[1. Файлы в C++.](#_Toc515770655)

[1.1. Структура стандартной библиотеки ввода-вывода.](#_Toc515770656)

[1.2. Предопределенные потоки](#_Toc515770657)

[1.3. От файла STREAM.H к файлу IOSTREAM.H](#_Toc515770658)

[1.4. Список классов iostream.](#_Toc515770659)

[1.5. Простота ввода/вывода в С ++](#_Toc515770660)

[1.5.1. Потоки cin, cout и сегг](#_Toc515770661)

[1.5.2. Операции выделения ( >> ) и вставки ( << )](#_Toc515770662)

[1.6. Опции ввода/вывода в С/С++.](#_Toc515770663)

[1.7. Операции и методы классов](#_Toc515770664)

[1.7.1. Вывод символов в C++](#_Toc515770665)

[1.7.2. Преобразование системы счисления в C++](#_Toc515770666)

[1.7.3. Форматирование строк в C++](#_Toc515770667)

[1.7.4. Форматирование чисел в C++](#_Toc515770668)

[1.7.5. Файловый ввод и вывод в C++](#_Toc515770669)

[1.8. Классы ios\_base и ios.](#_Toc515770670)

[1.9. Потоки вывода.](#_Toc515770671)

[1.9.1. Функции open, close и is\_open](#_Toc515770672)

[1.9.2. Функция flush](#_Toc515770673)

[1.9.3. Функция put](#_Toc515770674)

[1.9.4. Функция seekp и tellp](#_Toc515770675)

[1.9.5. Функция write](#_Toc515770676)

[1.9.6. Функция str](#_Toc515770677)

[1.9.7. Классы потокового вывода.](#_Toc515770678)

[1.10. Потоки ввода.](#_Toc515770679)

[1.10.1. Функция gcount](#_Toc515770680)

[1.10.2. Функция get](#_Toc515770681)

[1.10.3. Функция getline](#_Toc515770682)

[1.10.4. Функция ignore](#_Toc515770683)

[1.10.5. Функция peek](#_Toc515770684)

[1.10.6. Функция putback](#_Toc515770685)

[1.10.7. Функция read](#_Toc515770686)

[1.10.8. Функция seekg](#_Toc515770687)

[1.10.9. Функция sync](#_Toc515770688)

[1.10.10. Функция unget](#_Toc515770689)

[1.10.11. Функция str](#_Toc515770690)

[1.10.12. Классы потокового ввода.](#_Toc515770691)

[1.11. Потоки ввода-вывода.](#_Toc515770692)

[1.12. Классы буферизированных потоков.](#_Toc515770693)

[1.13. Класс строковых потоков.](#_Toc515770694)

[1.14. Форматирование потока.](#_Toc515770695)

[1.15. Резидентные в памяти потоки](#_Toc515770696)

[1.16. Буферы и синхронизация.](#_Toc515770697)

[1.17. Условные признаки файлов в C++](#_Toc515770698)

[1.18. Опрос и установка состояния потока](#_Toc515770699)

[1.19. Ошибки потоков](#_Toc515770700)

[1.20. Часто применяемые функции](#_Toc515770701)

[1.21. Двоичные файлы.](#_Toc515770702)

[1.22. Объединение программ на С и С++. Использование спецификатора extern "С".](#_Toc515770703)

[1.23. Манипуляторы.](#_Toc515770704)

[1.23.1. Манипуляторы без параметров.](#_Toc515770705)

[1.23.2. Манипуляторы с одним параметром.](#_Toc515770706)

[1.23.3. Манипуляторы с несколькими параметрами.](#_Toc515770707)

1. Файлы в C++.

http://www.itmathrepetitor.ru/s-lekciya-9-rabota-s-potokami-v-s/

## Структура стандартной библиотеки ввода-вывода.

В языке программирования С++ стандартная библиотека ввода-вывода организована как иерархия шаблонов классов. Эта библиотека содержит два стандартных множества конкретизированных шаблонов этой иерархии: одно для работы с символами типа **char**, другое для работы с символами типа **wchar\_t**. Классы для работы с символами типа **wchar\_t** имеют те же имена, что и классы для работы с символами типа **char**, но с префиксом **w**.

Ниже перечислены заголовочные файлы, которые содержат интерфейсы классов из стандартной библиотеки ввода-вывода:

|  |  |
| --- | --- |
| <ios> | ios\_base, ios |
| <istream> | istream |
| <ostream> | ostream |
| <iostream> | iostream, cin, cout, cerr |
| <fstream> | ifstream, fstream, ofstream, filebuf |
| <sstream> | istringstream, stringstream, ostringstream, stringbuf |
| <streambuf> | streambuf |

## Предопределенные потоки

Теперь самое время заняться изучением реализации потоков ввода-вывода. Наряду с функциями библиотеки времени выполнения, предназначенными для ввода-вывода, в C++ включен дополнительный набор объектно-ориентированных подпрограмм ввода-вывода. **Главным преимуществом системы ввода-вывода C++ является то, что она может перегружаться для создаваемых вами классов**. В настоящее время используются две версии библиотеки ввода-вывода C++: старая или традиционная, основанная на предварительном варианте стандарта C++, и новая, определенная международным стандартом Standard C++. Внешне эти версии почти не различаются. Однако внутренняя их реализация совершенно различна. Новая версия библиотеки широко использует шаблоны классов. Поскольку мы еще не рассмотрели использование шаблонов классов, эти детали реализации мы будем опускать.

**Поток ввода-вывода** - это логическое устройство, предназначенное для приема и выдачи информации пользователю. Поток связан с физическим устройством с помощью системы ввода-вывода C++. Тем самым поток обеспечивает пользователю единый интерфейс при работе с системой ввода-вывода. Это означает, что, например, для вывода информации на экран монитора и для записи ее в файл используется одна и та же функция. **Когда программа на C++ начинает выполняться, автоматически** **создаются восемь предопределенных стандартных потоков. Эти стандартные потоки связаны со стандартными файлами stdin, stdout и stderr языка С**.

**Предопределенные стандартные потоки и связанные с ними стандартные файлы языка С**

|  |  |  |
| --- | --- | --- |
| **Поток для "узких" символов** | **Поток для "широких" символов** | **Стандартный файл языка С** |
| cin | wcin | stdin |
| cout | wcout | stdout |
| cerr | wcerr | stderr |
| clog | wclog | stderr |

Как и стандартные файлы языка С, все эти потоки по умолчанию связаны с терминалом. Объект **cin** управляет вводом из буфера потока, связанного с объектом **stdin**, объявленным в <cstdio>. По умолчанию эти два потока синхронизированы. Объект **cout** управляет буфером потока, связанным с объектом **stdout**. объявленным в <cstdio>. По умолчанию эти два потока также синхронизированы. Поток **clog** - это просто буферизованная версия потока **cerr**. В буферизованной версии потока запись на реальное внешнее устройство делается, только когда буфер полон. Поэтому **clog** является более эффективным для перенаправления вывода в файл, в то время как **cerr** используется главным образом для вывода на экран терминала.

Система ввода-вывода содержит **две иерархии классов**: **одну**, предназначенную для работы с ASCII-символами, имеющими длину 8 бит, и **другую**, предназначенную для работы с UNICODE-символами, имеющими длину 16 бит. Символы первого набора называются "**узкими**", а второго - "**широкими**". Стандартные потоки, перечисленные в столбце 2 таблицы, предназначены для работы с "широкими " символами. На самом деле ситуация несколько сложнее, поскольку шаблонные классы представляют собой просто классы, содержащие типы данных в качестве параметров. Поэтому имена классов у каждой из иерархий одинаковы, но предназначены они для использования с разными наборами символов, которые им передаются в качестве параметра. В дальнейшем мы будем говорить о классах этих иерархий, употребляя только их имена. В основе иерархии потоковых классов лежит класс **basic\_ios**, который в качестве своего подобъекта абстрактный класс **basic\_streambuf**. Последний является базовым классом для создания буфера потока, который управляет передачей элементов в поток и из него для специализированных видов потоков.

Класс **basic\_ios** является базовым для нескольких производных классов, среди которых классы **basic\_istream**, **basic\_ostream** и **basic\_iostream**. Он содержит наиболее общие функции, необходимые для всех потоков, и обслуживает информацию о состоянии, которая отражает целостность потока и буфер потока. Этот класс также обслуживает связь потоковых классов с классами буферов потоков с помощью функции-члена **rdbuf ()**. Классы, производные от **basic\_ios**, специализируют операции ввода-вывода. В свою очередь, **basic\_ios** использует класс **ios\_base**, который также является базовым классом для всех потоковых классов. Он не зависит от типа символов и инкапсулирует информацию, необходимую для всех потоков. Эта информация включает в себя:

* управляющую информацию для синтаксического анализа и форматирования;
* дополнительную информацию для нужд пользователя (которая предоставляет путь для расширения потоков, как мы увидим позже):
* наполнение потока региональными (или локальными) символами.

Кроме того, этот класс определяет несколько типов данных, используемых всеми потоковыми классами, такие как флаги форматирования, биты состояния, режимы открытия файлов и т.д.

С точки зрения UNIX клавиатура и экран -- это файлы. Есть три стандартные константы:

FILE \*stdin

FILE \*stdout

FILE \*stderr

Это три стандартных заранее открытых файла.

**stdin** - это стандартный файл (поток) ввода, а **stdout** -- стандартный файл (поток) вывода. Таким образом:

scanf(...) в точности эквивалентно fscanf(stdin, ...)

printf(...) в точности эквивалентно fprintf(stdout, ...)

Такой гибкостью можно воспользоваться при написании программы для работы с файлами. Например, для отладки программы можно выводить информацию на экран монитора, а не в файл. Для этого в начале работы с файлом пишем две строчки:

//FILE \*f = fopen(...);

FILE \*f = stdin;

При этом код программы будет содержать такие функции: fscanf(f, ...) или fprintf(f, ...). А когда отладка законичится, просто снимаем/ставим соответствующие комментарии в двух строчках программы.

**stderr** -- это стандартный файл (поток) ошибок. По умолчанию выводит данные на экран.

Но существует заметное отличие этого "файла" от **stdin** и **stdout: stderr** -- небуферизованный файл (поток). Поэтому в этот файл (поток) все байты уходят без "задержки", которая могла бы возникнуть при буферизированном подходе. Понятно, что польза от этого подхода заключается в том, что вместо кода:

fprintf(stdout, ...);

fflush(stdout);

мы пишем:

fprintf(stderr, ...);

В языке C++ объекты для работы с файлами называются потоками (**streams**). В данном случае слово "поток" означает то же самое, что и "файл" в языке C.

Классы для работы с файлами в языке C++ называются **std::istream** и **std::ostream** для ввода и вывода соответственно.

## От файла STREAM.H к файлу IOSTREAM.H

Одним из самых впечатляющих усовершенствований компилятора C++ является новая библиотека ввода/вывода, библиотека **iostream**. Средства ввода/вывода не включены в сам язык C++; они реализованы на этом языке и поставляются как компоненты стандартной библиотеки C++, что позволяет по мере необходимости развивать возможности ввода/вывода. Эта новая библиотека iostream заменяет предыдущую версию библиотеки ввода/вывода, которая называлась библиотекой потоков версии Release 1.2 (описана в книге Страуструпа "Язык программирования C++" — Stroustrup, The C++ Programming Language [Addison-Wesley, Reading, Massachusetts: 1990]).

На нижнем уровне C++ интерпретирует файл как последовательность или как поток байтов. На этом уровне концепция типов данных отсутствует. Одна компонента библиотеки ввода/вывода отвечает за передачу этих байтов. Однако с точки зрения пользователя файл состоит из смешанной последовательности цифр и букв или, что возможно, из объектов класса. Другая компонента библиотеки ввода/вывода обеспечивает интерфейс между двумя этими уровнями. В библиотеке **iostream** предопределен набор операций для выполнения считывания и записи встроенных типов данных. Также библиотека обеспечивает пользовательские расширения для обработки классов.

**Базовые операции ввода обеспечивает класс istream, а операции вывода —** **класс ostream. Двунаправленный ввод/вывод поддерживается классом iostream, который является производным от istream и ostream.** Для пользователя предопределены четыре потоковых объекта:

|  |  |
| --- | --- |
| cin | Объект класса istream, связанный со стандартным вводом |
| cout | Объект класса ostream, связанный со стандартным выводом |
| сегг | Небуферизированный выходной объект класса ostream, связанный со стандартным устройством ошибок |
| clog | Буферизированный выходной объект класса ostream, связанный со стандартным устройством ошибок |

В любую программу, использующую библиотеку **iostream**, должен быть включен заголовочный файл **iostream.h**. Поскольку iostream.h рассматривает stream.h как псевдоним, программы, написанные с использованием stream.h, могут требовать изменений, а могут и не требовать — в зависимости от конкретных структур.

Новую библиотеку ввода/вывода можно также использовать для выполнения файловых операций ввода и вывода. Связать файл с программой можно при помощи описаний трех следующих классов:

|  |  |
| --- | --- |
| fstream | Производный от iostream, связывает файл с приложением для считывания и записи |
| ifstream | Производный от istream, связывает файл с приложением только для считывания |
| ofstream | Производный от ostream, связывает файл с приложением только для записи |

Код для открытия файла и его чтения выглядит примерно так:

ifstream ifs;

ifs.open("file1.txt");

// далее с помощью оператора >> можно читать из файла, если он успешно открылся;

Аналогично можно использовать конструктор с параметром: ifs("file1.txt"); после чего создается объект и открывается по возможности файл.

В классе **istream** есть метод **close()**, который закрывает файл (на подобие работы с файлами в языке C). Однако вызывать этот метод необязательно. Дело в том, что в деструкторе класса этот метод вызовется автоматически.

Работа с объектами классов ofstream и ofstream и fstream осуществляется по аналогичному сценарию.

## Список классов iostream.

За исключением классов буферизированных потоков, все объекты ввода/вывода, описанные в библиотеке классов iostream, используют один и тот же абстрактный базовый класс потоков, называемый ios. Как показано в таблице, эти производные классы делятся на четыре категории.

Классы потокового ввода

|  |  |
| --- | --- |
| istream | Используется как универсальное средство ввода или как родительский класс для других производных классов потокового ввода |
| ifstream | Используется для ввода из файлов |
| istream\_withassign | Используется для ввода из потока cin |
| istrstream | Используется для ввода строк |

Классы потокового вывода

|  |  |
| --- | --- |
| ostream | Используется как универсальное средство потокового вывода или как родительский класс для других производных классов потокового вывода |
| ofstream | Используется для вывода в файл |
| ofstream\_withassign | Используется для вывода в потоки cout, cerr и clog |
| ostrstream | Используется для вывода строк |

Классы потокового ввода/вывода

|  |  |
| --- | --- |
| iostream | Используется как универсальное средство ввода и вывода или как родительский класс для других производных классов потокового ввода/вывода |
| fstream | Класс файлового потокового ввода/вывода |
| strstream | Класс потокового ввода/вывода строк |
| stdiostream | Стандартный класс потокового ввода/вывода |

Классы буферизированных потоков

|  |  |
| --- | --- |
| streambuf | Используется как родительский класс для производных объектов |
| filebuf | Класс буферизированных потоков для дисковых файлов |
| strstreambuf | Класс буферизированнных потоков для строк |
| stdiobuf | Класс буферизированных потоков для стандартного файлового ввода/вывода |

На рисунке показаны взаимоотношения между перечисленными классами потока ios.

03_01

Для обработки реального ввода/вывода во всех производных от ios классах **iostream** используется объект класса **streambuf**. В библиотеке классов **iostream** используются следующие три производных класса для буферизированных потоков:

|  |  |
| --- | --- |
| filebuf | Обеспечивает буферизированный ввод/вывод дисковых файлов |
| strstreambuf | Поддерживает хранящийся в памяти байтовый массив, содержащий данные потока |
| stdiobuf | Обеспечивает буферизированный дисковый ввод/вывод, при этом всю буферизацию выполняет стандартная система ввода/вывода |

Напомним, что **все производные классы обычно являются расширениями наследуемых определений родительского класса**. Этим объясняется тот факт, что в некотором производном классе могут часто использоваться операции или методы класса, которые непосредственно не описываются в этом классе.

Это означает, что **если нужно хорошо понять работу любого производного класса, то следует вернуться к описанию базового или родительского класса**.

Во всех последующих примерах используется некий производный класс, полученный на основе какого-нибудь родительского класса. В одних примерах задействуются методы производного класса, в других — наследуемые характеристики. Эти примеры помогут понять многие преимущества производных классов и наследуемых свойств. Поначалу эти концепции могут показаться сложными, однако вскоре становится ясным, что для того, чтобы воспользоваться функциональными возможностями какого-либо ранее описанного класса, достаточно описать производный класс на основе предопределенного.

## Простота ввода/вывода в С ++

В пакет компилятора Micrisoft Visual C++ включена стандартная библиотека, содержащая функции, часто используемые в среде C++. В C++ по-прежнему можно пользоваться библиотекой стандартного ввода/вывода С, описанной в заголовочном файле stdio.h. В C++, однако, имеется другой заголовочный файл, называемый iostream.h, который реализует набор собственных функций ввода/вывода.

В языке C++ потоковый ввод/вывод описывается как набор классов в iostream.h. Эти классы используют перегруженные операции "**занести в**" и "**получить из**" **<<** и **>>**. Для того чтобы лучше понять, почему библиотека потоков в C++ более удобна, чем в С, давайте сначала посмотрим, как С управляет вводом и выводом.

Во-первых, напоминаем, что **в С отсутствуют встроенные операции ввода или вывода; функции, подобные printf(), являются частью стандартной библиотеки, но не самого языка. В C++ также нет встроенных средств ввода/вывода; это обеспечивает большую гибкость при разработке эффекгивного пользовательского интерфейса со структурами данных проектируемого приложения**.

