позволяют следующие методы:

> insert () — вставляет один или несколько элементов. Прототипы метода:
    pair<iterator, bool> insert(const pair &Val);
    iterator insert(const_iterator Where, const pair &Val);
    template<class It>
    void insert(It First, It Last);

Первый прототип вставляет экземпляр класса pair и возвращает объект того же класса. Через атрибут first будет доступен итератор, указывающий на вставленный элемент, а через атрибут second — логическое значение true, если элемент вставлен, и false в противном случае. Обратите внимание на то, что вставить можно только элемент, ключ которого не содержится в массиве. Пример:
typedef struct std::pair<std::string, int> PAIR;
std::map<std::string, int> m;
std::map<std::string, int>::const_iterator i;
std::pair< std::map<std::string, int>::iterator, bool > pr;
pr = m.insert(PAIR("one", 1));
if (pr.second) { // Проверка успешности вставки

Листинги на странице http://unicross.narod.ru/cpp/
std::cout << (pr.first)->first << " -> " << (pr.first)->second << std::endl;

} // one -> 1
else std::cout << "Error" << std::endl;

// Использование value_type вместо PAIR
m.insert(std::map<std::string, int>::value_type("two", 2));
for (i=m.begin(); i!=m.end(); ++i) {
    std::cout << i->first << " -> " << i->second << "; ";
} // one -> 1; two -> 2;

Второй прототип позволяет указать позицию вставки с помощью итератора. В качестве результата возвращается итератор, указывающий на вставленный элемент.
Третий прототип вставляет элементы из диапазона, ограниченного итераторами First и Last. Пример:

typedef struct std::pair<std::string, int> PAIR;
std::map<std::string, int> m1, m2;
std::map<std::string, int>::const_iterator i;
m1.insert(PAIR("0", 1));
m1.insert(PAIR("1", 2));
m1.insert(--m1.end(), PAIR("10", 3));
for (i=m1.begin(); i!=m1.end(); ++i) {
    std::cout << i->first << " -> " << i->second << "; ";
} // 0 -> 1; 1 -> 2; 10 -> 3;
std::cout << std::endl;
m2.insert(++m1.begin(), m1.end());
for (i=m2.begin(); i!=m2.end(); ++i) {
    std::cout << i->first << " -> " << i->second << "; ";
} // 1 -> 2; 10 -> 3;

// swap() — меняет элементы двух контейнеров местами. Прототип метода:

Листинги на странице http://unicross.narod.ru/cpp/
void swap(map &Right);
Пример:
typedef struct std::pair<std::string, int> PAIR;
std::map<std::string, int> m1, m2;
std::map<std::string, int>::const_iterator i;
m1.insert(PAIR("0", 1)); m1.insert(PAIR("1", 2));
m2.insert(PAIR("one", 1)); m2.insert(PAIR("two", 2));
m2.swap(m1);
for (i=m1.begin(); i!=m1.end(); ++i) {
    std::cout << i->first << " -> "
        << i->second << "; ";
} // one -> 1; two -> 2;
std::cout << std::endl;
for (i=m2.begin(); i!=m2.end(); ++i) {
    std::cout << i->first << " -> "
        << i->second << "; ";
} // 0 -> 1; 1 -> 2;

Удаление элементов
Для удаления элементов предназначены следующие методы:

> erase() — удаляет один элемент или элементы из диапазона. Прототипы метода:
size_type erase(const key_type &Keyval);
iterator erase(const_iterator Where);
iterator erase(const_iterator First, const_iterator Last);

Первый прототип удаляет элемент с указанным ключом. Пример:
typedef struct std::pair<std::string, int> PAIR;
std::map<std::string, int> m;
std::map<std::string, int>::const_iterator i;
m.insert(PAIR("one", 1)); m.insert(PAIR("two", 2));
m.erase("one");
for (i=m.begin(); i!=m.end(); ++i) {
    Листинги на странице http://unicross.narod.ru/cpp/
std::cout << i->first << " -> " << i->second << "; ";
} // two -> 2;

Второй прототип удаляет элемент на который указывает итератор. Пример:
typeid struct std::pair<std::string, int> PAIR;
std::map<std::string, int> m;
std::map<std::string, int>::const_iterator i;
m.insert(PAIR("one", 1)); m.insert(PAIR("two", 2));
m.erase(--m.end());
for (i=m.begin(); i!=m.end(); ++i) {
    std::cout << i->first << " -> "
               << i->second << "; ";
} // one -> 1;