Проблема с вводом и выводом в С заключается в реализации этих функций, плохо согласующихся между собой, если рассматривать возвращаемые значения и списки параметров. По этой причине программисты предпочитают использовать функции форматированного ввода/вывода printf(), scanf() и другие — в особенности, в тех случаях, когда обрабатываемыми объектами являются числа или другие несимвольные величины. Такие функции форматированного ввода/вывода удобны и, в своем большинстве, согласованы по интерфейсу; однако, они громоздки, поскольку должны работать с объектами различных форматов.

В языке C++ концепция класса обеспечивает модульное решение задачи манипуляции с данными. В стандартной библиотеке C++ имеются три класса ввода/вывода, являющиеся альтернативой функциям универсального ввода/вывода в С. Эти классы содержат описания для одной и той же пары операций, << и >>, которые оптимизируются для всех типов данных.

### Потоки cin, cout и сегг

Аналогами потоков **stdin**, **stdout** и **stderr**, имеющих прототипы в файле stdio.h, в C++ являются **cin**, **cout** и **сеrr**, описанные в iostream.h. Эти три потока открываются автоматически при запуске программы и становятся интерфейсом между программой и пользователем. Поток **cin** связан с клавиатурой терминала. Потоки **cout** и **сегг** связаны с видеодисплеем.

Еще в header <iostream> объявлены переменные std::cin и std::cerr для стандартного потока ввода и потока ошибок соответственно. Они являются объектами классов std::istream и std::ostream соотсветственно.

Аналогично тому, как stderr отличается от stdin, в языке C++ std::cerr отличается от std::cout отсутствием буферизации.

В классе std::istream есть перегруженный оператор >>. Можно считывать информацию из стандартного потока ввода (с клавиатуры).

### Операции выделения ( >> ) и вставки ( << )

Ввод и вывод в C++ значительно усовершенствован и упрощен, для чего использованы операции библиотеки потоков **>>** ("получить из" или выделение — extraction) и **<<** ("занести в" или вставка — insertion). **Одним из главных усовершенствований C++ по сравнению с С является возможность перегрузки операций, которая позволяет компилятору на основании информации о типах переменных выбирать одноименную функцию или операцию, которая должна выполняться.** Операции выделения и вставки являются хорошими примерами этой новой возможности C++. Каждая операция перегружена и поэтому может управлять всеми стандартными типами данных C++, включая классы. Два следующих фрагмента иллюстрируют легкость использования базовых операций ввода/вывода в C++. Сначала посмотрим на оператор вывода С, использующий printf():

printf("Integer value: %d, Float value: %f",lvalue,fvalue);

А это — эквивалентный оператор на C++:

cout << "Integer value: " **<< lvalue <<** ", Float value: " **<< fvalue;**

При внимательном анализе оператора C++ можно увидеть как перегруженная операция вставки используется для обработки трех различных типов данных: строки, целого числа и числа с плавающей точкой. Вам больше не потребуется искать символ %, необходимый в языке С в спецификациях формата для функций printf() и scanf(). В результате перегрузки операция вставки будет анализировать тип передаваемых ей данных и определять соответствующий формат.

Аналогичная ситуация существует для операции выделения, выполняющей ввод данных. Взгляните на следующий пример на С и на его эквивалент на C++:

/\* С-код \*/

scanf ("%d%f%c", &ivalue, &fvalue, &с) ;

// C++ -код

cin >> ivalue >> fvalue >> c;

Теперь не требуется использовать перед переменными операцию определения адреса (&). В C++ операция выделения берет на себя заботу по определению адреса переменной, требований к памяти и форматированию.

Увидев в двух примерах на C++ операции << и >>, вы можете удивиться тому, что выбраны именно эти операции. Чтобы запомнить, какая операция выполняет вывод, а какая — ввод, проще всего рассматривать две этих операции по отношению к файлам потокового ввода/вывода. Когда вам необходимо ввести информацию, вы выделяете ее (>>) из входного потока cin и заносите в некоторую переменную — например, ivalue. Для вывода информации вы берете копию информации из переменной ivalue и вставляете ее (<<) в выходной поток cout.

Непосредственным результатом перегрузки операций является то, что в C++ программа может расширять операции вставки и выделения. В следующем фрагменте программы показано, как можно перегрузить операцию вставки так, чтобы она печатала новый тип stclient:

ostream& operator **<< (ostream& osout, stclient staclient)**

{

**osout <<** " " **<< staclient.pszname;**

**osout <<** " " **<< staclient.pszaddress;**

**osout <<** " " **<< staclient.pszphone;**

}

Если предположить, что структурная переменная staclient была инициализирована, то для печати информации достаточно одного оператора:

cout << staclient;

И последней, но немаловажной, **особенностью операций вставки и выделения является их дополнительное преимущество — малый размер результирующего кода**. Универсальные функции ввода/вывода printf() и scanf() помимо своего кода включают в исполняемый модуль программы много зачастую ненужной информации. Если вы работаете в С только с целыми числами, то все равно включаете весь код для преобразования дополнительных стандартных типов данных. В C++, напротив, компилятор включает только те процедуры, которые действительно нужны.

Библиотека потоков C++ предоставляет несколько преимуществ в сравнении с функциями ввода-вывода библиотеки времени выполнения.

**Безопасность типов.** Давайте сравним вызов функций stdio с использованием стандартных потоков. Вызов stdio для чтения выглядит следующим образом:

int **i = 25;**

char **name[50] =** "Простая строка";

fprintf(stdout, "%d %s", i, name);

Он правильно напечатает:

25 Простая строка

Однако, если вы по невнимательности поменяете местами аргументы хотя fprintf(), ошибка обнаружится только во время исполнения программы. Может произойти все что угодно: от странного вывода до краха системы. Этого не может случиться в случае использования стандартных потоков:

cout << i << ' ' **<< name <<** '\n';

Tак как имеются перегруженные версии оператора сдвига operator << (), первый оператор всегда будет вызван. Функция cout << i вызывает operator << (int), a cout << name вызывает operator << (const char\*). Следовательно, использование стандартных потоков является безопасным по типам данных.

**Расширяемость для новых типов**. Другим преимуществом стандартных потоков является то, что определённые пользователем типы данных могут быть без труда в них встроены. Рассмотрим тип Pair, мы хотим напечатать:

struct **Pair {**int **x; string у;}**

Все, что нам нужно сделать, это -- перегрузить оператор operator << () для этого типа Pair, и мы сможем осуществлять вывод следующим образом:

Pair p(25, "December");

cout << р;

Соответствующий оператор operator << () может быть реализован так:

ostream<char>&operator<<(ostream<char>& о, const **Pair& p)**

{return **0 << р.х<<** ' ' **<< р.у;}**

**Простота и последовательность.** Библиотека потоков поддерживает единообразный интерфейс ввода-вывода, благодаря широкому использованию перегруженных функций и операторов. Это приводит к более простому и интуитивно понятному синтаксису.

**Операции помещения и извлечения из потока**

Вывод в поток выполняется с помощью оператора вставки (в поток), которым является перегруженный оператор сдвига влево <<. Левым его операндом является объект потока вывода. Правым его операндом может являться любая переменная, для которой определен вывод в поток (т.е. переменная любого встроенного типа или любого определенного пользователем типа, для которого она перегружена). Например,

cout << "Hello!\n";

приводит к выводу в предопределенный поток cout строки "Hello! ".

Оператор << возвращает ссылку на объект ostream, для которого он вызван. Это позволяет строить цепочки вызовов оператора вставки в поток, которые выполняются слева направо:

int **i = 5;**

double **d = 2.08;**

cout << "i = " **<< i <<** " d = " **<< d <<** '\n';

Эта инструкция приведет к выводу на экран следующей строки:

i = 5, d = 2.08

**Оператор вставки в поток поддерживает следующие встроенные типы данных: bool, char, short, int. long, char\* (рассматриваемый как строка), float, double, long double, и void\***:

ostream type& operator<< (bool **n);**

ostream\_type& operator<< (short **n);**

ostream type& operator<<(unsignedshort **n);**

ostream type& operator<< (int **n);**

ostream\_type& operator<< (unsignedint **n);**

ostream\_type& operator<< (long **n);**

ostream\_type& operator<< (unsignedlong **n);**

ostream\_type& operator<< (float **f);**

ostream\_type& operator<< (double **f);**

ostream\_type& operator<< (longdouble **f);**

ostream\_type& operator<< (constvoid **\*p);**

Целочисленные типы форматируются в соответствии с правилами, принятыми по умолчанию, для функции printf () (если они не изменены путем установки различных флагов форматирования). Например, следующие две инструкции дают одинаковый результат:

int **i;**

long **l;**

cout << i << " ' << l;

printf("%d%ld",i, l);

Тип void\* используется для отображения адреса указателя:

int **i;**

//Отображить адрес указателя

//в 16-ричной форме.

cout << &i;

Для ввода информации из потока используется оператор извлечения, который является перегруженный оператор сдвига вправо >>. Левым операндом оператора >> является объект класса istream. Это позволяет строить цепочки инструкций извлечения из потока, выполняемых слева направо. Правым операндом может быть любой тип данных, для которого определен поток ввода.

istream\_type& operator>>(bool& n);

istream\_tvpe& operator>>(short& n);

istream\_type& operator>>(unsignedshort& n);

istream\_type& operator>> (int& n);

istream\_type& operator>>(unsignedint& n);

istream\_type& operator>>(long& n);

istream\_type& operator>>(unsignedlong& n);

istream\_type& operator>>(float& f);

istream\_type& operator>>(double& f);

istream\_type& operator>>(longdouble& f);

istream\_type& operator>>(void\*& p);

По умолчанию оператор >> пропускает символы-заполнители, затем считывает символы, соответствующие типу заданной переменной. Пропуск ведущих символов-заполнителей устанавливается специально для этого предназначенным флагом форматирования. Рассмотрим следующий пример:

int **i;**

double **d;**

cin >> i >> d;

Последняя инструкция приводит к тому, что программа пропускает ведущие символы-заполнители и считывает целое число i. Затем она игнорирует любые символы-заполнители, следующие за целым числом, и считывает переменную с плавающей точкой d.

Для переменной типа char\* (рассматриваемого как строка) оператор >> пропускает символы-заполнители и сохраняет следующие за ними символы, пока не появится следующий символ-заполнитель. Затем в указанную переменную добавляется нуль-символ.

В следующей программе показано использование операции ввода (или выделения) >> для считывания различных типов данных:

//04.12INSTR1

// Программа на C++, демонстрирующая использование операции

// выделения >> для ввода типов char, int, float, double и строк

#include "stdafx.h"

#include <iostream>

using namespace std;

#define INUMCHARS 45

#define INULL\_CHAR 1

void main(void)

{

char canswer;

int ivalue;

float fvalue;

double dvalue;

char pszname[INUMCHARS+INULL\_CHAR];

// Эта программа позволяет вводить различные типы данных.

// Вы хотите ее испробовать ? // Напечатайте y — да или n — нет:

cout << "This program allows you to enter various data types." << "\n";

cout << "Would you like to try it?" << "\n\n";

cout << "Please type a y for YES and n for NO: ";

cin >> canswer;

if(canswer == 'y')

{

cout << "\n" << "Enter an integer value: "; // Введите целое число

cin >> ivalue;

cout << "ivalue=" << ivalue << "\n\n";

cout << "Enter a float value: "; // Введите вещественное число

cin >> fvalue;

cout << "fvalue=" << fvalue << "\n\n";

cout << "Enter a double value: "; // Введите число двойной точности

cin >> dvalue;

cout << "dvalue=" << dvalue << "\n\n";

cout << "Enter your first name: "; // Введите ваше имя

cin >> pszname;

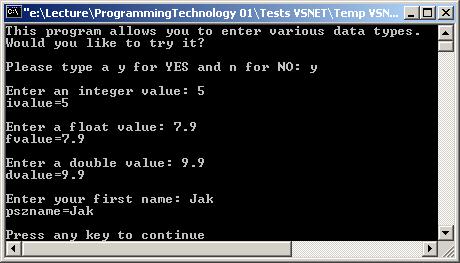
cout << "pszname=" << pszname;

cout << "\n\n";

}

}

Результаты работы программы:



В приведенном примере для вывода строк-запросов используется простейшая форма операции вставки <<. Обратите внимание на то, что в программе используются четыре различных типа данных, но все операторы ввода cin >> выглядят одинаково, отличаясь только именем переменной. Те из вас, кто быстро печатает, но устал от поиска редко используемых символов %, " и & (необходимых для scanf()), могут дать отдых пальцам и глазам. Операция выделения в C++ значительно упрощает код и уменьшает вероятность ошибок.

Поскольку C++ быстро эволюционирует, вам необходимо с осторожностью использовать коды С или C++, встречающиеся в устаревшей литературе. Если вы, к примеру, запустите приведенную программу с компилятором C++ Release 1.2, то результат работы программы будет выглядеть несколько некорректно и будет оличаться от приведенного.

Это объясняется тем, что входной поток в C++ Release 1.2 обрабатывает символ перевода строки, введенный после печати буквы "y”. Операция вставки >> считывает, но не отбрасывает символ перевода строки. В следующей программе эта проблема решена, для чего добавлен еще один оператор ввода:

// 04.12INSRT2

// Программа на C++, демонстрирующая использование операции

// выделения >> для ввода типов char, int, float, double и строк

#include "stdafx.h"

#include <iostream>

using namespace std;

#define INUMCHARS 45

#define INULL\_CHAR 1

void main(void)

{

char canswer, c0x0Anewline;

int ivalue;

float fvalue;

double dvalue;

char pszname[INUMCHARS+INULL\_CHAR];

cout << "This program allows you to enter various data types." << "\n";

cout << "Would you like to try it?" << "\n\n";

cout << "Please type a y for YES and n for NO: ";

cin >> canswer;

cin.get(c0x0Anewline);

if(canswer == 'y')

{

cout << "\n" << "Enter an integer value: ";

cin >> ivalue;

cout << "ivalue=" << ivalue << "\n\n";

cout << "Enter a float value: ";

cin >> fvalue;

cout << "fvalue=" << fvalue << "\n\n";

cout << "Enter a double value: ";

cin >> dvalue;

cout << "dvalue=" << dvalue << "\n\n";

cout << "Enter your first name: ";

cin >> pszname;

cout << "pszname=" << pszname;

cout << "\n\n";

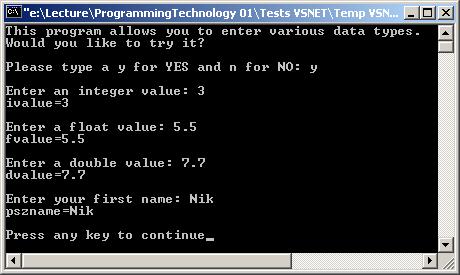
}

}

Вы заметили изменения? После считывания canswer программа выполняет оператор

cin.get(c0x0Anewline);

Он обрабатывает символ перевода строки, и после запуска программа выдает следующий результат:



Оба алгоритма работают правильно с момента появления версии C++ Release 2.0. Стоит, однако, еще раз напомнить, что необходимо с осторожностью подходить к старым листингам программ. Если использовать старые версии программ на С и C++ с существующими компиляторами, то можно потратить много часов, пытаясь понять, почему ввод/вывод выполняется не так, как ожидалось.

В следующем примере демонстрируется использование различных форм операции вывода или вставки <<:

// 04.12EXTRCT

// Программа на C++, демонстрирующая использование операции

// вставки << для вывода типов char, integer, float, double и строк

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#include <iostream>

using namespace std;

void main(void)

{

char c = 'A';

int ivalue = 10;

float fvalue = (float)45.67;

double dvalue = 2.3e32;

char fact[] = "For all have ..."; // Сколько не имей …

cout << "Once upon a time there were "; // Жили однажды

cout << ivalue << " people." << endl; // ... человек

cout << "Some of them earrend " << fvalue; // Некоторые зарабатывали

cout << " dollars per hour." << "\n"; //... долларов в час

cout << "While others earrend " << dvalue << " per year!"; // Другие зарабатывали ... в год!

cout << "\n\n" << "But you know what they say: "; // И знаете, что они говорили:

cout << "\"" << fact << "\"" << "\n\n";

cout << "So, none of them get an "; // Поэтому никто из них не получает

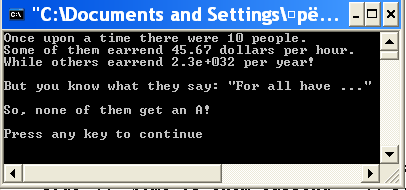
cout << c; // высший балл

cout << "!";

cout << "\n\n";

}

Результат работы программы следующий:



Первое, что **нужно отметить при сравнении исходного текста программы на C++ и полученного результата, это то, что операция вставки << не генерирует автоматически символ перевода строки. При необходимости это можно делать непосредственно, включая в программу символ \n или endl**.

Константа **endl** очень полезна при выводе данных в интерактивном режиме, поскольку помимо вывода в поток символа перевода строки выполняется очистка выходного буфера. Для этого можно также использовать команду **flush**; однако, при этом не происходит перевода строки. Заметим также, что символ перевода строки может включаться после соответствующей операции вставки << и как часть литеральной строки, что видно в программе по второму и четвертому операторам <<. Нужно отметить, что, хотя операция вставки и удобна для форматирования целых и вещественных чисел, она не очень хорошо выводит числа двойной точности. Еще одна особенность этой операции проявляется в версии C++ Release 1.2 при выводе символов. Взгляните на следующий оператор:

cout << с;

В версии Release 1.2 вы получили бы следующий результат:

So, none of them get an 65!

Это вызвано тем, что символ преобразуется в эквивалентный ASCII-код. Для вывода символьной информации в версии Release 1.2 нужно использовать функцию put(). Поэтому данный оператор следует записать так:

cout.put(с);

Попробуйте запустить следующий пример:

// 04.12STRING

// Программа на C++, показывающая, что происходит при использовании

// операции выделения >> со строковыми данными

#include "stdafx.h"

#include <iostream>

using namespace std;

#define INUMCHARS 45

#define INUL\_CHARACTER 1

void main(void)

{

char pszname [INUMCHARS+INUL\_CHARACTER];

// Введите, пожалуйста, ваше имя и фамилию

cout << "Please enter your first and second name: ";

cin >> pszname;

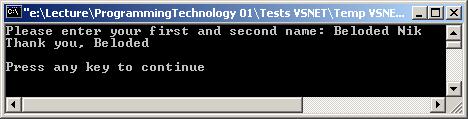
// Спасибо

cout << "Thank you, " << pszname;

cout << "\n\n";

}

Результат выполнения программы следующий:



Это — еще одна особенность, о которой нужно помнить при вводе строк. Операция выделения >> прекращает считывание информации, когда встречает разделительный символ, который может представлять собой символ пробела, табуляции или перевода строки. Поэтому при печати pszname выводится только первое имя. Эту проблему можно решить, если переписать программу, используя функцию cin.get():

// 04.12CINGET

// Программа на C++, показывающая, что происходит при использовании

// операции выделения >> вместе с функцией cin.get() при обработке полной строки

#include "stdafx.h"

#include <iostream>

using namespace std;

#define INUMCHARS 45

#define INUL\_CHARACTER 1

void main(void)

{

char pszname [INUMCHARS+INUL\_CHARACTER];

cout << "Please enter your first and second name: ";

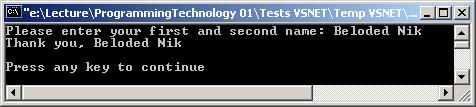
cin.get(pszname, INUMCHARS);

cout << "Thank you, " << pszname;

cout << "\n\n";

}

Результат выполнения этой программы следующий:



Функция cin.get() имеет два дополнительных параметра. В приведенном примере использовался только один из них — количество вводимых символов. Функция cin.get() считывает всю информацию, включая пробельные символы, до тех пор, пока не будет введено указанное максимальное число символов или считывание будет продолжаться до следующего символа перевода строки; это зависит от того, какое условие выполнится раньше. Третий, необязательный и не показанный здесь параметр задает символ окончания считывания. Например, следующий оператор будет считывать в переменную pszname INUMCHARS символов или все символы, предшествующие символу \* или символу перевода строки, в зависимости от того, что раньше выполнится:

cin.get(pszname, INUMCHARS, '\*');

## Опции ввода/вывода в С/С++.

Ранее были представлены концепции и синтаксис объектно-ориентированных классов, конструкторов, деструкторов, методов классов и операций. Теперь можно переходить к более глубокому анализу ввода/вывода в С++.

Так же как и в С, в языке С++ отсутствуют какие-либо встроенные процедуры ввода/ввода. Вместо них все компиляторы С++ поставляются с объектно-ориентированными классами iostream. Эти стандартные объекты классов ввода/вывода синтаксически согласованы, поскольку разрабатывались авторами языка С++. Если нужно написать некоторое приложение С++, переносимое на другие компиляторы С++, то можно использовать классы iostream. В компиляторе Visual С/С++ имеются следующие средства (5 вариантов) для осуществления ввода/вывода в С/С++.

Библиотека С небуферизированного ввода/вывода — Компилятор С обеспечивает небуферизированный ввод/вывод при помощи функций \_read() и \_write(). Эти функции весьма популярны у программистов на С по причине своей эффективности и легкости их адаптации.

Буферизированный ввод/вывод ANSI С — В С имеются также буферизированные функции fread() и fwrite(). Они описаны в библиотеке stdio.h и выполняют собственную буферизацию перед непосредственным обращением к базовым процедурам ввода/вывода.

Библиотека С ввода/вывода на консоль и в порты — В С имеются дополнительные процедуры ввода/вывода, не имеющие аналогов в С++: \_getch(), \_ungetch() и \_kbhit(). Во всех приложениях, не относящихся к Windows, можно использовать эти функции, обеспечивающие прямой доступ к аппаратному обеспечению.

Библиотека Microsoft классов iostream — Библиотека классов iostream предоставляет программам на С++ возможности объектно-ориентированного ввода/вывода. Ее можно использовать вместо таких функций, как scanf(), printf(), fscanf() и fprintf(). Однако, эти классы iostream не являются обязательными для программ на С++ и многие символьные объекты, например cin, cout, cerr и clog, не совместимы с графическим интерфейсом пользователя Windows.

Библиотека Microsoft Foundation Class — Класс Microsoft CFile, находящийся в библиотеке МFС, обеспечивает приложения С++ и, в особенности, Windows - приложения средствами объектного дискового ввода/вывода. Использование этих библиотечных процедур гарантирует переносимость приложения и легкость сопровождения.

**Замечание**

Общий совет заключается в том, что не надо смешивать техники для работы с файлами.

Например, не надо в одной и той же программе использовать функции из стандартной библиотеки C (fread/fwrite) и классы-потоки из языка C++ (istream/ostream).

## Операции и методы классов

Для того чтобы принимать аргументы любых встроенных типов данных, включая char \*, операции выделения >> и вставки << были модифицированы. Их также можно расширить, и они будут принимать в качестве параметров классы.

Первый случай несовместимости при преобразовании программ на C++, использующих старые библиотеки ввода/вывода, может возникнуть при модификации устаревшего расширения cout << form. В новой версии каждый объект класса iostream имеет переменные, управляющие операциями форматирования: например, преобразованием системы счисления для целых чисел или точностью числа с плавающей точкой.

Флагами статуса формата можно манипулировать при помощи функций **setf()** и **unsetf()**. Метод класса setf() используется для установки определенного флага статуса формата. Имеются два перегруженных варианта:

setf(long);

setf(long, long);

Первым аргументом может быть либо битовый флаг формата, либо битовое поле формата. В таблице перечислены флаги формата, которые можно использовать в варианте setf(long) (когда имеется только флаг формата).

|  |  |
| --- | --- |
| **Флаг** | **Значение** |
| ios:: showbase | Отображает числовые константы в формате, который может читать компилятор C++ |
| ios:: showpoint | Отображает числа float с точкой и нулями в конце |
| ios:: dec | Форматирует числа в десятичной системе счисления (система счисления по умолчанию) |
| ios:: oct | Форматирует числа в восьмеричной системе счисления |
| ios:: hex | Форматирует числа в шестнадцатеричной системе счисления |
| ios:: fixed | Показывает числа float в фиксированном формате |
| ios:: scientific | Показывает числа float в экспоненциальном формате |
| ios:: showpos | Отображает знак "плюс" (+) перед положительными числами |
| ios:: skipws | Пропускает пробельные символы на входе |
| ios:: left | Выравнивает слева все значения (дополняя справа указанным заполняющим символом) |
| ios: right | Выравнивает справа все значения (дополняя слева указанным заполняющим символом, опция по умолчанию) |
| ios:: internal | Добавляет заполняющие символы после любого знака или указателя системы счисления до начала числа |
| ios:: uppercase | Отображает заглавные "А" — "F" для шестнадцатеричных чисел; и "Е" для экспоненциальных значений |
| ios:: unitbuf | Заставляет ostream::osfx очищать поток после каждой операции вставки (по умолчанию сегг буферизируется) |
| ios::stdio | Заставляет ostream::osfx очищать stdout и stderr после каждой вставки |

В следующей таблице перечислены некоторые битовые поля формата, которые можно использовать в варианте setf(long, Iong) (когда имеется флаг формата и битовое поле формата):

|  |  |  |
| --- | --- | --- |
| **Битовое поле** | **Значение** | **Флаги** |
| ios:: basefield | Целые числа | ios:: hex, ios:: oct, ios:: dec |
| ios:: floatfield | Числа с плавающей точкой | ios:: fixed, ios:: scientific |

Имеются некоторые предопределенные установки по умолчанию. Например, целые числа пишутся и читаются как десятичные. Можно изменить основание системы счисления на восьмеричное, шестнадцатеричное или же обратно на десятичное. По умолчанию число с плавающей точкой выводится с точностью шесть цифр. Точность можно изменить при помощи метода определения точности. Эти новые методы классов используются в следующей программе на C++:

// 04.12ADVIO

// Программа на C++, демонстрирующая использование методов

// класса для преобразования и форматирования, имеющихся в версиях выше

// Release 2.0. Также в программе показано, как преобразовать все

// устаревшие операторы form версии Release 1.2

//

//#include "stdafx.h"

#include <stdio.h>

#include <strstream>

#include <iostream>

//using namespace std;

//

#include <string.h>

#include <strstrea.h>

#define INULL\_TERMINATOR 1

void row (void);

int main()

{

char c = 'A',

psz1[] = "In making a living today many no ",

psz2[] = "longer leave any room for life.";

int iln = 0,

ivalue = 1234;

double dPi = 3.14159265;

// новые объявления, необходимые для Release 2.0

char psz\_padstring5[5+INULL\_TERMINATOR],

psz\_padstring38[38+INULL\_TERMINATOR];

/////////////////////////////// преобразования ˙ ///////////////////////////////

// печать символа с

// R1.2 cout << form("\n[%2d] %c", ++ln, c) ;

// обратите внимание на то, что для вывода char операция << перегружена

row(); // [ 1]

cout << c;

// печать ASCII-кода символа с

// R1.2 form("\n[%2d] %d",++ln, c);

row(); // [ 2]

cout << (int)c;

// печать символа с ASCII-кодом 90

// R1.2 form("\n[%2d] %c",++ln, 90);

row(); // [ 3]

cout << (char) 90;

// печать ivalue в виде восьмеричного числа

// R1.2 form("\n[%2d] %o",++ln, ivalue);

row(); // [ 4]

cout << oct << ivalue;

// печать шестнадцатеричного числа строчными буквами

// R1.2 form("\n[%2d] %õ",++ln, ivalue);

row(); // [ 5]

cout << hex << ivalue;

// печать шестнадцатеричного числа заглавными буквами

// R1.2 form("\n[%2d] %X",++ln, ivalue);

row(); // [ 6]

cout.setf (ios:: uppercase);

cout << hex << ivalue;

cout.unsetf(ios:: uppercase); // выключить заглавные

cout << dec; // вернуться к десятичному счислению

// опции преобразования и форматирования

// минимальная ширина 1

// R1.2 form("\n[%2d] %c",++ln, c);

row(); // [ 7]

cout << c;

// минимальная ширина 5, выравнивание вправо

// R1.2 form("\n[%2d] %5c",++ln,c) ;

row(); // [ 8]

ostrstream (psz\_padstring5, sizeof (psz\_padstring5)) << " " << c << ends;

cout << psz\_padstring5;

// минимальная ширина 5, выравнивание влево

// R1.2 form("\n[%2d] %-5c",++ln, c) ;

row(); // [ 9]

ostrstream(psz\_padstring5, sizeof(psz\_padstring5)) << c << " " << ends;

cout << psz\_padstring5;

// ширина 33 автоматически

// R1.2 form("\n[%d] %s",++ln, pszl);

row(); // [10]

cout << psz1;

// ширина 31 автоматически

// R1.2 form("\n[%d] %s",++ln, psz2);

row(); // [11]

cout << psz2;

// минимальная ширина 5 отменяется, ширина определяется автоматически

// R1.2 form("\n[%d] %5s",++ln, pszl);

// заметьте, что ширину 5 отменить не удалось!

row(); // [12]

cout.write(psz1,5);

// минимальная ширина 38, выравнивание вправо

// R1.2 form("\n[%d] %38s",++ln, psz1);

// обратите внимание на то, что при ширине 38 в конце появляется мусор

row(); // [13]

cout.write(psz1, 38);

// ниже следует правильный вариант

cout << "\n\nCorrected approach:\n";

ostrstream(psz\_padstring38,sizeof(psz\_padstring38)) << " " << psz1 << ends;

row(); // [14]

cout << psz\_padstring38;

// минимальная ширина 38, выравнивание влево

// R1.2 form("\n[%d] %-38s",++ln, psz2);

ostrstream(psz\_padstring38, sizeof(psz\_padstring38)) << psz2 << " " << ends;

row(); // [15]

cout << psz\_padstring38;

// ширина lvalue по умолчанию

// R1.2 form("\n[%d] %d",++ln, ivalue);

row(); // [16]

cout << ivalue;

// печать lvalue со знаком (+)

// R1.2 form("\n[%d] %+d",++ln, ivalue);

row(); // [17]

cout.setf(ios:: showpos); // для номера строки (+) не нужен

cout << ivalue;

cout.unsetf(ios:: showpos);

// минимальная ширина З отменяется, ширина определяется автоматически

// R1.2 form("\n[%d] %3d",++ln, lvalue);

row(); // [18]

cout.width(3); // номер строки не нужно дополнять до 3

cout << ivalue;

// минимальная ширина 10, выравнивание вправо

// R1.2 form("\n[%d] %10d",++ln, ivalue);

row(); // [19]

cout.width(10); // влияет только на первое выводимое значение

cout << ivalue;

// минимальная ширина 10, выравнивание влево

// R1.2 form("\n[%d] %-d",++ln, ivalue);

row(); // [20]

cout.width(10);

cout.setf(ios:: left);

cout << ivalue;

cout.unsetf(ios:: left);

// выравнивание вправо с начальными нулями

// R1.2 form("\n[%d] %010d",++ln, ivalue);

row(); // [21]

cout.width(10);

cout.fill('0');

cout << ivalue;

cout.fill(' ');

// число цифр по умолчанию

// R1.2 form("\n[%d] %f",++ln, dPi) ;

row(); // [22]

cout << dPi;

// минимальная ширина 20, выравнивание вправо

// R1.2 form("\n[%d] %20f",++ln, dPi) ;

row(); // [23]

cout.width(20);

cout << dPi;

// выравнивание вправо с ведущими нулями

// R1.2 form("\n[%d] %020f",++ln, dPi) ;

row(); // [24]

cout.width(20);

cout.fill('0');

cout << dPi;

cout.fill(' ');

// минимальная ширина 20, выравнивание влево

//R1.2 form("\n[%d] %-20f",++ln, dPi);

row(); // [25]

cout.width(20);

cout.setf(ios:: left);

cout << dPi;

// выравнивание влево с нулями в конце

// R1.2 form("\n[%d] %-020f",++ln, dPi);

row(); // [26]

cout.width(20);

cout.fill('0');

cout << dPi;

cout.unsetf(ios:: left);

cout.fill(' ');

// дополнительная точность при форматировании

// минимальная ширина 19, печать всех 19

// R1.2 form("\n[%d] %19.19s",++ln, psz1);

row(); // [27]

cout << psz1;

// печать первых 2 символов

// R1.2 form("\n[%d] %.2s",++ln, psz1);

row(); // [28]

cout.write(psz1,2);

// печать 2 символов, выравнивание вправо

// R1.2 form("\n[%d] %19.2s",++ln, psz1);

row(); // [29]

cout << " ";

cout.write(psz1,2);

// печать 2 символов, выравнивание влево

// R1.2 form("\n[%d] %-19.2s",++ln, psz1);

row(); // [30]

cout.write(psz1,2);

// использование аргументов печати

// R1.2 form("\n[%d] %\*.\*s", ++ln, 19, 6, psz1);

row(); // [31]

cout << " ";

cout.write(psz1, 6);

// ширина 10, 8 справа от '.'

// R1.2 form("\n[%d] %10.8f",++ln, dPi);

row(); // [32]

cout.precision(9);

cout << dPi;

// ширина 20, точность 2 с выравниванием вправо

// R1.2 form("\n[%d] %20.2f",++ln, dPi) ;

row(); // [33]

cout.width(20) ;

cout.precision(2);

cout << dPi;

//4 десятичных позиции, выравнивание влево

// R1.2 form("\n[%d] %-20.4f",++ln, dPi);

row(); // [34]

cout.precision(4);

cout << dPi;

//4 десятичных позиции, выравнивание вправо

// R1.2 form("\n[%d] %20.4f",++ln, dPi);

row(); // [35]

cout. width (20);

cout << dPi;

// ширина 20, экспоненциальный формат

// R1.2 form("\n[%d] %20.2e",++ln, dPi);

row(); // [36]

cout.setf (ios:: scientific);

cout.width(20);

cout << dPi;

cout.unsetf (ios:: scientific);

cout << "\n";

return (0);

}

void row (void)

{

static int ln=0;

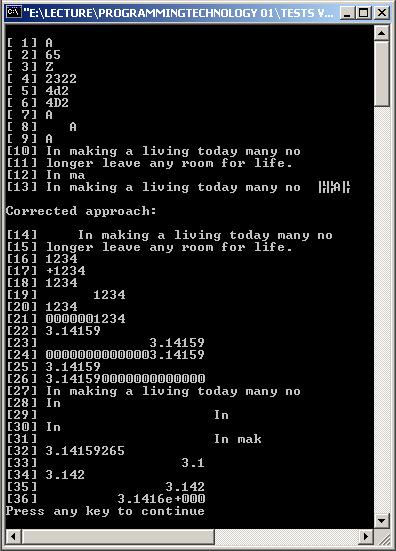
cout << "\n[";

cout.width(2);

cout << ++ln << "] ";

}

Результат работы программы можно использовать при написании ваших собственных операторов сложного вывода:



В следующих разделах рассматриваются те операторы вывода, использованные в приведенной программе, которые требуют особого обсуждения. Одно необходимое замечание: файл strstream.h автоматически включает iostream.h. Файл strstream.h требуется для форматирования выводимых строк. Если в вашей программе нужен вывод только числовых данных или простой вывод символов и строк, то необходимо включить только файл iostream.h.