Третий прототип удаляет элементы из диапазона, ограниченного итераторами First и Last. Пример:
typeid struct std::pair<std::string, int> PAIR;
std::map<std::string, int> m;
std::map<std::string, int>::const_iterator i;
m.insert(PAIR("1", 1)); m.insert(PAIR("2", 2));
m.insert(PAIR("3", 3)); m.insert(PAIR("4", 4));
m.erase(m.begin(), --m.end());
for (i=m.begin(); i!=m.end(); ++i) {
    std::cout << i->first << " -> "
               << i->second << "; ";
} // 3 -> 4;

clear() — удаляет все элементы. Прототип метода:
void clear();

empty() — возвращает значение true, если контейнер не содержит элементов, и false — в противном случае. Прототип метода:
bool empty() const;

Листинги на странице http://unicross.narod.ru/cpp/
size() — возвращает количество элементов в контейнере. Прототип метода:

    size_type size() const;

Пример использования методов clear(), empty() и size():

typedef struct std::pair<std::string, int> PAIR;
    std::map<std::string, int> m;
    m.insert(PAIR("0", 1)); m.insert(PAIR("1", 2));
    std::cout << m.size() << std::endl; // 2
    m.clear();
    if (m.empty()) {
        std::cout << "Нет элементов" << std::endl;
    }
    std::cout << m.size() << std::endl; // 0

max_size() — возвращает максимальное количество элементов, которое может содержаться в контейнере. Прототип метода:

    size_type max_size() const;

Доступ к элементам

К любому элементу можно обратиться как к элементу массива. Достаточно указать его ключ внутри квадратных скобок. Можно как получить значение, так и изменить его. Если производится присваивание значения по ключу и ключа не существует, то производится вставка элемента, если элемент уже существует, то его значение изменяется на новое. Если производится получение значения по ключу и ключа не существует, то элемент вставляется со значением по умолчанию для типа, а затем это значение возвращается. Пример:

    std::map<std::string, int> m;
    std::map<std::string, int>::const_iterator i;
    m["one"] = 1;  // Вставляем новый элемент
    m["two"] = 2;
    m["one"] = 100;  // Изменяем значение существующего элемента
    std::cout << m["one"] << std::endl; // 100
    std::cout << m["two"] << std::endl; // 2
    std::cout << m["key"] << std::endl; // 0 (элемент вставлен)

    for (i=m.begin(); i!=m.end(); ++i) {

Листинги на странице http://unicross.narod.ru/cpp/
std::cout << i->first << " -> "
    << i->second << "; ";
} // key -> 0; one -> 100; two -> 2;

Для доступа к элементам предназначены следующие методы:

- **at()** — возвращает ссылку на элемент, ключ которого соответствует значению Keyval. Метод позволяет как получить значение, так и изменить его. Если ключ не существует, то метод генерирует исключение. Прототипы метода:

  mapped_type & at(const key_type & Keyval);
  const mapped_type & at(const key_type & Keyval) const;

Пример:

  std::map<std::string, int> m;
  // m.at("one") = 1; // Ошибка. Ключ не существует
  m["one"] = 1;    // Вставляем новый элемент
  m.at("one") = 100; // Изменяем значение элемента
  std::cout << m.at("one") << std::endl; // 100

- **count()** — возвращает количество элементов, у которых ключ соответствуют значению Keyval. Прототип метода:

  size_type count(const key_type & Keyval) const;

Пример:

  std::map<std::string, int> m;
  m["one"] = 1;
  std::cout << m.count("one") << std::endl; // 1
  std::cout << m.count("two") << std::endl; // 0

- **find()** — возвращает итератор, установленный на элемент, ключ которого соответствует значению Keyval. Если элемент не найден, то метод возвращает итератор, указывающий на позицию после последнего элемента. Прототипы метода:

  iterator find(const key_type & Keyval);
  const_iterator find(const key_type & Keyval) const;

Пример:

  std::map<std::string, int> m;
  std::map<std::string, int>::iterator i;

Листинги на странице http://unicross.narod.ru/cpp/
m["one"] = 1;
i = m.find("one");
if (i != m.end()) {
    std::cout << i->second << std::endl; // 1
}

i = m.find("two");
if (i == m.end()) {
    std::cout << "No" << std::endl; // No
}

lower_bound() — возвращает итератор, установленный на элемент, ключ которого больше или равен значению keyval. Если элемент не найден, то метод возвращает итератор, указывающий на позицию после последнего элемента. Прототипы метода:

```
iterator lower_bound(const key_type &Keyval);
const_iterator lower_bound(const key_type &Keyval) const;
```