### Вывод символов в C++

В новой библиотеке ввода/вывода (версии Release 2.0 и выше) для обработки символьных данных операция вставки « была перегружена. В ранних версиях следующий оператор напечатал бы ASCII-код символа с:

cout << с;

В нынешней библиотеке печатается сама символьная переменная. Если же нужно получить ASCII-код, то используется такой оператор:

cout << (int)c;

### Преобразование системы счисления в C++

Есть два способа для вывода значения в разных системах счисления:

cout << hex << ivalue;

и другой способ

cout.setf(ios::hex, ios::basefield);

cout << ivalue;

В обоих случаях с момента выполнения оператора и далее устанавливается новая система счисления (хотя это нужно не всегда). Каждое выводимое значение будет форматироваться как шестнадцатеричное. Переход к некоторому другому основанию выполняется при помощи функции unsetf():

cout.unsetf(ios::hex, ios::basefield);

Если вам нужно выводить шестнадцатеричные числа заглавными буквами, то используйте следующий оператор:

cout.setf(ios::uppercase);

Когда необходимость в этом пропадает, опцию нужно выключить:

cout.unsetf(ios::uppercase);

### Форматирование строк в C++

Распечатать строку в C++ несложно. Однако, форматирование строк изменилось, так как расширение cout << form, существовавшее в версии Release 1.2, теперь отсутствует. Для форматирования строки можно объявить массив символов, а затем выбрать необходимый формат вывода при распечатке буфера строки:

pszpadstring38[38 + INULL\_TERMINATOR];

…

ostrstream(pszpadstring38,sizeof(pszpadstring38)) << "->" **<< psz1;**

Метод класса ostrstream() является частью файла strstream.h и имеет три параметра: указатель на массив символов, размер массива и выводимая информация. Для выравнивания вправо этот оператор в начало строки psz1 добавляет пробелы. Часть строки можно вывести при помощи метода write объекта cout:

cout.write(psz1,5);

Этот оператор выведет первые пять символов строки psz1.

### Форматирование чисел в C++

Возможен ли форматированный вывод, которым мы пользовались в языке C фунцией printf()? Например, как вывести ту же переменную N в 16-ой записи? В языке C++ форматированный вывод возможен при помощи вывода на экран специальной управляющей команды:

std::cout << std::ios::hex << N;

В точности то же самое выведет команда printf("%x", N);

Чтобы не писать перед кажой переменной ее формат, можно использовать функцию:

std::cout.setf(std::ios::hex);

Она установит формат вывода в стандартный поток вывода на экран. Этот подход настолько же мощный, как и использование форматной печати с помощью printf.

Можно без затруднений форматировать числовые данные: выравнивать вправо или влево, менять точность и формат (с плавающей точкой или экспоненциальный), добавлять в начало или в конец заполняющие символы, управлять знаком. Имеются некоторые установки по умолчанию. Например: по умолчанию выполняется выравнивание вправо, и точность чисел с плавающей точкой равна шести. В следующем фрагменте программы выводится значение dPi, выравненное влево в поле шириной 20 символов и с нулями в конце:

cout.width(20);

cout.setf(ios:: left);

cout.fill ('0');

cout << dPi;

Если добавить следующий оператор, то переменная dPi будет выводиться с точностью 2:

cout.precision(2);

Многие флаги вывода (например, выравнивание влево, выбор заглавных букв для шестнадцатеричных чисел, изменение системы счисления и многие другие) необходимо сбрасывать после того, как в них отпадает необходимость. Следующий оператор выключает выравнивание влево:

cout.unsetf(ios::left);

Для выбора экспоненциального формата достаточно установить соответствующий битовый флаг:

cout.setf (ios:: scientific);

Если установить флаг showpos, то можно печатать значения со знаком плюс (+) перед числом:

cout.setf (ios:: showpos);

Среди функций последней версии библиотеки ввода/вывода существует множество мелких деталей, которые поначалу могут вносить некоторую путаницу. Это объясняется тем, что некоторые операции, будучи однажды выполненными, изменяют состояние до момента выключения, а действие других операций распространяется только на следующий оператор вывода. Например: изменение ширины вывода cout.width(20) действует только на следующее печатаемое значение. Вот почему функция row() должна была постоянно менять ширину для печати форматированных номеров строк, занимающих две позиции — например, [ 1]. Другие операции форматирования, подобные изменению основания системы счисления, выбору заглавных букв, точности и формата (с плавающей точкой или экспоненциального), остаются активными до тех пор, пока их не выключат явно.

### Файловый ввод и вывод в C++

Для осуществления операций с файлами библиотека ввода-вывода предусматривает три класса: ifstream, ofstream и fstream. Эти классы являются производными, соответственно, от классов istream, ostream и iostream. Поскольку эти последние классы, в свою очередь, являются производными от класса ios классы файловых потоков наследуют все функциональные возможности своих родителей (перегруженные операции << и >> для встроенных типов, функции и флаги форматирования, манипуляторы). Для реализации файлового ввода-вывода нужно включить в программу заголовочный файл <fstream>.

Существуют небольшие отличия между использованием предопределённых и файловых потоков. Файловый поток должен быть связан с файлом прежде, чем его можно будет использовать. С другой стороны, предопределенные потоки могут использоваться сразу после запуска программы, даже в конструкторах статических классов, которые выполняются даже раньше вызова функции main(). Вы можете позиционировать файловый поток в произвольную позицию в файле, в то время как для предопределённых потоков это обычно не имеет смысла.

Для создания файлового потока эти классы предусматривают следующие формы конструктора:

**создать поток, не связывая его с файлом**:

explicit **ifstream();**

explicit **ofstream();**

explicit **fstream();**

**создать поток, открыть файл и связать поток с файлом**:

explicit **ifstream(**constchar **\*name,**

**os::openmode mode = ios::in);**

explicit **ofstream(**constchar **\* name,**

**ios::openmode mode = ios::out |**

**ios::trunc);**

explicit **fstream(**constchar **\* name,**

**ios::openmode mode = ios::in |**

**ios::out);**

Чтобы открыть файл для ввода или вывода, можно использовать вторую форму нужного конструктора

fstream fs("FileName.dat");

или вначале создать поток с помощью первой формы конструктора, а затем открыть файл и связать поток с открытым файлом, вызвав функцию-член open (). Эта функция определена в каждом из классов потокового ввода-вывода и имеет следующие прототипы:

void **ifstream::open(**constchar **\*name,**

**ios::openmode mode = ios::in);**

void **ofstream::open(**constchar **\* name,**

**ios::openmode mode = ios::out |**

**ios::trunc);**

void **fstream::open(**constchar **\* name,**

**ios::openmode mode = ios::in |**

**ios::out);**

Здесь name - имя файла, mode - режим открытия файла.

Параметр mode является перечислением и может принимать значения, приведенные в таблице.

**Режимы открытия и их назначение**

|  |  |
| --- | --- |
| **Режим открытия** | **Назначение** |
| ios::in | Открыть файл для чтения |
| ios::out | Открыть файл для записи |
| ios::ate | Начало вывода устанавливается в конец файла |
| ios::app | Окрьть файл для добавления в конец |
| ios:: trunc | Усечь файл, т.е. удалить его содержимое |
| ios::binary | Двоичный режим операций |

Режимы открытия файла представляют собой битовые маски, поэтому вы можете задавать два или более режима, объединяя их операцией ИЛИ. В следующем фрагменте кода файл открывается для вывода с помощью функции open ():

ofstream ofs;

ofs.open("FileName.dat");

Обратите внимание, что по умолчанию режим открытия файла соответствует типу файлового потока. У потока ввода или вывода флаг режима всегда установлен неявно. Например, для потока вывода в режиме добавления файла можно вместо инструкции

ofstream ofs("FName.txt", ios:: out | ios::app);

написать следующую:

ofstream ofs("FName.txt", ios::app);

Между режимами открытия файла ios::app и ios::ate имеется небольшая разница.

Если файл открывается в режиме добавления весь вывод в файл будет осуществляться в позицию, начинающуюся с текущего конца файла, безотносительно к операциям позиционирования в файле. В режиме открытия ios:: ate (от англ. "at end" ) вы можете изменить позицию вывода в файл и осуществлять запись начиная с нее. Для потоков вывода режим открытия эквивалентен ios :: out | ios :: trunc, то есть вы можете опустить режим усечения файла. Однако для потоков ввода-вывода его нужно указывать явно. Файлы, которые открываются для вывода, создаются, если они еще не существуют.

Если открытие файла завершилось неудачей, объект, соответствующий потоку в булевом выражении, будет возвращать false:

if **(!ofs){**

**cout <<** "Файл не открыт\n";

}

Проверить успешность открытия файла можно также с помощью функции-члена is\_open(), имеющей следующий прототип:

bool **is\_open()** const;

Функция возвращает true, если поток удалось связать с открытым файлом. Например,

if **(!osf.is\_open()){**

**cout <<** "Файл не открыт\n";

}

Если при открытии файла не указан режим **ios::binary**, файл открывается в текстовом режиме и после того, как файл успешно открыт, для выполнения операций ввода-вывода можно использовать операторы извлечения и вставки в поток. Можно даже использовать функции ввода-вывода, принятые в языке С, такие, как **fprintf()** и **fscanf()**.Для проверки, достигнут ли конец файла, можно использовать функцию **eof()** класса **ios**, имеющую следующий прототип:

bool **eof()** const;

Завершив операцию ввода-вывода, необходимо закрыть файл, вызвав функцию-член **close()**:

ofs.close();

Функция close() не имеет параметров и возвращаемого значения:

void.close();

**Закрытие файла происходит автоматически при выходе объекта потока из области видимости, когда вызывается деструктор потока.**

Рассмотрим пример, демонстрирующий файловый ввод-вывод с использованием потоков:

//04.01

#include <fstream>

#include <iostream>

using namespace std;

int main()

{

setlocale(LC\_ALL, "Rus");

int n = 50;

//Открываем файл для вывода

ofstream ofs("Test.txt");

if (!ofs){

cout << "Файл не открыт.\n";

return 1;

}

ofs << "Hello!\n" << n;

//Вывод другой информации в файл

//…

//Закрываем файл

ofs.close();

//открываем тот же файл для ввода

ifstream file("Test.txt");

if (!file){

cout << "Файл не открыт.\n";

return 2;

}

char str[80];

file >> str >> n;

cout << str << " " << n << endl;

//Закрываем файл

file.close();

getchar(); getchar();

return 0;

}



Следует иметь в виду, что стандартная библиотека ввода-вывода отличается от традиционной. Этот факт нужно учитывать при переносе старых программ в современные системы программирования. Во-первых, в традиционной библиотеке функция open() имела третий параметр, задававший режим защиты файла. Во-вторых, конструктор потока fstream и функция open() для него не предусматривали установки по умолчанию режимов открытия ios:: in | ios :: out. В-третьих, стандартная библиотека не поддерживает режимы открытия ios:: noncreate и ios:: noreplace, которые были в традиционной.

Во всех предыдущих примерах использовались предопределенные потоки cin и cout. Возможно, что программе потребуются собственные потоки для ввода/вывода. Если в приложении нужно создать файл для ввода и вывода, то в него необходимо включить заголовочный файл fstream.h (файл fstream.h включает в себя iostream.h). Классы ifstream и ofstream являются производными от istream и ostream и наследуют операции выделения и вставки, соответственно. В следующей программе на C++ показано, как объявить файл для считывания и для записи при помощи ifstream и ofstream, соответственно:

// 04.12FSTRM

// Программа на C++, иллюстрирует объявление ifstream и ofstream для

// файлового ввода и вывода

//

//#include "stdafx.h"

//#include <stdio.h>

//#include <strstream>

//#include <iostream>

//using namespace std;

#include<fstream.h>

void **main(**void)

{

char **c;**

**ifstream ifsin(**"E:\\LECTURE\\text.in", ios::in);

if(!ifsin)

**cerr <<** "\nUnable to open 'text.in' for input.";

**ofstream ofsout(**"E:\\LECTURE\\text.out",ios::out);

if(!ofsout)

**cerr <<** "\nUnable to open 'text.out' for output."; // Нельзя открыть для вывода

while(ofsout && ifsin.get(c))

**ofsout.put(c);**

**ifsin.close();**

**ofsout.close();**

**cout <<** "\n\n";

}

Программа объявляет переменную ifsin класса ifstream и связывает ее с файлом TEXT.IN, расположенным на диске. В любой программе, работающей с файлами, желательно всегда проверять существование или результат создания указанного файла в заданном режиме. Используя описатель файла ifsin, при помощи простого оператора if можно проверить результат операции. Аналогичные действия выполняются с ofsout за исключением того, что файл является производным от класса ofstream. Цикл while начинает ввод и вывод отдельных символов, если файл ifsin существует и считанный символ не равен EOF. В конце программы оба файлa закрываются. Закрытие выходного файла может оказаться существенным для сбрасывания на диск всех буферизированных в памяти данных.

Могут возникнуть ситуации, когда файл описывается не сразу или когда несколько файловых потоков необходимо связать с одним дескриптором файла. Это демонстрируется в следующей программе:

ifstream ifsin;

…

ifsin.open("week1.in");

…

ifsin.close();

ifsin.open("week2.in");

…

ifsin.close();

Если в программе нужно изменить способ открытия или использования файла, то в конструкторах файловых потоков можно добавить второй аргумент. Например:

ofstream ofsout("week1.out", ios::app | ios::noreplace);

Этот оператор объявляет поток ofsout и связывает его с файлом по имени week1.out. Поскольку задана опция ios::noreplace (не замещать), то при наличии файла с таким именем он создаваться не будет. Параметр ios::app указывает на добавление всех операций записи к существующему файлу. В следующей таблице перечислены флаги второго аргумента для конструкторов файловых потоков, которые могут логически объединяться по условию ИЛИ:

|  |  |
| --- | --- |
| **Бит режима** | **Действие** |
| ios:: in | Открыть для считывания |
| ios:: out | Открыть для записи |
| ios:: ate | Искать EOF после создания файла |
| ios:: app | Все записи добавляются к файлу |
| ios:: trunc | Если файл существует, то он очищается |
| ios:: nocreate | Открыть только существующий файл |
| ios:: noreplace | Открыть только несуществующий файл |
| ios:: binary | Открыть файл в двоичном режиме (по умолчанию — текстовый) |

Для открытия файла для считывания и записи можно также использовать объект класса **fstream**. Например: следующее описание открывает файл **UPDATE.DAT** в режиме считывания и записи с добавлением:

fstream io("update.dat", ios::in | ios::app);

**Файлы с произвольным доступом.**

Произвольный доступ в системе ввода-вывода реализуется с помощью функций **seekg()** и **seekp()**, используемых для позиционирования, соответственно, входного и выходного потока. Каждая из них имеет по два прототипа:

istream& seekg(pos type pos);

istream\_type& seekg (of f\_type& offset,

**ios\_base::seekdir dxr);**

ostream& seekp(pos\_type pos);

ostream\_type& seekp (off\_type offset,

**ios\_base::seekdir dir);**

Здесь параметр **pos** задает абсолютную позицию в файле относительно начала файла. Параметр **offset** задаст смещение в файле, а параметр **dir** — направление смещения которое может принимать значения:

ios:: beg — смещение от начала файла;

ios:: cur — смещение относительно текущей позиции;

ios:: end — смещение от конца файла

С каждым потоком связан указатель позиционирования, который изменяет свое значение в результате операции ввода или вывода. Для выполнения операций произвольного доступа файл должен открываться в двоичном режиме.

Получить текущее значение позиции в потоке ввода или вывода можно с помощью функций **tellg()** и **tellp()**, соответственно. Эти функции имеют следующие прототипы:

pos\_type tellg();

pos\_type tellp();

Следующий пример демонстрирует возможность позиционирования потока ввода информации:

//04.02

#include <cmath>

#include <iomanip>

#include <string.h>

#include <fstream>

#include <iostream>

using namespace std;

//using std::cout;

//using std::cin;

int main(int argc, char\* argv[])

{

setlocale(LC\_ALL, "Rus");

int size = 0;

if (argc > 1){

const char \*FileName = argv[1];

ifstream file;

file.open(FileName,ios::in | ios::binary);

if (file){

file.seekg(0, ios::end);

size = file. tellg();

if (size < 0){

cerr << FileName

<< " не найден.";

return 1;

}

cout << FileName << " size = " << size;

}

}

else

cout << "Вы не задали имя файла.";

getchar(); getchar();

return 0;

}



Программа выводит на экран длину заданного файла.

Позиционирование для всех типов класса iostream можно осуществлять при помощи методов класса **seekg()** или **seekp()**, которые могут устанавливаться на абсолютный адрес в файле или смещаться на заданное число байт от конкретного положения. Оба метода, **seekg()** (установить или считать положение указателя считывания) и **seekp()** (установить или читать положение указателя записи), могут иметь один или два аргумента. Если используется один параметр, то **iostream** устанавливается в заданное положение указателя; если два — вычисляется относительное положение. В следующем листинге отмечены эти различия; предполагается приведенное выше объявление для **io**:

streampos current\_position = io.tellp();

io << obj1 << obj2 << obj3;

io.seekp(current\_position);

io.seekp(sizeof(MY\_OBJ), ios:: cur);

io << objnewobj2;

Сначала от **streampos** порождается указатель **current\_position**, начальное значение которого получается при помощи функции **tellp()** и равно текущему положению файлового указателя записи. После запоминания этой информации в поток **io** записываются три объекта. При помощи **seekp()** указатель записи позиционируется на начало файла. Во втором операторе **seekp()** с помощью операции **sizeof()** вычисляется количество байтов, необходимое для смещения в файле на длину одного объекта. В результате объект **obj1** пропускается, и возможна запись нового объекта **objnewobj2**.

Если метод **seekg()** или **seekp()** используется с двумя аргументами, то второй параметр определяет тип перемещения: **ios:: beg** (от начала), **ios:: cur** (от текущего положения) и **ios:: end** (от конца файла). Например: следующий оператор выполняет перемещение указателя считывания на 5 байт от текущего положения:

io.seekg(5, ios::cur);

Следующий оператор перемешает указатель **get\_file** на 7 байт назад относительно конца файла:

io.seekg(-7, ios::end);

## Классы ios\_base и ios.

В классах ios\_base и ios определены методы, которые являются общими для входных и выходных потоков. Класс ios\_base содержит методы, которые не зависят от параметров шаблона. Наоборот, класс ios содержит шаблонно-зависимые методы. Объект класса ios\_base не может быть создан непосредственно, а только в наследуемых классах.

Объекты типа ios\_base поддерживают следующую информацию о состоянии потока.

Информацию о форматировании ввода-вывода:

* флаги форматирования,
* длину полей ввода-вывода,
* разрешимость дисплея.

Информацию о состоянии потока:

* состояние ошибки,
* маску исключений,

Другая информация:

* стек вызовов callback функций при наступлении некоторых событий,
* внутренний массив с элементами типа long,
* внутренний массив с элементами типа void\*.

Перечислим некоторые методы класса ios\_base:

flags чтение-установка флагов форматирования,

precision чтение-установка точности double,

setf установка некоторых флагов форматирования,

unsetf сброс флагов форматирования.

Например, в следующей программе показано как установить в качестве базовой 16 с/с.

#include<iostream>

usingnamespace **std;**

int **main ()**

{

// установить 16c/c в качестве базовой

**cout.setf ( ios\_base::hex, ios\_base::basefield );**

// показывать базу

**cout.setf ( ios\_base::showbase );**

**cout << 100 << endl;**

// не показывать базу

**cout.setf ( 0, ios\_base::showbase );**

**cout << 100 << endl;**

**cin.get();**

return **1;**

}



**Класс ios поддерживает следующую информацию о потоке:**

* символ заполнитель,
* указатель на поток вывода, связанный с объектом типа ios,
* указатель на объект типа streambuf, связанный с объектом типа ios.

Перечислим некоторые методы класса ios:

operator возвращает true, если установлен любой из флагов failbit или badbit, иначе возвращает false,

bad возвращает true в случае неустранимой ошибки ввода-вывода,

clear сбрасывает управляющие состояния,

eof проверяет на конец файла,

exception читает-устанавливает маску исключений,

fail проверяет на ошибку, исключая конец файла,

fill читает-устанавливает символ заполнитель,

good возвращает true, если нет ошибок,

rdstate читает управляющие состояния,

setstate устанавливает управляющие состояния.

Кроме того, класс ios поддерживает режимы работы потока, которые задаются следующими флагами:

ios::app данные всегда записываются в конец файла,

ios::ate первый байт записывается в конец файла, остальные байты с текущей позиции,

ios::binary бинарный файл,

ios::in входной файл,

ios::nocreate открыть существующий файл, если файла нет, то ошибка,

ios::noreplace открыть новый файл, если файл уже есть, то ошибка,

ios::out выходной файл,

ios::trunk файл открывается и его содержимое стирается,

Например, в следующей программе показано, как проверить, успешно ли открыт файл.

#include<iostream>

#include<fstream>

usingnamespace **std;**

int **main ()**

{

**ifstream in;**

**in.open (**"test.txt");

if **(!in)**

**cerr <<** "Error: open file 'test.txt' failed." **<< endl;**

**cin.get();**

return **0;**

}



## Потоки вывода.

Потоки вывода создаются на базе класса ostream, который обеспечивает методы для записи данных в буфер потока. Перечислим основные методы класса ostream:

operator<< форматированный вывод данных в поток,

flush очищает буфер,

put выводит символ,

seekp устанавливает позицию для функции put,

tellp читает позицию указателя для функции put,

write пишет последовательность байтов,

От класса ostream наследуются класс ofstream, который содержит дополнительные методы:

open открыть файл,

close закрыть файл,

is\_open проверка открыт ли файл.

Сейчас приведем прототипы этих функций и примеры их использования.

### Функции open, close и is\_open

**Функции open, close и is\_open** имеют следующие прототипы:

void **open (** constchar **\* filename, openmode mode = out | trunc );**

void **close ( );**

bool **is\_open ( );**

При открытии потока вывода нужно указать имя файла и режимы работы потока с этим файлом. При этом нужно запомнить, что режимы работы с потоками ввода и вывода совпадают.

Например, следующая программа добавляет предложение в конец текстового файла и показывает, как использовать вышеперечисленные функции.

//#include <iostream.h>

#include<iostream>

#include<fstream>

usingnamespace **std;**

int **main ()**

{

**ofstream out;**

**out.open (**"test.txt", ofstream::out | ofstream::app);

if **(out.is\_open())**

**out <<** "This sentence is appended to the file content.\n";

else

**cerr <<** "Open file failed." **<< endl;**

**out.close();**

//cin.get();

return **0;**

}

### Функция flush

**Функция flush очищает буфер.** Эта функция имеет следующий прототип:

ostream& flush ( );

Ниже приведен пример использования функции flush. Этот пример также показывает, как можно создавать текстовый файл

//#include <fstream.h>

#include<fstream>

#include<iostream>

usingnamespace **std;**

int **main ()**

{

**ofstream out(**"test.txt");

for **(**int **n=0; n<100; n++)**

**{**

**out << n <<** ' ';

**out.flush();**

**}**

**out <<** '\n'; // переход на новую строку

for **(**int **n=0; n<100; n++)**

**out << (n+0.1) <<** ' ' **<< flush;**

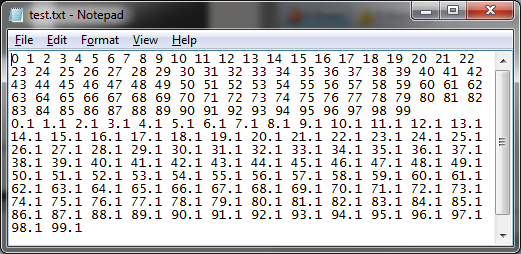
**out << endl;** // переход на новую строку

**out.close();**

**cin.get();**

return **0;**

}



### Функция put

**Функция put выводит символ в файл.** Эта функция имеет следующий прототип:

ostream& put ( char **ch );**

В следующем примере показано, как водить символы с консоли и выводить их в файл.

//#include <iostream.h>

//#include <fstream.h>

#include<iostream>

#include<fstream>

usingnamespace **std;**

int **main ()**

{

char **ch;**

**ofstream out(**"test.txt");

**cout <<** "Input chars then input '.' to exit." **<< endl;**

do

**{**

**ch = cin.get();**

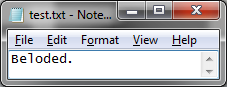
**out.put(ch);**

**}** while **(ch!=**'.');

return **0;**

}





### Функция seekp и tellp

**Функция seekp устанавливает, а функция tellp читает позицию в файле для функции put.** Эти функции имеют следующие прототипы:

streampos tellp ( );

ostream& seekp ( streampos pos );

где pos - новая позиция в потоке;

ostream& seekp ( streamoff off, ios\_base::seekdir dir );

где параметры имеют следующее назначение:

off смещение в потоке относительно позиции, указанной в параметре dir.

dir направление поиска позиции, может принимать одно из следующих значений:

* ios\_base::beg – поиск от начала файла;
* ios\_base::cur - поиск от текущей позиции;
* ios\_base::end – поиск от конца файла.

Ниже приведен пример использования этих функций.

//#include <fstream.h>

#include<fstream>

#include<iostream>

usingnamespace **std;**

int **main ()**

{

long **pos;**

**ofstream out;**

**out.open (**"test.txt");

**out <<** "This is a string." **<< endl;**

**pos=out.tellp();**

**out.seekp (pos - 11);**

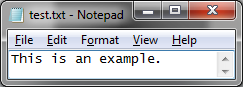
**out <<** "an example.";

**out.close();**

**cin.get();**

return **0;**

}



### Функция write

**Функция write пишет последовательность символов (байтов) в файл.** Эта функция имеет следующий прототип:

ostream& write ( constchar\* s , size n );

где s указывает на область памяти, из которой данные записываются в файл, а n – количество записываемых символов (байтов). Отметим, что эта функция используется для записи данных в бинарные файлы, то есть содержимое области памяти записывается в файл без изменения. Ниже приведен пример создания бинарного файла, используя функцию write.

//#include <iostream.h>

//#include <fstream.h>

#include<iostream>

#include<fstream>

usingnamespace **std;**

struct **emp**

{

int **code;**

char **name[20];**

double **salary;**

};

int **main()**

{

**ofstream out;** // выходной поток

struct **emp s;** // для записей файла

// открываем выходной поток в бинарном режиме

**out.open(**"demo.bin", ofstream::binary);

if(!out)

**{**

**cerr <<** "Open file failed." **<< endl;**

return **1;**

**}**

**cout <<** "Input code, name and salary." **<< endl;**

**cout <<** "Press Ctrl+z to exit." **<< endl;**

**cout <<** '>';

// вводим первую запись с консоли

**cin >> s.code >> s.name >> s.salary;**

while **(!cin.eof())**

**{**

// пишем запись в файл

**out.write((**constchar\*)&s, sizeof(struct **emp));**

**cout <<** '>';

// вводим следующие записи с консоли

**cin >> s.code >> s.name >> s.salary;**

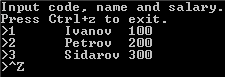
**}**

// закрываем выходной поток

**out.close();**

return **0;**

}



От класса ostream также наследуются класс ostringstream, который обеспечивает интерфейс для работы со строками.

Класс ostringstream содержит дополнительный метод:

str – чтение и запись строки в поток.

### Функция str

**Функция str выполняет запись и чтение строки в поток.** Эта функция имеет следующий прототип:

string str ( ) constчтение строки из потока

void **str ( string & s );** запись строки s в поток

Например, в следующей программе показано, как ввести строку в поток типа ostringstream, а затем прочитать его содержимое в объект типа string.

#include<iostream>

#include<sstream>

#include<string>

usingnamespace **std;**

int **main ()**

{

**ostringstream oss;**

**string s;**

**oss.str(**"Sample string\_1.");

**s = oss.str();**

**cout << s << endl;**

**oss <<** "Sample string\_2.";

**s = oss.str();**

**cout << s << endl;**

**cin.get();**

return **0;**

}



### Классы потокового вывода.

Все классы **ofstream**, производные от **fstreambase** и **ostream**, позволяют выполнять форматированный и неформатированный вывод в поток **streambuf**. Ниже приводится программа, которую можно сравнить с примером из предыдущего материала.

В программе используется конструктор ofstream, похожий на конструктор ifstream, описанный выше. Указывается имя файла, "**myostrm.out**", и режим открытия файла, **ios::out**.

// 05.17OSTRM.CPP

// применение класса ofstream

// допустимые методы класса ofstream

// ofstream::open ofstream::rdbuf

// допустимые методы класса ostream

// ostream::flush ostream::ostream

// ostream::put ostream::seekp

// ostream::tellp ostream::write

//

#include "stdafx.h"

#include <iostream>

#include <fstream>

#include <math.h>

using namespace std;

#define iSTRING\_MAX 40

void main(void)

{

int i=0;

long ltellp;

char pszString[iSTRING\_MAX]="Simple test string\n";/\* 6+1+4+1+6=18+(’\’)+(’n’)=20 \*/

// по умолчанию файл открывается втекстовом режиме

ofstream ofMyOutputStream("myostrm.out", ios::out);

// посимвольный вывод строки

// строка '\n' преобразуется в два символа

while(pszString[i] != '\0')

{

ofMyOutputStream.put(pszString[i]);

ltellp = ofMyOutputStream.tellp();

cout << "\ntellp value: " << ltellp;

i++;

}

// выод строки целиком при помощи метода класса

ltellp = ofMyOutputStream.tellp();

cout << "\ntellp's value before writing 2nd string: " << ltellp; // значение tellp перед выводом 2 строки

ofMyOutputStream.write(pszString, strlen(pszString));

ltellp=ofMyOutputStream.tellp();

cout << "\ntellp's updated value: " << ltellp << '\n'; // обновленное значение tellp

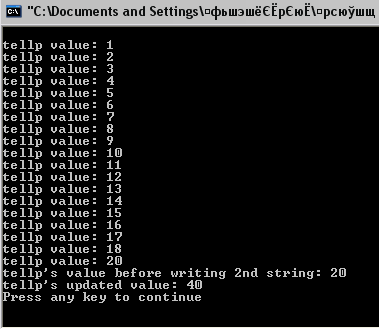
ofMyOutputStream.close();

}

В первом цикле while выполняется вывод строки pszString с использованием метода put(). После печати каждого символа переменной ltellp присваивается значение текущей позиции указателя, которое возвращает метод класса tellp(). В данный момент важно обратить внимание на распечатку результатов работы программы.

После инициализации строковая переменная pszString представляет собой 19 символов, заканчивающихся null-символом '\0'; таким образом, общее число символов — 20. Хотя программа выводит значения tellp от 1 до 20, однако двадцатый символ не является null-символом. Это объясняется тем, что в текстовом режиме символы \n из строки pszString преобразуются в два байта: один байт для символа возврата каретки (19-й символ) и второй — для перевода строки (20-й символ). null-символ не выводится.

В остальной части программы вычисляется позиция указателя вывода до и после вызова метода write() для печати строки pszString целиком. Заметим, что напечатанные значения tеllр говорят о том, что функция write() также преобразует один null-символ в два выводимых байта. Если бы этого преобразования не было, то последнее значение tе11р было бы равно 39 (с учетом того, что после выполнения put() счетчик остановился на числе 20, а не на 19-ти). Сокращенная распечатка результатов работы программы выглядит следующим образом:



К счастью, производные от iostream методы get() и read() автоматически преобразуют два байта, получающиеся при выводе, в один символ '\n'. Приведенная программа обращает внимание на особенности файлового ввода/вывода, требующие осторожности. Если файл, созданный данной программой, использовать в качестве входного файла, открываемого в двоичном режиме, то возникнет конфликт, поскольку в двоичных файлах подобного преобразования не выполняется, поэтому как позиционирование, так и содержимое файла будут неправильными.

## Потоки ввода.

Потоки ввода являются объектами класса istream, который обеспечивает методы для чтения информации из буфера потока. Перечислим основные методы класса istream:

operator >> форматированный ввод,

gcount возвращает количество символов, прочитанных последней операцией ввода,

get ввод символа,

getline ввод строки,

ignore удаление символов из буфера,

peek читает символ, но не удаляет его из буфера,

putback возвращает символ в буфер,

read читает блок данных,

seekg устанавливает указатель позиции файла для метода get,

sync синхронизирует буфер потока с внешним устройством,

tellg получает указатель позиции файла для метода get,

unget возвращает символ в поток,

От класса istream наследуется класс ifstream, который обеспечивает интерфейс для работы с файлами. Класс ifstream содержит дополнительные методы:

open открыть файл,

close закрыть файл,

is\_open проверить, открыт ли файл.

### Функция gcount

**Функция gcount возвращает количество символов, прочитанных последней не форматирующей операцией ввода.** Эта функция имеет следующий прототип:

streamsize gcount ( ) const;

К не форматирующим операциям ввода относятся следующие операции: get, getline, ignore, read и readsome. Пример использования функции gcount показан в следующей программе:

#include<iostream>

#include<string>

usingnamespace **std;**

int **main()**

{

char **line[80];**

int **length;**

**cout <<** "Input a string: " **<< endl;**

**cin.getline(line, 80);**

**length = cin.gcount();**

**cout <<** "String length = " **<< length-1 << endl;**

**cin.get();**

return **0;**

}



### Функция get

**Функция get вводит символ из входного потока.** Эта функция имеет следующий прототип:

int **get();**

возвращает символ, извлеченный из потока;

istream& get (char& c );

извлекает символ из потока и сохраняет его в переменной с;

istream& get (char\* s, streamsize n, char **delim = ‘\n’ );**

вводит из потока символы и записывает их в массив s, ввод символов прекращается в следующих случаях:

* если введен (n-1) символ;
* если в потоке встретился символ разделитель delim, сам символ разделитель из потока не извлекается;
* файл закончился раньше, чем прочитан (n-1) символ.

После записи прочитанных символов в массив s функция get записывает в этот массив пустой символ NULL, отмечающий конец строки. Ниже приведена программа, которая распечатывает содержимое текстового файла.

#include<iostream>

#include<fstream>

usingnamespace **std;**

int **main()**

{

**ifstream in;**

**in.open (**"test.txt", ifstream::in);

while **(in.good())**

**cout << (**char) in.get();

**cout << endl;**

**in.close();**

**cin.get();**

return **0;**

}



А теперь приведена программа, которая читает символы из входного потока, используя функцию get.

//#include <iostream.h>

#include<iostream>

usingnamespace **std;**

int **main()**

{

char **c;**

char **a[3];**

**cout <<** "Input a char: ";

**c = cin.get();**

**cout << c << endl;**

**cin.get();** // удаляем символ конца строки

**cout <<** "Input a char: ";

**cin.get(c);**

**cout << c << endl;**

**cin.get();** // удаляем символ конца строки

**cout <<** "Input a string: ";

**cin.get(a, 3);**

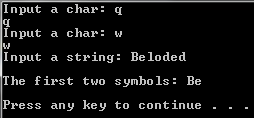
**cout << endl <<** "The first two symbols: ";

**cout << a << endl << endl;**

**system(**"pause");

return **0;**

}



### Функция getline

**Функция getline вводит строки из файла.** Эта функция имеет следующий прототип:

istream& getline (char\* s, streamsize n, char **delim =** '\n');

вводит из потока символы и записывает их в массив s, ввод символов прекращается в следующих случаях:

* если введен (n-1) символ;
* если в потоке встретился символ разделитель delim, причем сам символ разделитель извлекается из потока, но не записывается в массив s;
* файл закончился раньше, чем прочитан (n-1) символ.

После записи прочитанных символов в массив s функция getline записывает в этот массив пустой символ NULL, отмечающий конец строки. Ниже приведена программа, которая вводит строки из потока и распечатывает их на консоли.