Пример:
```
std::map<int, int> m;
std::map<int, int>::iterator i;
m[0] = 0; m[1] = 1; m[4] = 4; m[5] = 5;
i = m.lower_bound(2);
if (i != m.end()) {
    std::cout << i->second << std::endl; // 4
}

i = m.lower_bound(1);
if (i != m.end()) {
    std::cout << i->second << std::endl; // 1
}
```

upper_bound() — возвращает итератор, установленный на элемент, ключ которого больше значения keyval. Если элемент не найден, то метод возвращает итератор, указывающий на позицию после последнего элемента. Прототипы метода:

```
iterator upper_bound(const key_type &Keyval);
```

Листинги на странице http://unicross.narod.ru/cpp/
const_iterator upper_bound(const key_type &Keyval) const;

Пример:
std::map<int, int> m;
std::map<int, int>::iterator i;
m[0] = 0; m[1] = 1; m[4] = 4; m[5] = 5;
i = m.upper_bound(1);
if (i != m.end()) {
    std::cout << i->second << std::endl; // 4
}

equal_range() — возвращает экземпляр класса pair. Через атрибут first будет доступен итератор, являющийся результатом выполнения метода lower_bound(), а через атрибут second — итератор, являющийся результатом выполнения метода upper_bound(). Прототипы метода:
pair<iterator, iterator> equal_range(const key_type &Keyval);
pair<const_iterator, const_iterator>
equal_range(const key_type &Keyval) const;

Пример:
typedef std::map<int, int> M;
M m;
std::pair<M::iterator, M::iterator> pr;
m[0] = 0; m[1] = 1; m[4] = 4; m[5] = 5;
pr = m.equal_range(1);   // Ключ существует
if (pr.first != m.end()) {
    std::cout << pr.first->second << std::endl; // 1
}
if (pr.second != m.end()) {
    std::cout << pr.second->second << std::endl; // 4
}
pr = m.equal_range(2);   // Ключ не существует
if (pr.first != m.end()) {
    std::cout << pr.first->second << std::endl; // 4
}

Листинги на странице http://unicross.narod.ru/cpp/
begin() — возвращает итератор, установленный на первый элемент контейнера. Прототипы метода:
iter\_\_iterator begin();
\_\_const iter\_\_iterator begin() const;

cbegin() — возвращает константный итератор, установленный на первый элемент контейнера. Прототип метода:
const iter\_\_iterator cbegin() const;

rbegin() — возвращает обратный итератор, установленный на последний элемент контейнера. Прототипы метода:
reverse iter\_\_iterator rbegin();
\_\_const reverse iter\_\_iterator rbegin() const;

crbegin() — возвращает константный обратный итератор, установленный на последний элемент контейнера. Прототип метода:
const reverse iter\_\_iterator crbegin() const;
end() — возвращает итератор, установленный на позицию после последнего элемента контейнера Прототипы метода:
iter\_\_iterator end();
\_\_const iter\_\_iterator end() const;

cend() — возвращает константный итератор, установленный на позицию после последнего элемента контейнера. Прототип метода:
const iter\_\_iterator cend() const;

rend() — возвращает обратный итератор, установленный на позицию перед первым элементом контейнера. Прототипы метода:
reverse iter\_\_iterator rend();
\_\_const reverse iter\_\_iterator rend() const;

crend() — возвращает константный обратный итератор, установленный на позицию перед первым элементом контейнера. Прототип метода:

Листинги на странице http://unicross.narod.ru/cpp/
const_reverse_iterator crend() const;

Класс multimap. Ассоциативный массив с повторяющимися ключами

Класс multimap реализует ассоциативный массив, в котором одному ключу могут соответствовать несколько значений. Класс содержит такие же методы, что и класс map, поэтому в этом разделе мы рассмотрим только основные операции и отличия между классами. Прежде чем использовать класс необходимо добавить заголовок:
#include <map>

Создание экземпляра класса multimap выполняется точно так же как и создание экземпляра класса map. Над двумя объектами класса multimap определены операции ==, !={}, <, <=, >, и >=. Кроме того, один объект можно присвоить другому объекту.