//#include <iostream.h>

#include<iostream>

usingnamespace **std;**

int **main ()**

{

char **line[80];**

**cout <<** "Input strings. " **<< endl;**

**cout <<** "To exit press CTRL+Z." **<< endl;**

**cout <<** '>';

while **(!cin.eof())**

**{**

**cin.getline(line, 80);**

**cout <<** ' ' **<< line << endl;**

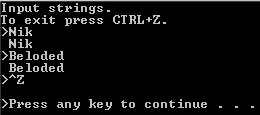
**cout <<** '>';

**}**

**system(**"pause");

return **0;**

}



### Функция ignore

**Функция ignore удаляет символы из буфера потока.** Эта функция имеет следующий прототип:

istream& ignore ( streamsize n = 1, int **delim = EOF );**

где n – количество символов, которые необходимо удалить из потока, а delim – это символ ограничитель. Функция удаляет символы из потока до тех пор, пока не удалит n символов или не встретит символ разделитель. Например, в следующей программе выводятся только первые символы введенных слов.

#include<iostream>

usingnamespace **std;**

int **main ()**

{

char **c;**

**cout <<** "Input a word: ";

**c = cin.get();**

**cin.ignore(256,**'\n');

**cout <<** "The first letter is: " **<< c << endl;**

**cout <<** "Input another word: ";

**cin.ignore();**

**c = cin.get();**

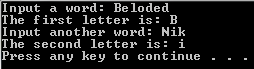
**cin.ignore(256,**'\n');

**cout <<** "The second letter is: " **<< c << endl;**

**system(**"pause");

return **0;**

}



### Функция peek

**Функция peek читает символ, но не удаляет его из буфера**. Эта функция имеет следующий прототип:

int **peek ( );**

возвращает прочитанный символ, а если файл закончился или в потоке произошла ошибка, то возвращает EOF.

Ниже приведена программа, в которой используется функция peek.

#include<iostream>

usingnamespace **std;**

int **main ()**

{

char **c;**

int **n;**

char **s[256];**

**cout <<** "Enter a number or a word: ";

**c = cin.peek();**

if **((c >=** '0') && (c <= '9'))

**{**

**cin >> n;**

**cout <<** "It is a number: " **<< n << endl;**

**}**

else

**{**

**cin >> s;**

**cout <<** "It is a word: " **<< s << endl;**

**}**

**system(**"pause");

return **0;**

}





### Функция putback

**Функция putback возвращает символ в буфер.** Эта функция имеет следующий прототип:

istream& putback (char **ch);**

если введенный символ совпадает с символом ch, то функция возвращает его в буфер ввода, иначе поведение функции непредсказуемо.

Использование функции putback показано в следующей программе.

#include<iostream>

usingnamespace **std;**

int **main ()**

{

char **c;**

int **n;**

char **s[256];**

**cout <<** "Enter a number or a word: ";

**c = cin.get();**

**cin.putback (c);**

if **((c >=** '0') && (c <= '9'))

**{**

**cin >> n;**

**cout <<** "It is a number: " **<< n << endl;**

**}**

else

**{**

**cin >> s;**

**cout <<** " It is a word: " **<< s << endl;**

**}**

**system(**"pause");

return **0;**

}





### Функция read

**Функция read читает блок данных из файла и имеет следующий прототип:**

istream& read (char\* s, streamsize n);

где s указывает на область памяти, в которую данные читаются из файла, а n – количество читаемых символов (байтов). Отметим, что эта функция используется для чтения данных из бинарных файлов, то есть содержимое файла без изменения читается в область памяти. Ниже приведен пример чтения записей бинарного файла, используя функцию read.

//чтение бинарного файла

//#include <iostream.h>

//#include <fstream.h>

#include<iostream>

#include<fstream>

usingnamespace **std;**

struct **emp**

{

int **code;**

char **name[20];**

double **salary;**

};

int **main()**

{

**ifstream in;** // выходной поток

struct **emp s;** // для записей файла

unsigned **i;** // номер записи

// открываем входной поток в бинарном режиме

**in.open(**"demo.bin");

if(!in)

**{**

**cout <<** "Open file failed.\n";

return **1;**

**}**

**cout <<** "Press Ctrl+z to exit.\n";

// читаем индекс

**cout <<** "Input an index: ";

**cin >> i;**

while **(!cin.eof())**

**{**

// устанавливает указатель на нужную запись

**in.seekg(i\***sizeof(struct **emp));**

// читаем запись из файла

**in.read((**char\*)&s, sizeof(struct **emp));**

if **(!in.good())**

**{**

**cout <<** "The wrong index.\n";

**cout <<** "Input an index: ";

**cin >> i;**

**in.clear();** // очищаем ошибку

continue;

**}**

// выводим запись на консоль

**cout <<** "\tcode = " **<< s.code**

**<<** " name = " **<< s.name**

**<<** " sal = " **<< s.salary << endl,**

// читаем индекс

**cout <<** "Input an index: ";

**cin >> i;**

**}**

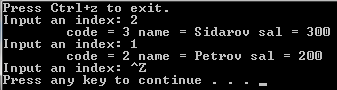
// закрываем входной поток

**in.close();**

**system(**"pause");

return **0;**

}



### Функция seekg

**Функция seekg устанавливает указатель позиции файла для метода get, а функция tellg получает указатель позиции файла для метода get.** Эти функции имеют следующие прототипы:

istream& seekg (streampos pos);

устанавливает позицию файла на значение pos, тип streampos эквивалентен типу long.

istream& seekg (streamoff off, ios\_base::seekdir dir);

сдвигает позицию файла на значение off относительно значения параметра dir, который может принимать следующие значения:

ios\_base::beg смещение относительно начала файла;

ios\_base::cur смещение относительно текущей позиции;

ios\_base::end смещение относительно конца файла.

streampos tellg ( );

возвращает указатель позиции файла для функции get, тип streampos эквивалентен типу long.

Ниже приведена программа, которая демонстрирует использование вышеуказанных функций.

// чтение файла в память

//#include <iostream.h>

//#include <fstream.h>

#include<iostream>

#include<fstream>

usingnamespace **std;**

struct **emp**

{

int **code;**

char **name[20];**

double **salary;**

};

int **main()**

{

int **length;**

**emp\* buffer;**

**ifstream in;**

**in.open(**"demo.bin", ios::binary);

if **(!in)**

**{**

**cout <<** "Open file failed" **<< endl;**

return **1;**

**}**

// определеям длину файла

**in.seekg(0, ios::end);**

**length = in.tellg();**

**in.seekg(0, ios::beg);**

// распределяем память под файл

**buffer = (emp\*)** newchar[length];

// читаем файл

**in.read((**char\*)buffer,length);

**in.close();**

// выводим содержимое на консоль

for **(**int **i=0; i<length/(**int)sizeof(emp); ++i)

**cout << buffer[i].code <<** ' '

**<< buffer[i].name <<** ' '

**<< buffer[i].salary << endl;**

**system(**"pause");

return **0;**

}



### Функция sync

**Функция sync синхронизирует буфер потока с внешним устройством.** Эта функция имеет следующий прототип:

int **sync ( );**

Как работает эта функция, и что такое синхронизация буфера потока с внешним устройством смотри в параграфе 1.6.

### Функция unget

**Функция unget возвращает в поток символ, предварительно извлеченный из потока.** Эта функция имеет следующий прототип:

int **unget();**

возвращает символ, извлеченный из потока.

Ниже приведена программа, в которой показано использование функции unget.

#include<iostream>

usingnamespace **std;**

int **main ()**

{

char **c;**

int **n;**

char **s[256];**

**cout <<** "Enter a number or a word: ";

**c = cin.get();**

**cin.unget();**

if **((c >=** '0') && (c <= '9'))

**{**

**cin >> n;**

**cout <<** "It is a number: " **<< n << endl;**

**}**

else

**{**

**cin >> s;**

**cout <<** "It is a word: " **<< s << endl;**

**}**

**system(**"pause");

return **0;**

}





От класса istream наследуется класс istringstream, который обеспечивает интерфейс для работы со строками. Класс istringstream содержит дополнительный метод:

str

чтение или запись строки в поток.

### Функция str

**Функция str для записи и чтения строки в поток.**

Прототипы этой функции были рассмотрены раньше в пункте 6. Сейчас же приведем только программу, в которой показано, как записать строку в поток типа istringstream, а затем вывести содержимое этого потока.

#include<iostream>

#include<sstream>

#include<string>

usingnamespace **std;**

int **main ()**

{

int **n;**

**istringstream iss;**

**string s =** "0 1 2";

**iss.str(s);**

for **(**int **i = 0; i < 3; ++i)**

**{**

**iss >> n;**

**cout << n + 1 << endl;**

**}**

**system(**"pause");

return **0;**

}



### Классы потокового ввода.

Используемый в следующем примере класс ifstream порожден от классов fstreambase и istream. Он выполняет операции ввода из потока filebuf. Данная программа осуществляет потоковый ввод текста.

// 05.17IFSTRM.CPP

// применение класса ifstream

// допустимые методы класса ifstream

// ifstream::open ifstream::rdbuf

// istream::gcount istream::get

// istream::getline istream::ignore

// istream::istream istream::peek

// istream::putback istream::read

// istream::seekg istream::tellg

//

#include "stdafx.h"

#include <iostream>

#include <fstream>

#include <math.h>

using namespace std;

#define iCOLUMNS 80

void main(void)

{

char cOneLine[iCOLUMNS];

ifstream ifMyInputStream("ifstrm.cpp", ios::in);

while(ifMyInputStream)

{

ifMyInputStream.getline(cOneLine, iCOLUMNS);

cout << '\n' << cOneLine;

}

ifMyInputStream.close();

}

В первом операторе программы для создания объекта ifstream и связывания его с дескриптором открытого файла ifMyInputStream используется конструктор класса ifstream. Этот оператор содержит имя файла, включая при необходимости путь к файлу ("ifstrm.cpp "), а также один или несколько режимов (например, ios::in | ios::nocreate | ios::binary). По умолчанию ввод — текстовый. При наличии опции ios::nocreate выполняется проверка существования файла. Целое значение дескриптора файла ifMyInputStream можно использовать при логических проверках, например в операторах if и while, это значение автоматически устанавливается в ноль при достижении конца файла — условие ЕОF.

Метод getline(), унаследованный от класса ifstream, позволяет читать полные строки текста, заканчивающиеся null-символом. Этот метод имеет три формальных параметра: указатель char \*, количество вводимых символов — включая null-символ — и необязательный символ-разделитель (по умолчанию — '\n').

Поскольку имена массивов char на самом деле являются указателями на символы, переменная cOneLine удолетворяет требованиям первого параметра метода. Количество вводимых символов соответствуют размеру массива — iCOLUMNS. Опция разделителя отсутствует. Однако, если входные строки разделяются специальным символом — например, '\*' -, то оператор getline() можно записать так:

ifMyInputStream.getline(cOneLine, iCOLUMNS, '\*') ;

Затем в приведенной программе печатаются строки, и файл закрывается явно при помощи метода ifMyInputStream.close().

Как работать с этими функциями, было показано в предыдущем параграфе. Сейчас же подробно рассмотрим работу с функциями класса istream.

## Потоки ввода-вывода.

Потоки ввода-вывода создаются на базе класса iostream, который является производным классом от классов istream и ostream и поэтому наследует все их методы. Для работы с файлами и со строками из класса iostream наследуются соответственно классы fstream и stringstream, которые позволяют, как писать данные в потоки, так и читать данные из потоков. Для наглядного представления отношения наследования между этими классами смотри диаграмму наследования в параграфе 1.1.

Здесь же заметим, что заголовочный файл <iostream> содержит потоки ввода-вывода: cin, cout, cerr и clog, примеры работы с которыми смотри в параграфе 1.8.

## Классы буферизированных потоков.

Класс streambuf является основой для потокового ввода/вывода в С++. В этом базовом классе описаны все основные операции, выполняемые с символьными буферами. Класс streambuf также используется для порождения класса файловых буферов (класс filebuf) и классов istream и ostream, содержащих указатели на объекты streambuf.

Все производные классы, базирующиеся на классе ios, наследуют указатель на streambuf. Как показано на рисунке, класс filebuf происходит от streambuf и адаптирует этот родительский класс к работе с файлами.

03_02

В начале следующей программы описываются два описателя (handle) типа filebuf — fbMyInputBuf и fbMyOutputBuf после этого при помощи метода ореn() открываются текстовые файлы. Затем — при условии отсутствия ошибок открытия файлов — каждый описатель связывается с соответствующим объектом istream (для ввода) и ostream (для вывода). Когда оба файла открыты, в цикле while выполняется обычная эхо-печать от входного потока is.get() в выходной поток os.put(), и при этом подсчитывается число символов перевода строки '\n'. Необязательный в данном случае метод сlоsе() используется для явного закрытия файлов.

// 05.17FILBUF.CPP

// применение класса filebuf

// допустимые методы класса

// filebuf::attach filebuf::close

// filebuf::fd filebuf::~filebuf

// filebuf::filebuf filebuf::is\_open

// filebuf::open filebuf::overflow

// filebuf::seekoff filebuf::setbuf

// filebuf::sync filebuf::underflow

//

#include "stdafx.h"

#include <iostream>

#include <fstream>

#include <math.h>

using namespace std;

void main(void)

{

char ch;

int iLineCount = 0;

filebuf fbMyInputBuf, fbMyOutputBuf;

//fbMyInputBuf.open("filbuf.cpp",\_O\_RDONLY | \_O\_TEXT);

fbMyInputBuf.open("filbuf.cpp", 1);

if(fbMyInputBuf.is\_open() == 0)

{

cerr << "Can't open input file" << '\n';

exit(1);

}

istream is(&fbMyInputBuf);

//fbMyOutputBuf.open("output.dat",\_O\_WRONLY | \_O\_TEXT);

fbMyOutputBuf.open("output.dat", 2);

if(fbMyOutputBuf.is\_open() == 0)

{

cerr << "Can't open output file" << '\n';

exit(2);

}

ostream os(&fbMyOutputBuf);

while(is)

{

is.get(ch);

os.put(ch);

iLineCount += (ch == '\n');

}

fbMyInputBuf.close();

fbMyOutputBuf.close();

cout << "You have " << iLineCount << " lines in your file" << endl;

// В Вашем файле ... строк

}

## Класс строковых потоков.

Класс streambuf можно использовать для расширения возможностей класса iostream. На рисунке ранее показаны соотношения между классом ios и производными классами. Именно класс ios предоставляет порожденным классам интерфейс программирования и средства форматирования. Однако, все необходимые действия выполняют глобальные методы и виртуальные функции класса streambuf. Все производные от ios классы вызывают эти процедуры.

Все буферизированные объекты класса streambuf используют фиксированный буфер памяти, называемый областью резервирования. Эту область можно разделить на get-область для ввода и put-область для вывода. При необходимости области ввода и вывода могут накладываться друг на друга. При помощи защищенных методов класса программа получает к ним доступ и может манипулировать двумя отдельными указателями для ввода и для вывода, использующимися для символьного ввода/вывода. Учитывая реализацию производного класса в конкретной программе, каждое приложение определяет порядок использования буферов и указателей.

Для объектов streambuf имеются два конструктора, имеющие следующий синтаксис:

streambuf::streambuf();

streambuf::streambuf (char **\* рг,** int **nLength);**

Первый конструктор используется косвенно всеми порожденными от streambuf классами. Он устанавливает в null все внутренние указатели объекта типа streambuf. Второй конструктор создает объект типа streambuf, связывающийся с существующим символьным массивом. В следующей программе показано, как объявить строковый объект strstreambuf, порожденный от базового класса streambuf. После создания объект stbMyStreamBuf программа выводит отдельный символ при помощи метода sputc() и затем считывает его обратно с помощью метода sgetc().

// 05.17STRBUF.CPP

// применение класса streambuf

//

#include "stdafx.h"

#include <iostream>

#include <fstream>

#include <math.h>

#include <strstream>

using namespace std;

#define iNYBUFFSIZE 1024

void main(void)

{

char c;

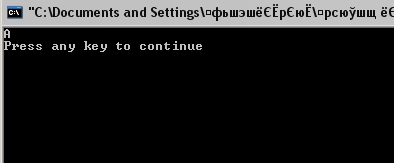
strstreambuf stbMyStreamBuf(iNYBUFFSIZE);

stbMyStreamBuf.sputc('A'); // отдельный символ выдается в буфер

c = stbMyStreamBuf.sgetc();

cout << c << endl;

}



Необходимо помнить о том, что для объектов, созданных на основе класса streambuf, имеются два различных указателя — для ввода и для вывода. Каждый указатель работает независимо от другого. Метод sgetc() считывает символ 'А'. Это объясняется тем, что возвращается содержимое буфера по адресу, на который ссылается указатель ввода. Метод sputc() смещает указатель вывода, но не затрагивает указатель ввода и символ, находящийся в буфере.