Вставка элементов производится с помощью метода insert(). Если ключ вставляемого элемента уже существует в массиве, то это не приводит к ошибке. В этом случае элемент все равно вставляется. Прототипы метода insert():

iterator insert(const pair & Val);
iterator insert(const_iterator Where, const pair & Val);
template<class It> void insert(It First, It Last);

Первый прототип вставляет экземпляр класса pair. В отличие от метода insert() класса map метод возвращает итератор, а не объект класса pair. Второй прототип позволяет указать позицию вставки с помощью итератора where. Третий прототип вставляет элементы из диапазона, ограниченного итераторами First и Last. Пример:

typedef struct std::pair<std::string, int> PAIR;
std::multimap<std::string, int> m;
std::multimap<std::string, int>::const_iterator i;
i = m.insert(PAIR("one", 1));
std::cout << i->first << " -> " << i->second << std::endl;
// one -> 1

// Использование value_type вместо PAIR
m.insert(std::multimap<std::string, int>::value_type("two", 2));
m.insert(m.begin(), PAIR("one", 6)); // Вставка одинакового ключа
for (i=m.begin(); i!=m.end(); ++i) { // Перебор элементов
    std::cout << i->first << " -> " << i->second << "; ";
} // one -> 6; one -> 1; two -> 2;

Листинги на странице http://unicross.narod.ru/cpp/
Обратите внимание на то, что для вставки и извлечения элементов нельзя использовать оператор [], так как ключи могут повторяться. Кроме того, класс multimap не имеет метода at(). Чтобы получить доступ к элементам следует воспользоваться методами find(), lower_bound(), upper_bound() и equal_range(). Получить количество элементов с указанным ключом позволяет метод count(). Пример:

typedef struct std::pair<std::string, int> PAIR;
std::multimap<std::string, int> m;
m.insert(PAIR("one", 1)); m.insert(PAIR("one", 2));
std::cout << m.count("one") << std::endl; // 2

Классы set и multiset. Множества

Класс set реализует множество, состоящее из уникальных элементов, а класс multiset — множество, в котором элементы могут повторяться. Прежде чем использовать классы необходимо добавить заголовок:

#include <set>

Работа с классами производится аналогичным образом, поэтому в этом разделе мы рассмотрим только возможности класса set.

Создание объекта

Создать экземпляр класса set можно следующими способами:

- объявить экземпляр класса без инициализации. Для этого перед названием переменной указывается название класса. После названия класса внутри угловых скобок задаются типы данных элемента. В этом случае объект не содержит элементов. Пример объявления без инициализации:

std::set<int> s;

Внутри угловых скобок после типа данных можно дополнительно указать функцию сравнения:

std::set<int, std::less<int> > s;

Внимание

Между закрывающими угловыми скобками пробел добавлен специально. Если пробел не добавить, то некоторые компиляторы считают это ошибкой.

- указать объект класса set внутри круглых скобок или после оператора =:

std::set<int> s1;

Листинги на странице http://unicross.narod.ru/cpp/
s1.insert(1); s1.insert(2);
std::set<int> s2(s1);
std::set<int> s3 = s1;

→ указать диапазон внутри контейнера с помощью итераторов. В первом параметре
передается итератор, указывающий на начало диапазона, а во втором параметре —
итератор, указывающий на конец диапазона. Пример:
std::set<int> s1;
s1.insert(1); s1.insert(2);
std::set<int> s2(s1.begin(), s1.end());

Над двумя объектами класса set определены операции ==, !=, <, <=, > и >=. Кроме
того, один объект можно присвоить другому объекту. Пример:
std::set<int> s1, s2;
std::set<int>::const_iterator i;
s1.insert(1); s1.insert(2);
s2 = s1;
for (i=s2.begin(); i!=s2.end(); ++i) {
    std::cout << *i << " ";
} // 1 2

Вставка элементов
Вставить элементы позволяют следующие методы:

→ insert () — вставляет один или несколько элементов. Прототипы метода:
pair<iterator, bool> insert(const value_type &Val);
iterator insert(const_iterator Where, const value_type &Val);
template<class It>
void insert(It First, It Last);

Первый прототип вставляет значение и возвращает объект класса pair. Через атрибут first будет доступен итератор, указывающий на вставленный элемент, а через атрибут second — логическое значение true, если элемент вставлен, и false
в противном случае. Пример:
std::set<int> s;
std::set<int>::const_iterator i;
std::pair<std::set<int>::iterator, bool> pr;

Листинги на странице http://unicross.narod.ru/cpp/
s.insert(1);
pr = s.insert(2);
if (pr.second) { // Проверка успешности вставки
    std::cout << *(pr.first) << std::endl; // 2
}
else std::cout << "Error" << std::endl;
for (i=s.begin(); i!=s.end(); ++i) {
    std::cout << *i << " ";
} // 1 2

Обратите внимание на то, что метод insert() в классе multiset возвращает итератор, а не объект класса pair. Прототип метода в классе multiset:

```
iterator insert(const value_type &Val);
```

Второй прототип позволяет указать позицию вставки с помощью итератора. В качестве результата возвращается итератор, указывающий на вставленный элемент. Третий прототип вставляет элементы из диапазона, ограниченного итераторами First и Last. Пример:

```
std::set<int> s1, s2;
std::set<int>::const_iterator i;
s1.insert(1); s1.insert(2);
s1.insert(s1.begin(), 3);
for (i=s1.begin(); i!=s1.end(); ++i) {
    std::cout << *i << " ";
} // 1 2 3
std::cout << std::endl;
```

см. урок 1 — меняет элементы двух контейнеров местами. Прототип метода:

```
void swap(set &Right);
```

Пример:

Листинги на странице http://unicross.narod.ru/cpp/
std::set<int> s1, s2;
std::set<int>::const_iterator i;
s1.insert(1); s1.insert(2);
s2.insert(3); s2.insert(4);
s2.swap(s1);
for (i=s1.begin(); i!=s1.end(); ++i) {
    std::cout << *i << " ";
} // 3 4
std::cout << std::endl;
for (i=s2.begin(); i!=s2.end(); ++i) {
    std::cout << *i << " ";
} // 1 2

Удаление элементов

Для удаления элементов предназначены следующие методы:

→ erase() — удаляет один элемент или элементы из диапазона. Прототипы метода:

size_type erase(const key_type &Keyval);
iterator erase(const_iterator Where);
iterator erase(const_iterator First, const_iterator Last);

Первый прототип удаляет элемент с указанным значением. Пример:
std::set<int> s;
std::set<int>::const_iterator i;
s.insert(1); s.insert(2);
s.erase(1);
for (i=s.begin(); i!=s.end(); ++i) {
    std::cout << *i << " ";
} // 2

Второй прототип удаляет элемент на который указывает итератор. Пример:
std::set<int> s;
std::set<int>::const_iterator i;
s.insert(1); s.insert(2);

Листинги на странице http://unicross.narod.ru/cpp/
s.erase(--s.end());
for (i=s.begin(); i!=s.end(); ++i) {
    std::cout << *i << " ";
} // 1

Третий прототип удаляет элементы из диапазона, ограниченного итераторами First и Last. Пример:

typedef struct std::pair<std::string, int> PAIR;
std::set<std::string, int> m;
std::set<std::string, int>::const_iterator i;
m.insert(PAIR("0", 1)); m.insert(PAIR("1", 2));
m.insert(PAIR("2", 3)); m.insert(PAIR("3", 4));
m.erase(m.begin(), --m.end());
for (i=m.begin(); i!=m.end(); ++i) {
    std::cout << i->first << " -> "
            << i->second << "; ";
} // 3 -> 4;

clear() — удаляет все элементы. Прототип метода:
void clear();

empty() — возвращает значение true, если контейнер не содержит элементов, и false — в противном случае. Прототип метода:
bool empty() const;

size() — возвращает количество элементов в контейнере. Прототип метода:
size_type size() const;

Пример использования методов clear(), empty() и size():
std::set<int> s;
std::set<int>::const_iterator i;
s.insert(1); s.insert(2);
std::cout << s.size() << std::endl; // 2
s.clear();
if (s.empty()) {

Листинги на странице http://unicross.narod.ru/cpp/
std::cout << "Нет элементов" << std::endl;
}
std::cout << s.size() << std::endl; // 0

- `max_size()` — возвращает максимальное количество элементов, которое может содержаться в контейнере. Прототип метода:
  
  ```cpp
  size_type max_size() const;
  ```

### Доступ к элементам

Для доступа к элементам предназначены следующие методы:

- `count()` — возвращает количество элементов, у которых ключ соответствуют значению `Keyval`. Прототип метода:
  
  ```cpp
  size_type count(const key_type &Keyval) const;
  ```

  Пример:
  
  ```cpp
  std::set<int> s;
  s.insert(1); s.insert(2);
  std::cout << s.count(2) << std::endl; // 1
  ```

- `find()` — возвращает итератор, установленный на элемент, ключ которого соответствует значению `Keyval`. Если элемент не найден, то метод возвращает итератор, указывающий на позицию после последнего элемента. Прототипы метода:

  ```cpp
  iterator find(const key_type &Keyval);
  const_iterator find(const key_type &Keyval) const;
  ```

  Пример:
  