В следующем списке перечислены имена и назначение всех глобальных методов класса streambuf и указано, какие методы работают с указателями ввода и вывода:

|  |  |
| --- | --- |
| **Глобальный**  **метод** | **Назначение** |
| sgetc | Возвращает символ, на который ссылается указатель ввода, однако не перемещает указатель |
| sgetn | Читает последовательность символов из буфера streambuf |
| sputc | Помещает символ в область вывода и перемещает указатель вывода |
| sputn | Помещает цепочку символов в буфер streambuf и затем перемещает указатель вывода |
| snextc | Перемещает указатель ввода и возвращает следующий символ |
| sbumpc | Возвращает текущий символ и затем перемещает указатель ввода |
| stossc | Увеличивает указатель ввода на одну позицию, но не возвращает символ |
| sputbackc | Пытается сместить указатель ввода на одну позицию назад. Записываемый снова символ должен соответствовать предыдущему прочитанному символу |
| out\_waiting | Возвращает число символов в области вывода |
| in\_avail | Возвращает число символов в области ввода |
| dbp | Выводит статистику буфера streambuf и значения указателей |

В следующем перечне приведены имена и назначение всех виртуальных функций класса streambuf:

|  |  |
| --- | --- |
| **Виртуальная**  **функция** | **Назначение** |
| seekoff | Отрабатывает указанное смещение |
| seekpos | Переходит на указанную позицию |
| overflow | Очищает область вывода |
| underflow | Заполняет при необходимости область ввода |
| pbackfail | Расширяет метод sputbackc |
| setbuf | Пытается связать область резервирования с буфером streambuf |
| sync | Очищает области вывода и ввода |

В следующем списке перечислены имена и назначение всех защищенных методов класса streambuf:

|  |  |
| --- | --- |
| **Защищенный**  **метод** | **Назначение** |
| allocate | Выделяет буфер, вызывая функцию doallocate |
| doallocate | Выделяет область резервирования (виртуальная функция) |
| base | Возвращает указатель на начало области резервирования |
| ebuf | Возвращает указатель на конец области резервирования |
| blen | Возвращает размер области резервирования |
| pbase | Возвращает указатель на начало области вывода |
| pptr | Возвращает указатель вывода |
| gptr | Возвращает указатель ввода |
| eback | Возвращает нижнюю границу области ввода |
| epptr | Возвращает указатель на конец области вывода |
| egptr | Возвращает указатель на конец области ввода |
| setp | Устанавливает все указатели области вывода |
| setg | Устанавливает все указатели области ввода |
| pbump | Инкрементирует/декрементирует указатель вывода |
| gbump | Инкрементирует/декрементирует указатель ввода |
| setb | Устанавливает область резервирования |
| unbuffered | Устанавливает или проверяет переменную состояния буфера streambuf |

Очевидно, что в классе streambuf имеются практически все функции, которые могут понадобиться в программе для работы с потоковым буфером. Поскольку класс streambuf используется для построения класса файловых буферов (класс filebuf) и классов istream и ostream, все эти классы наследуют характеристики базового класса streambuf.

## Форматирование потока.

До сих пор мы использовали для вывода информации во всех примерах форматы, заданные в C++ по умолчанию. Для управления форматированием библиотека ввода-вывода предусматривает три вида средств:форматирующие функции, флаги и манипуляторы. Все эти средства являются членами класса basic\_ios и потому доступны для всех потоков.

Рассмотрим вначале форматирующие функции-члены. Их всего три: width (). precision () и fill ().

По умолчанию при выводе любого значения оно занимает столько позиций, сколько символов выводится. Функция width() позволяет задать минимальную ширину поля для вывода значения. При вводе она задает максимальное число читаемых символов. Если выводимое значение имеет меньше символов, чем заданная ширина поля, то оно дополняется символами-заполнителями до заданной ширины (по умолчанию - пробелами). Однако если выводимое значение имеет больше символов, чем ширина отведенного ему поля, то поле будет расширено до нужного размера. Эта функция имеет следующие прототипы:

streamsize width(streamsize wide);

streamsize width() const;

Тип streamsize определен в заголовочном файле <iostream> как целочисленный. Функция с первым прототипом задает ширину поля wide, а возвращает предыдущее значение ширины поля. Функция со вторым прототипом возвращает текущее значение ширины поля. По умолчанию она равна нулю, то есть вывод не дополняется и не обрезается. В ряде компиляторов после выполнения каждой операции вывода значение ширины поля возвращается к значению, заданному по умолчанию.

Функция precision () позволяет узнать или задать точность (число выводимых цифр после запятой), с которой выводятся числа с плавающей точкой. По умолчанию числа с плавающей точкой выводятся с точностью, равной шести цифрам. Функция имеет precision () следующие прототипы:

streamsize precision(streamsize prec);

streamsize precision() const;

Функция с первым прототипом устанавливает точность равной рrес и возвращает предыдущую точность. Функция со вторым прототипом возвращает текущую точность.

**Замечание**. Если не установлен флаг scientific или fixed, задает precision() задает общее число цифр.

Функция fill() позволяет прочесть или установить символ-заполнитель. Она имеет следующие прототипы:

char\_type fill(char\_type ch);

char\_type fill() const;

Функция с первым прототипом устанавливает ch в качестве текущего символа-заполнителя и возвращает предыдущий символ-заполнитель. Функция со вторым прототипом возвращает текущий символ-заполнитель. По умолчанию в качестве символа-заполнителя используется пробел. Тип данных char\_type является параметром класса basic\_ios и может обозначать набор "узких" или "широких" символов.

Рассмотрим пример программы, в котором используются форматирующие функции:

//05.01

#include <cmath>

#include <string.h>

#include <iostream>

using namespace std;

int main()

{

setlocale(LC\_ALL, "Rus");

double x;

cout.precision(4);

cout.fill('0');

cout << " x sqrt(x) x^2\n\n";

for (x= 1.0; x<= 6.0; x++){

cout.width(7);

cout << x << " ";

cout.width(7);

cout << sqrt(x) << " ";

cout.width(7);

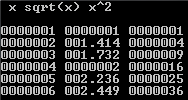
cout << x\*x << '\n';

}

getchar(); getchar();

return 0;

}



Эта программа выводит на экран небольшую таблицу значений переменной x, ее квадратного корня и квадрата:

х sqrt(x) х^2

0000001 00000001 0000001

0000002 0001.414 0000004

0000003 0001.732 0000009

0000004 0002.000 0000016

0000005 0002.236 0000025

0000006 0002.449 0000036

С каждым потоком связан набор флагов, которые управляют форматированием потока. Они представляют собой битовые маски, которые определены в классе ios как данные enum-типа fmt\_flags. Сами флаги принадлежат типу fmtflags, который определен следующим образом:

typedef int fmtflags;

**Флаги форматирования и их назначение**

|  |  |
| --- | --- |
| **Флаг** | **Назначение** |
| boolalpha | Значения булева типа вставляются и извлекаются в виде слов true и false |
| Hex | Значения целого типа преобразуются к основанию 16 (как шестнадцатиричные') |
| Dec | Значения целого типа преобразуются к основанию 10 |
| Oct | Значения целого типа преобразуются к основанию 8 (как восьмиричные) |
| Fixed | Числа с плавающей точкой выводятся в формате с фиксированной точкой (т. е., nnn.ddd) |
| Scientific | Числа с плавающей точкой выводятся в научной записи (т.е. n.хххЕуу) |
| showbase | Выводится основание системы счисления в виде префикса к целому числовому значению |
| showpoint | При выводе значений с плавающей точкой выводится десятичная точка и последующие нули |
| showpos | При выводе положительных числовых значений выводится знак плюс |
| uppercase | Заменяет определенные символы нижнего регистра на символы верхнего регистра (символ "е" при выводе чисел в научной нотации на "Е" и символ "х" при выводе 16-ричных чисел на "X") |
| Left | Данные при выводе выравниваются по левому краю поля |
| Right | Данные при выводе выравниваются по правому краю поля |
| internal | Добавляются символы-заполнители между всеми цифрами и знаками числа для заполнения поля вывода |
| skipws | Ведущие символы-заполнители (знаки пробела, табуляции и перевода на новую строку) отбрасываются |
| unitbuf | Выходной буфер очищается после каждой операции вставки в поток |
| adjustfield | = left | right | internal |
| basefield | = dec | oct | hex |
| floatfield | = scientific | fixed |

Флаги left и right взаимно исключают друг друга. Флаги dec, oct и hex также взаимно исключают друг друга.

Прочесть текущие установки флагов позволяет функция-член flags () класса ios. Для этого используется следующий прототип этой функции:

fmtflags flags() const;

Функция flags() имеет и вторую форму, которая может использоваться для установки значений флагов. Для этого используется следующий прототип этой функции:

fmtflags flags(fmtflags fmtfl);

В этом случае битовый шаблон копируется fmtfl в переменную, предназначенную хотя хранения флагов форматирования. Функция возвращает предыдущие значения флагов. Поскольку эта форма функции меняет весь набор флагов, она применяется редко. Вместо нее используется функция-член setf () класса ios, которая позволяет установить значение одного или нескольких флагов. Она имеет следующие прототипы:

fmtflags setf (fmtflags mask);

fmtflags setf (fmtflags fmtfl, fmtflags mask);

Первая функция-член неявно вызывает функцию flags (mask | flags ()) для установки битов, указанных параметром mask, и возвращает предыдущее значения флагов. Второй вариант функции присваивает битам, указанным параметром mask, значения битов параметра fmtfl, а затем возвращает предыдущие значения флагов. Например, следующий вызов функции setf () устанавливает для потока cout флаги hex и uppercase:

cout.setf (ios::hex | ios:: uppercase);

Сбросить установленные флаги можно с помощью функции-члена unsetf () класса ios, имеющей следующий прототип:

void **unsetf(fmtflags mask);**

Она сбрасывает флаги, заданные параметром mask.

Следующий пример демонстрирует некоторые флаги:

//05.02

#include <cmath>

#include <string.h>

#include <iostream>

using namespace std;

int main()

{

setlocale(LC\_ALL, "Rus");

double d = 1.321e9;

int n = 1024;

//Вывести Значения

cout << "d = " << d << '\n';

cout << "n = " << n << '\n' << '\n';

//Изменить флаги и

//вывести значения снова

cout.setf(ios::hex | ios::uppercase);

cout.setf(ios::showpos);

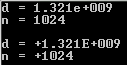
cout << "d = " << d << '\n';

cout << "n = " << n << '\n';

getchar(); getchar();

return 0;

}



При выполнении программа выводит на экран:

d = 1.321е+009

n = 1024

d = +1.321E+009

n = +1024

Система ввода-вывода C++ предусматривает еще один способ форматирования потока. Этот способ основан на использовании манипуляторов ввода-вывода. Список манипуляторов и их назначение приведены в таблице. Манипуляторы ввода-вывода представляют собой просто вид функций-членов класса ios, которые, в отличие от обычных функций-членов, могут располагаться внутри инструкций ввода-вывода. В связи с этим ими пользоваться обычно удобнее.

**Манипуляторы ввода-вывода и их назначение**

|  |  |  |
| --- | --- | --- |
| **Манипулятор** | **Использование** | **Назначение** |
| boolalpha | ввод-вывод | Устанавливает флаг boolalpha |
| dec | ввод-вывод | Устанавливает флаг dec |
| endl | вывод | Вставляет символ новой строки и очищает буфер |
| ends | вывод | Вставляет символ конца строки |
| fixed | вывод | Устанавливает флаг fixed |
| flush | вывод | Очищает буфер потока |
| hex | ввод-вывод | Устанавливает флаг hex |
| internal | вывод | Устанавливает флаг internal |
| left | вывод | Устанавливает флаг left |
| noboolalpha | ввод-вывод | Сбрасывает флаг boolalpha |
| noshowbase | вывод | Сбрасывает флаг noshowbase |
| noshowpoint | вывод | Сбрасывает флаг noshowpoint |
| noshowpos | вывод | Сбрасывает флаг noshowpos |
| noskipws | ввод | Сбрасывает флаг noskipws |
| nounitbuf | вывод | Сбрасывает флаг nounitbuf |
| nouppercase | ввод-вывод | Сбрасывает флаг |
| oct | ввод-вывод | Устанавливает флаг oct |
| Resetiosflags (ios base:: fmtflags mask.) | ввод-вывод | Сбрасывает ios-флаги, в соответствии с mask |
| right | вывод | Устанавливает флаг right |
| scientific | ввод-вывод | Устанавливает флаг  scientific |
| setbase (int base) | ввод-вывод | Задает основание системы счисления для целых |
| setfill (charT c) | ввод-вывод | Устанавливает символ-заполнитель |
| setiosflags (ios base:: \_fmtflags\_mask) | ввод-вывод | Устанавливает ios-флаги, в соответствии с mask |
| setprecision (int n) | ввод-вывод | Устанавливает точность чисел с плавающей точкой |
| setw(int n) | ввод-вывод | Устанавливает минимальную ширину поля |
| showbase | вывод | Устанавливает флаг showbase |
| showpoint | вывод | Устанавливает флаг showpoint |
| showpos | вывод | Устанавливает флаг showpos |
| skipws | ввод | Устанавливает флаг skipws |
| unitbuf | вывод | Устанавливает флаг unitbuf |
| uppercase | вывод | Устанавливает флаг uppercase |
| ws | ввод | Устанавливает пропуск символов-заполнителей |

За исключением setw (). все изменения в потоке, внесенные манипулятором, сохраняются до следующей установки.

При внимательном изучении таблицы можно заметить, что манипуляторы охватывают функциональные возможности, предоставляемые обычными функциями-членами и флагами форматирования. В частности, манипулятор setiosflags () реализует те же функциональные возможности, что и функция-член setf (), а манипулятор resetiosflags () - те же, что и функция-член unsetf ().

Для доступа к манипуляторам с параметрами необходимо включить в программу стандартный заголовочный файл <iomanip>. При использовании манипулятора без параметров скобки за ним не ставятся, так как на самом деле он представляет собой указатель на функцию-член, который передается перегруженному оператору <<.

Рассмотрим пример, демонстрирующий использование манипуляторов.

//05.03

#include <cmath>

#include <iomanip>

#include <string.h>

#include <iostream>

using namespace std;

int main ()

{

setlocale(LC\_ALL, "Rus");

double x;

cout << setprecision(4);

cout << setfill('0');

cout << "x \tsqrt(x) x^2\n\n";

for (x=1.0; x<=6.0; x++ ){

cout << setw(7) << x << " ";

cout << setw(7) << sqrt(x) << " ";

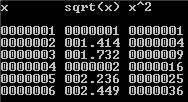
cout << setw(7) << x\*x << "\n";

}

getchar(); getchar();

return 0;

}



Этот пример функционально полностью эквивалентен приведенному ранее, но для управления форматом вывода использует манипуляторы, а не функции форматирования.

Следующий пример продолжает обсуждение форматированного вывода С++, начатое ранее. Первая программа показывает, как распечатать таблицу факториалов, используя переменные типа long double и принятое по умолчанию выравнивание вывода вправо:

// 05.17FACT1.CPP

// форматированный ввод/вывод

// факториал чисел от 1 до 25

//

#include "stdafx.h"

#include <iostream>

#include <math.h>

using namespace std;

void main(void)

{

long double number, factorial;

number = 1.0;

factorial = 1.0;

cout.precision(0); // нет знаков после запятой

cout.setf(ios::fixed); // фиксированный формат

for(int i=0; i< 25; i++)

{

factorial \*= number;

number = number + 1.0;

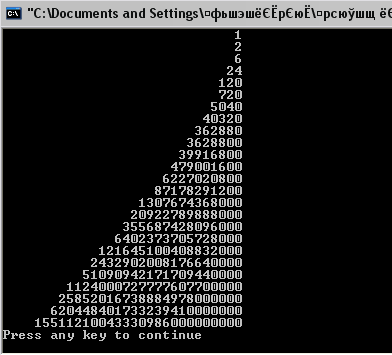
cout.width(30); // ширина 30 символов

cout << factorial << endl;

}

}

В цикле повторяются методы класса precision(), width () и setf (). Программа выводит следующие значения:



В следующем примере показано, как изменить ширину выходного столбца и установленное по умолчанию выравнивание вправо:

// 05.17FACT2.CPP

// форматированный ввод/вывод

//

#include "stdafx.h"

#include <iostream>

#include <math.h>

using namespace std;

void main(void)

{

long double number, factorial;

number = 1.0;

factorial = 1.0;

cout.precision(0); // нет знаков после запятой

cout.setf(ios::left); // выравнивание влево

cout.setf(ios::fixed); // фиксированный формат

for(int i=0; i< 25; i++)

{

factorial \*= number;

number = number + 1.0;

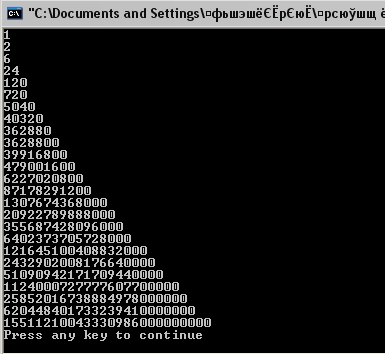
cout.width(30); // ширина 30 символов

cout << factorial << endl;

}

}

Числа, выровненные влево, выглядят следующим образом:



В третьем примере форматирования распечатывается таблица чисел, значения их квадратов и квадратных корней. Эта программа показывает, как легко в С++ выравнивать столбцы, дополнять их пробелами, заполнять пробелы нулями и устанавливать точность.

// 05.17SQRT.CPP

// форматированный ввод/вывод

// вычисление квадрата и квадратного корня для чисе от 1 до 15

//

#include "stdafx.h"

#include <iostream>

#include <math.h>

using namespace std;

void main(void)

{

double number,square,sqroot;

cout << "num\t" << "square\t\t" << "square root\n";

cout << "--------------------------------------\n";

number = 1.0;

cout.setf(ios::fixed); // фиксированный формат

for(int i=1; i< 16; i++)

{

square=number\*number;

sqroot=sqrt(number);

cout.fill('0'); // пробелы заполнить нулями

cout.width(2); // ширина столбца 2

cout.precision(0); // нет знаков после десятичной точки

cout << number << "\t";

cout.width(6);

cout.precision(1); // один знак после запятой

cout << square << "\t\t";

cout.width(8);

cout.precision(6);

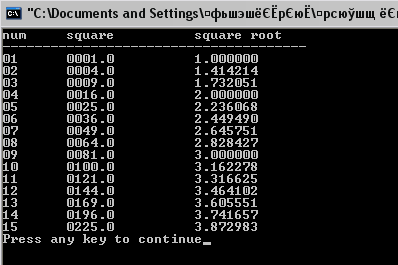
cout << sqroot << endl;

number = number + 1.0;

}

}

Форматированная таблица выглядит так:



## Резидентные в памяти потоки

Язык C++ поддерживает два вида ввода-вывода: файловый ввод-вывод и резидентные в памяти потоки. Файловый ввод-вывод предполагает передачу данных из внешнего устройства или в него. Напротив, резидентные потоки не предполагают работу с внешними устройствами. В этом случае не требуется преобразования кодов и выполнения операций с внешним устройством: осуществляется только форматирование информации. Результат такого форматирования сохраняется в памяти и может быть восстановлен в виде символьной строки. Класс istrstream предназначен для чтения символов из массива в памяти. Он использует private-объект strstreambuf для управления связанным с ним массивом. Поскольку он является производным от basic\_istream, он может использовать функции для форматированного и неформатированного ввода. Класс ostrstream предназначен для записи массива в память. Он также использует private-объект strstreambuf для управления связанным с ним массивом. Поскольку он является производным от basic\_ostream, он также может использовать функции для форматированного и неформатированного вывода. Класс strstream позволяет читать и записывать в массив символов в памяти и является производным от класса basic\_iostrean. Все эти классы определены в заголовочном файле <strstream>. В контексте операций ввода-вывода массивы символов в памяти часто называют резидентными в памяти потоками (строк).