  ```cpp
  std::set<int> s;
  std::set<int>::iterator i;
  s.insert(1); s.insert(2);
  i = s.find(1);
  if (i != s.end()) {
    std::cout << *i << std::endl; // 1
  }
  i = s.find(3);
  if (i == s.end()) {
  ```

Листинги на странице http://unicross.narod.ru/cpp/
std::cout << "No" << std::endl; // No

lower_bound() — возвращает итератор, установленный на элемент, ключ которого больше или равен значению keyval. Если элемент не найден, то метод возвращает итератор, указывающий на позицию после последнего элемента. Прототипы метода:

```cpp
iterator lower_bound(const key_type &keyval);
const_iterator lower_bound(const key_type &keyval) const;
```

Пример:
```cpp
std::set<int> s;
std::set<int>::iterator i;
s.insert(0); s.insert(1); s.insert(4); s.insert(5);
i = s.lower_bound(2);
if (i != s.end()) {
    std::cout << *i << std::endl; // 4
}
i = s.lower_bound(1);
if (i != s.end()) {
    std::cout << *i << std::endl; // 1
}
```

upper_bound() — возвращает итератор, установленный на элемент, ключ которого больше значения keyval. Если элемент не найден, то метод возвращает итератор, указывающий на позицию после последнего элемента. Прототипы метода:

```cpp
iterator upper_bound(const key_type &keyval);
const_iterator upper_bound(const key_type &keyval) const;
```

Пример:
```cpp
std::set<int> s;
std::set<int>::iterator i;
s.insert(0); s.insert(1); s.insert(4); s.insert(5);
i = s.upper_bound(1);
if (i != s.end()) {
```

equal_range() — возвращает экземпляр класса pair. Через атрибут first будет доступен итератор, являющийся результатом выполнения метода lower_bound(), а через атрибут second — итератор, являющийся результатом выполнения метода upper_bound(). Прототипы метода:
pair<iterator, iterator> equal_range(const key_type &Keyval);
pair<const_iterator, const_iterator>
    equal_range(const key_type &Keyval) const;

Пример:
typedef std::set<int> Set;
Set s;
std::pair<Set::iterator, Set::iterator> pr;
s.insert(0); s.insert(1); s.insert(4); s.insert(5);
pr = s.equal_range(1); // Ключ существует
if (pr.first != s.end()) {
    std::cout << *(pr.first) << std::endl; // 1
}
if (pr.second != s.end()) {
    std::cout << *(pr.second) << std::endl; // 4
}
pr = s.equal_range(2); // Ключ не существует
if (pr.first != s.end()) {
    std::cout << *(pr.first) << std::endl; // 4
}
if (pr.second != s.end()) {
    std::cout << *(pr.second) << std::endl; // 4
}

begin() — возвращает итератор, установленный на первый элемент контейнера.
Прототипы метода:

Листинги на странице http://unicross.narod.ru/cpp/
Класс priority_queue. Очередь с приоритетами

Класс priority_queue реализует очередь с приоритетами на основе последовательного контейнера. По умолчанию в качестве контейнера используется класс vector. При получении значения возвращается элемент с наивысшим приоритетом (наибольшим значением). По умолчанию для сравнения элементов Листинги на странице http://unicross.narod.ru/cpp/
используется функтор less. Объявление класса priority_queue выглядит следующим образом:

```cpp
template <class T, class Container=vector<T>,
          class Compare=less<typename Container::value_type> >
class priority_queue;
```

Прежде чем использовать класс необходимо добавить заголовок:

```cpp
#include <queue>
```

Создать экземпляр класса priority_queue можно следующими способами:

- объявить экземпляр класса без инициализации. Для этого перед названием переменной указывается название класса. После названия класса внутри угловых скобок задается тип данных. В этом случае объект не содержит элементов. Пример объявления без инициализации:

```cpp
std::priority_queue<int> q;
```

Внутри угловых скобок после типа данных можно дополнительно указать тип контейнера и функцию сравнения. По умолчанию в качестве контейнера используется вектор, а сравнение производится с помощью функтора less. Вместо вектора можно использовать класс deque. Пример указания класса deque и функтора greater:

```cpp
std::priority_queue<int, std::deque<int>,
std::greater<int> > q;
```

**Внимание**

Между закрывающими угловыми скобками пробел добавлен специально. Если пробел не добавить, то некоторые компиляторы считают это ошибкой.

- указать объект класса priority_queue внутри круглых скобок или после оператора =:

```cpp
std::priority_queue<int> q1;
q1.push(1); q1.push(2);
std::priority_queue<int> q2(q1);
std::cout << q2.top() << std::endl; // 2
std::priority_queue<int> q3 = q1;
std::cout << q3.top() << std::endl; // 2
```

→ указать диапазон внутри контейнера с помощью итераторов. В первом параметре
передается итератор, указывающий на начало диапазона, а во втором параметре —
итератор, указывающий на конец диапазона. Пример:

```
std::vector<int> v;
v.push_back(1); v.push_back(2);
std::priority_queue<int> q(v.begin(), v.end());
std::cout << q.top() << std::endl; // 2
```

Один объект можно присвоить другому объекту. Пример:

```
std::priority_queue<int> q1, q2;
q1.push(1); q1.push(2);
q2 = q1;
std::cout << q2.top() << std::endl; // 2
```

Класс priority_queue содержит следующие методы:

→ push() — добавляет элемент. Прототип метода:

```
void push(const value_type &Val);
```

→ top() — возвращает ссылку на элемент с наивысшим приоритетом. Элемент из
очереди не удаляется. Прототипы метода:

```
reference top();
const_reference top() const;
```

→ pop() — удаляет элемент с наивысшим приоритетом. Прототип метода:

```
void pop();
```

→ swap() — меняет элементы двух контейнеров местами. Прототип метода:

```
void swap(priority_queue &Right);
```

→ size() — возвращает количество элементов в очереди. Прототип метода:

```
size_type size() const;
```

→ empty() — возвращает значение true, если очередь не содержит элементов, и
false — в противном случае. Прототип метода:

```
bool empty() const;
```

Основные операции над объектом класса priority_queue показаны в листинге 13.8.
Листинг 13.8. Класс priority_queue

```cpp
#include <iostream>
#include <queue>

int main() {
    std::priority_queue<int> q1, q2;
    q1.push(1); q1.push(2);
    q2.push(3); q2.push(4);
    std::cout << q1.top() << std::endl; // 2
    std::cout << q2.top() << std::endl; // 4
    q1.swap(q2);
    std::cout << q1.top() << std::endl; // 4
    std::cout << q2.top() << std::endl; // 2
    q1.pop();
    std::cout << q1.top() << std::endl; // 3
    if (q1.empty()) {
        std::cout << "Нет элементов" << std::endl;
    }
    else {
        std::cout << q1.size() << std::endl; // 1
    }
    std::cin.get();
    return 0;
}
```

Класс queue. Очередь

Класс queue реализует очередь на основе последовательного контейнера. По умолчанию в качестве контейнера используется класс deque. При получении значения возвращается элемент, который был добавлен раньше других элементов. Объявление класса queue выглядит следующим образом:

```cpp
template <class T, class Container = deque<T>> class queue;
```

Прежде чем использовать класс необходимо добавить заголовок:

```cpp
#include <queue>
```

Создать экземпляр класса queue можно следующими способами:

Листинги на странице http://unicross.narod.ru/cpp/
объявить экземпляр класса без инициализации. Для этого перед названием переменной указывается название класса. После названия класса внутри угловых скобок задается тип данных. В этом случае объект не содержит элементов. Пример объявления без инициализации:
std::queue<int> q;

Внутри угловых скобок после типа данных можно дополнительно указать тип контейнера. По умолчанию в качестве контейнера используется класс deque. Вместо класса deque можно использовать класс list. Вектор в этом случае использовать нельзя, так как у него нет метода для удаления элемента из начала контейнера. Пример указания класса list:
std::queue<int, std::list<int>> q;

указать объект класса queue внутри круглых скобок или после оператора =:
std::queue<int> q1;
q1.push(1); q1.push(2);
std::queue<int> q2(q1);
std::cout << q2.front() << std::endl; // 1
std::queue<int> q3 = q1;
std::cout << q3.front() << std::endl; // 1

Над двумя объектами класса queue определены операции ==, !==, <, <=, > и >=. Кроме того, один объект можно присвоить другому объекту. Пример:
std::queue<int> q1, q2;
q1.push(1); q1.push(2);
q2 = q1;
std::cout << q2.front() << std::endl; // 1

Класс queue содержит следующие методы:
push() — добавляет элемент в конец очереди. Прототип метода:
void push(const value_type &val);

front() — возвращает ссылку на первый элемент в очереди. Элемент из очереди не удаляется. Прототипы метода:
reference front();
const_reference front() const;

Листинги на странице http://unicross.narod.ru/cpp/
→ back() — возвращает ссылку на последний элемент в очереди. Элемент из очереди не удаляется. Прототипы метода:
    reference back();
    const_reference back() const;
→ pop() — удаляет первый элемент из очереди. Прототип метода:
    void pop();
→ swap() — меняет элементы двух контейнеров местами. Прототип метода:
    void swap(queue &Right);
→ size() — возвращает количество элементов в очереди. Прототип метода:
    size_type size() const;
→ empty() — возвращает значение true, если очередь не содержит элементов, и false — в противном случае. Прототип метода:
    bool empty() const;
Основные операции над объектом класса queue показаны в листинге 13.9.

```cpp
#include <iostream>
#include <queue>

int main() {
    std::queue<int> q1, q2;
    q1.push(1); q1.push(2);
    q2.push(3); q2.push(4);
    std::cout << q1.front() << std::endl;  // 1
    std::cout << q2.front() << std::endl;  // 3
    q1.swap(q2);
    std::cout << q1.front() << std::endl;  // 3
    std::cout << q2.front() << std::endl;  // 1
    q1.pop();
    std::cout << q1.front() << std::endl;  // 4
    if (q1.empty()) {
        std::cout << "Нет элементов" << std::endl;
    }
}
```

Листинги на странице http://unicross.narod.ru/cpp/
else {
  std::cout << q1.size() << std::endl;  // 1
}
std::cin.get();
return 0;

Класс stack. Стек

Класс stack реализует стек на основе последовательного контейнера. По умолчанию в качестве контейнера используется класс deque. При получении значения возвращается элемент, который был добавлен позже других элементов. Объявление класса stack выглядит следующим образом:

template <class T, class Container=deque<T> > class stack;

Прежде чем использовать класс необходимо добавить заголовок:
#include <stack>

Создать экземпляр класса stack можно следующими способами:

→ объявить экземпляр класса без инициализации. Для этого перед названием переменной указывается название класса. После названия класса внутри угловых скобок задается тип данных. В этом случае объект не содержит элементов. Пример объявления без инициализации:

std::stack<int> s;

Внутри угловых скобок после типа данных можно дополнительно указать тип контейнера. По умолчанию в качестве контейнера используется класс deque. Вместо класса deque можно использовать классы list и vector. Пример указания класса list:

std::stack<int, std::list<int> > s;

→ указать объект класса stack внутри круглых скобок или после оператора :=

std::stack<int> s1;
s1.push(1); s1.push(2);
std::stack<int> s2(s1);
std::cout << s2.top() << std::endl;  // 2
std::stack<int> s3 = s1;

Листинги на странице http://unicross.narod.ru/cpp/
Над двумя объектами класса stack определены операции ==, !=, <, <=, > и >=. Кроме того, один объект можно присвоить другому объекту. Пример:

```cpp
std::stack<int> s1, s2;
s1.push(1); s1.push(2);
s2 = s1;
std::cout << s2.top() << std::endl; // 2
```

Класс stack содержит следующие методы:

- **push()** — добавляет элемент в стек. Прототип метода:
  ```cpp
  void push(const value_type &Val);
  ```

- **top()** — возвращает ссылку на элемент, расположенный на вершине стека. Элемент из стека не удаляется. Прототипы метода:
  ```cpp
  reference top();
  const_reference top() const;
  ```

- **pop()** — удаляет элемент, расположенный на вершине стека. Прототип метода:
  ```cpp
  void pop();
  ```

- **swap()** — меняет элементы двух стеков местами. Прототип метода:
  ```cpp
  void swap(stack &Right);
  ```

- **size()** — возвращает количество элементов в стеке. Прототип метода:
  ```cpp
  size_type size() const;
  ```

- **empty()** — возвращает значение true, если стек не содержит элементов, и false — в противном случае. Прототип метода:
  ```cpp
  bool empty() const;
  ```

Основные операции над объектом класса stack показаны в листинге 13.10.

### Листинг 13.10. Класс stack

```cpp
#include <iostream>
#include <stack>

int main() {
  std::stack<int> s1, s2;

  Листинги на странице http://unicross.narod.ru/cpp/
```
s1.push(1); s1.push(2);
s2.push(3); s2.push(4);
std::cout << s1.top() << std::endl;  // 2
std::cout << s2.top() << std::endl;  // 4
s1.swap(s2);
std::cout << s1.top() << std::endl;  // 4
std::cout << s2.top() << std::endl;  // 2
s1.pop();
std::cout << s1.top() << std::endl;  // 3
if (s1.empty()) {
    std::cout << "Нет элементов" << std::endl;
}
else {
    std::cout << s1.size() << std::endl;  // 1
}
std::cin.get();
return 0;