Для работы с резидентным в памяти потоком нужно сначала связать этот поток с некоторой областью памяти (массивом символов). Затем весь ввод-вывод в этот массив можно выполнять с помощью операторов излечения и помещения в поток. Чтобы открыть массив для ввода из него нужно создать объект класса istrstream и, тем самым, связать нужный поток с заданным массивом. Этот класс описывает объект, который управляет извлечением элементов из stream\_buffer класса strstreambuf и объявлен следующим образом:

class **istrstream:** public **istream**

{

public:

explicit **istrstream(**constchar **\*s);**

explicit **istrstream(**char **\*s);**

**istrstream (**constchar **\*s, streamsize n);**

**istrstream(**char **\*s, streamsize n);**

**strstreambuf \*rdbuf()** const;

char **\*str();**

};

Таким образом, для создания объекта класса istrstream можно использовать один из четырех объявленных конструкторов. Конструкторы

istrstream <поток\_ввода>(char **\*s, streamsize n);**

и

istrstream <поток\_ввода> (constchar **\*s, streamsize n);**

используется, когда ввод осуществляется из буфера (которым служит массив), заданного своим указателем s, в том случае, если буфер не заканчивается нулевым символом. В этом случае параметр n задает размер буфера. Если буфер заканчивается нулевым символом можно воспользоваться более простыми формами конструктора:

istrstream <поток\_ввода>(constchar **\*s);**

и

istrstream <поток\_ввода>(char **\*s);**

Для вывода информации в символьный массив нужно создать объект класса ostrstream, который объявлен следующим образом:

class **ostrstream:** public **ostream**

{

public:

**ostrstream();**

**ostrstream (**char **\*s, streamsize n,**

**ios\_base :: opermode mode =ios\_base:: out);**

**strstreambuf \*rdbuf()** const;

void **freeze(**bool **frz =** true);

char **\*str();**

**streamsize pcount()** const;

};

Этот класс описывает объект, который управляет вставкой элементов в streambuffer класса trstreambuf.

Обычно для вывода информации в символьный массив используется конструктор вида:

ostrstream <поток\_вывода>(char **\*s, streamsize n,**

**ios base:: openmode mode =ios base::out);**

Здесь <поток\_вывода> — это поток, который связывается с буфером s, заданным своим указателем (которым является символьный массив), n — размер буфера, mode — задает режим открытия потока вывода. По умолчанию поток открывается для вывода, поэтому этот параметр можно опускать. Функция-член str () возвращает указатель на буфер и "замораживает" массив. При использовании динамических объектов вызов str() делает термический буфер вашей собственностью. Это означает, что в дальнейшем вы должны будете удалить этот буфер или вернуть его в собственность потока ostrstream, вызвав

<поток\_вывода>->rdbuf () ->freeze (0);

Отметим, что перед использованием функции str() нужно явным образом вывести в поток нулевой символ, иначе она возвратит указатель на строку, не ограниченную нулем. Другая функция-член — pcount() — возвращает число записанных в буфер байтов информации.

Чтобы открыть массив для ввода и вывода, создайте объект класса strstream:

strstream <поток\_вв>(char **\*s, streamsize n,**

**ios\_base::openmode mode =**

**ios\_base::in | ios\_base::out);**

Здесь <поток\_вв> — это поток ввода-вывода, который для выполнения этих операций связывается с массивом s (который служит буфером) длиной n. После того, как вы создали нужный объект, вы можете пользоваться всеми ранее описанными функциями ввода-вывода, включая функции ввода-вывода для двоичных файлов и для файлов с произвольным доступом. Рассмотрим пример использования резидентного в памяти потока:

//04.05

#include <strstream>

#include <fstream>

#include <iostream>

using namespace std;

//using std::cout;

//using std::cin;

int main()

{

setlocale(LC\_ALL, "Rus");

char buf[] = "test 9 50.32 \*";

char str[80];

int n;

float f;

char c;

//Открываем резидентный в памяти поток

//для ввода

istrstream istr(buf);

istr >> str >> n >> f >> c;

cout << str << ' ' << n << ' '

<< f << ' ' << c << endl;

getchar(); getchar();

return 0;

}



В этом примере содержимое буфера извлекается в переменные, а затем значения переменных выводятся на экран.

Заметим, что, наряду с перечисленными тремя потоками, для работы с резидентными в памяти потоками могут использоваться три аналогичных им потока istringstream, ostringstream и stringstream. Эти потоки во всем аналогичны рассмотренным, за исключением того, что они используют буфер типа basic\_stringbuf (то есть работают со строками типа string) и требуют для их использования подключения стандартного файла <sstream>.

## Буферы и синхронизация.

Потоки выполняют обмен данными между прикладной программой и файлом через буферы, которые являются объектами классов filebuf и stringbuf соответственно. Эти классы являются производными от абстрактного класса streambuf. Каждый буфер содержит в зависимости от режима доступа один или два символьных массива: один для ввода, а второй для вывода данных. Каждый поток поддерживает внутренний указатель на буфер. Ввод-вывод данных из буфера в файл или в строку называется синхронизацией буфера с внешним устройством ввода-вывода. Синхронизация буфера с внешним устройством может выполняться неявно или явно. Неявно синхронизация выполняется в случае закрытия файла или заполнения буфера. Для явной синхронизации используют манипуляторы flush и endl или функцию sync.

Например, в следующей программе показана синхронизация буфера с входным потоком.

#include<iostream>

usingnamespace **std;**

int **main ()**

{

char **a, b;**

**cout <<** "Enter a word: ";

**a = cin.get();**

**cin.sync();**

**cout <<** "Enter another word: ";

**b = cin.get();**

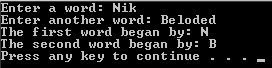
**cout <<** "The first word began by: " **<< a << endl;**

**cout <<** "The second word began by: " **<< b << endl;**

**system(**"pause");

return **0;**

}



## Условные признаки файлов в C++

С каждым потоком связано слово состояния ошибок. При возникновении некоторой ошибки в слове состояния устанавливаются разряды, соответствующие ее типу. В соответствие с принятым соглашением при выводе объектов в поток ostream, имеющий установленные разряды ошибок, операции записи игнорируются и не изменяют состояние потока. Объекты библиотеки ostream имеют набор предопределенных условных признаков, отображающих текущее состояние потока. В следующей таблице перечислены семь методов классов, имеющихся в распоряжении:

|  |  |
| --- | --- |
| **Методы функций** | **Действие** |
| eof() | Возвращает ненулевое значение при достижении конца файла |
| fail() | Возвращает ненулевое значение, если операция завершена с ошибкой |
| bad() | Возвращает ненулевое значение, если возникла ошибка |
| good() | Возвращает ненулевое значение, если отсутствуют установленные разряды |
| rdstate() | Возвращает текущее состояние потока |
| сleаг() | Устанавливает состояние потока (int=0) |

Эти методы классов можно использовать при реализации различных алгоритмов для определения конкретных состояний ввода/вывода и для большей читаемости программы:

ifstream pfsinfile("sample.dat", os::in);

if(pfsinfile.eof())

**pfsinfile.clear();** // устанавливает состояние pfsinfile в 0

if(pfsinfile.fail())

**cerr <<** ">>>>sample.dat creation error <<<<"; //ошибка создания sample.dat

if **(pfsinfile.good())**

**cin >> my\_object;**

if(!pfsinfile) // усеченный файл

**cout <<** ">>>> sample.dat creation error <<<<"; //ошибка создания sample.dat

Этот материал служит введением в концепцию ввода/вывода в C++. Для дальнейшего понимания различных возможностей форматирования необходимо познакомиться с классами C++ и разными способами перегрузки.

## Опрос и установка состояния потока

Класс ios поддерживает информацию о состоянии потока после каждой операции ввода-вывода. Текущее состояние потока хранится в объекте типа iostate, который объявлен следующим образом:

typedefint **iostate;**

Состояния потока являются элементами перечислимого типа io\_state, который может иметь значения, представленные в таблице.

**Состояния потока и их значения**

|  |  |
| --- | --- |
| **Состояние** | **Значение** |
| Goodbit | Ошибок нет |
| Eofbit | Достигнут конец файла |
| Failbit | Имеет место ошибка форматирования или преобразования |
| Badbit | Имеет место серьёзная ошибка |

Для опроса и установки состояния потока можно использовать функции-члены класса ios. Имеется два способа получения информации о состоянии операции ввода-вывода. Во-первых, можно вызвать функцию rdstate(), имеющую следующий прототип:

iostate rdstate() const;

Функция возвращает состояние операции ввода-вывода. Во-вторых, можно воспользоваться одной из следующих функций-членов:

bool **good()** const;

bool **eof()** const;

bool **fail()** const;

bool **bad()** const;

Каждая из этих функций возвращает true, если установлен соответствующий бит состояния (точнее, функция fail() возвращает true, если установлен бит failbit или badbit).

Если прочитано состояние, которое сигнализирует об ошибке, его можно сбросить с помощью функции clear ():

void **clear(iostate state = ios::goodbit);**

Установить нужное состояние можно с помощью функции setstate():

void **setstate(iostate state);**

Кроме перечисленных функций, класс ios содержит функцию приведения типа

operatorvoid\*() const;

(она возвращает NULL, если установлен бит badbit) и перегруженный оператор логического отрицания

booloperator!() const;

(он возвращает true, если установлен бит badbit). Это позволяет сравнивать выражения, в которые входит поток или его отрицание с нулем, то есть писать выражения вида:

while(!strmOb.eof())

{

//…

}

Следующий пример иллюстрирует получение информации о состоянии ввода-вывода.

//04.03

#include <fstream>

#include <iostream>

using namespace std;

//using std::cout;

//using std::cin;

int main(int argc, char\* argv[])

{

char c;

if (argc > 1){

ifstream ifs(argv[1] );

if (!ifs){

cout << "Файл не открыт\n";

return 1;

}

while (!ifs.eof()){

ifs.get(c);

//Контроль состояния потока

if (ifs.fail()){

cout << "Ошибка!";

break;

}

cout << c;

}

ifs.close();

}

return 0;

}

В этом примере осуществляется ввод символов из файла, заданного в командной строке при запуске программы. Если при извлечении символов встречается ошибка, чтение прекращается и выводится сообщение об этом.

## Ошибки потоков

По умолчанию поток ввода-вывода не выбрасывает никаких исключений. Вы должны сами активизировать исключение. Для этой цели поток содержит маску исключений. Каждый из флагов этой маски соответствует одному из флагов ошибки (состояния потока). Ключевую роль в выбрасывании исключений играют две функции-члена класса ios:

iostate exceptions() const;

Эта функция возвращает маску, которая определяет, какие флаги состояния, установленные в маске флагов состояния, будут вызывать выбрасывание исключения. Она неявно использует вызов функции rdstate (). Другая функция

void **exceptions(iostate except);**

устанавливает маску исключений потока в значение, определяемое переменной except, а затем вызывает clear (rdstate ()) для сброса, возможно, установленных битов маски состояния потока. Например, если флаг badbit установлен в маске исключений, исключение будет выброшено, если этот флаг будет установлен в маске состояния потока. Следующий пример демонстрирует, как можно активизировать исключение в потоке ввода:

try

{

**ifs.exceptions (ios::badbit | ios::failbit);**

**in >> x;**

//…

}

catch(ios::failure& e)

{

**cerr << e.what() << endl;**

throw;

}

При вызове функции exceptions() вы указываете, какие флаги состояния потока будут вызывать выбрасывание исключения. Объекты, выброшенные потоковыми операциями ввода-вывода, являются производными от исключения типа ios :: failure. Заметим, что любое изменение состояния потока будет приводить к выбрасыванию исключения. Это происходит потому, что функции setstate () и exception () приводят к выбрасыванию исключения, если в маске исключений установлен соответствующий бит. Следовательно, оператор catch, приведенньй выше, будет ловить все исключения, связанные с потоками. Рекомендуется активизировать исключение по флагу badbit и подавить исключения по флагам eofbit и faiibit, поскольку они, вообще говоря, не представляют исключительные состояния.

## Часто применяемые функции

Помимо уже описанных функций, библиотека ввода-вывода C++ содержит широкий набор различных функций. Здесь мы приведем лишь некоторые, наиболее часто употребляемые из них. Большинство этих функций используется для неформатируемого ввода-вывода.

Для извлечения символа из потока можно использовать функцию-член get() потока istream. Она имеет много прототипов, однако чаще других используются следующие два:

int\_type get();

istream\_type& get(char\_type& с);

Тип int\_type определен как целочисленный. Приведем пример использования функции get ():

//05.05

#include <cmath>

#include <iomanip>

#include <string.h>

#include <fstream>

#include <iostream>

using namespace std;

//using std::cout;

//using std::cin;

int main()

{

setlocale(LC\_ALL, "Rus");

char ch;

cout << "Введите число. "

<< "Для завершения ввода нажмите <ENTER>: ";

while (cin.get(ch)){

// Проверка на код клавиши <ЕNTER>

if (ch == '\n')

break;

}

getchar(); getchar();

return 0;

}



Для вставки символа в поток вывода используется функция put(), которая имеет следующий прототип:

ostream& put(char **ch);**

Функция get() может также использоваться для чтения строки символов. В этом случае используются ее варианты, определяемые следующими прототипами:

istream\_type& get(char\_type\* str, streamsxze len, **char\_type delim);**

istream\_type &get(char\_type\* str, streamsize len);

Эта функция извлекает из входного потока символы в буфер str, пока не встретится символ-разграничитель delim (по умолчанию - перевод строки) или не будет прочитано (lеn-1) символов, или не будет прочитан конец файла. Символ-разграничитель не извлекается из входного потока.

Ввиду того, что функция get() не извлекает из входного потока символ-разграничитель, она используется редко. Гораздо чаще используется функция getline (), которая не извлекает из входного потока символ-разграничитель, но не помещает его в буфер. Она имеет следующие прототипы:

istream\_type& getline (char\_type\* str,streamsize len, char\_type delim);

istream\_type& getline (char\_type\* str,streamsize len);

Здесь параметры имеют то же назначение, что и в функции get().

Функция gcount () возвращает число символов, извлеченных из потока последней операцией неформатируемого ввода (т.е. функцией get(), getline() или read()). Она имеет следующий прототип:

streamsize gcount() const;

Рассмотрим пример, в котором используются две последние функции:

//05.06

#include <cmath>

#include <iomanip>

#include <string.h>

#include <fstream>

#include <iostream>

using namespace std;

//using std::cout;

//using std::cin;

int main()

{

setlocale(LC\_ALL, "Rus");

char \*name;

int len = 100;

int count = 0;

name = new char[len];

cout << "Введите свое имя: ";

cin.getline(name, len);

count = cin.gcount();

//Уменьшаем значение счетчика на 1, т.к.

//getline() не помещает

//разграничитель в буфер

cout << "\nЧисло прочитанных символов: "

<< count - 1;

getchar(); getchar();

return 0;

}



Для того, чтобы пропустить при вводе несколько символов, используется функция ignore():

istream\_type& ignore(streamsize n=1,

**int\_type delim=traits::eof());**

Эта функция игнорирует вплоть до n символов во входном потоке. Пропуск символов прекращается, если она встречает символ-разграничитель, которым по умолчанию является символ конца файла. Символ-разграничитель извлекается из входного потока.

Функция peek (), имеющая следующий прототип:

int\_type peek();

позволяет "заглянуть" во входной поток и узнать следующий вводимый символ. При этом сам символ из потока не извлекается. С помощью функции putback (), имеющей прототип:

istream\_type& putback(char\_type ch);

можно вернуть символ ch в поток ввода.

При выполнении вывода данные не сразу записываются в файл, а временно хранятся в связанном с потоком буфере, пока он не заполнится. Функция flush() позволяет вызвать принудительную запись в файл до заполнения буфера. Эта функция имеет следующий прототип:

ostream\_type& flush();

Она неявно используется манипулятором endl. Этот манипулятор вставляет в поток символ перевода строки и очищает буфер. Таким образом, инструкция

cout << endl;

эквивалентна следующим:

cout << '\n'; cout.flush ();

Функция rdbuf() позволяет получить указатель на связанный с потоком буфер. Эта функция имеет следующий прототип:

streambuf\_type\* rdbuf() const;

Наконец, функция setbuf () позволяет связать с потоком другой буфер. Она имеет такой прототип:

void **setbuf (**char\* buf, streamsize n);

Здесь buf указатель на другой буфер длины n.

## Двоичные файлы.

Когда вы открываете файл в текстовом режиме, происходит следующее:

* при вводе каждая пара символов '\r'+'\n' (возврат каретки + перевод строки) преобразуется в символ перевода строки ('\n');
* при выводе каждый символ перевода строки ('\n') преобразуется в пару ' \r' + ' \n' (возврат каретки + перевод строки).

Это не всегда удобно. Если вы собираетесь использовать выводимый файл для последующего ввода в (возможно, другую) программу, лишние байты информации ни к чему. С этой целью система ввода-вывода предоставляет вам возможность осуществления неформатируемого ввода-вывода, то есть записи и чтения двоичной информации (иногда говорят - сырых данных). Для осуществления ввода-вывода в двоичном режиме, включите флаг ios :: binary в параметр open\_mode, передаваемый конструктору потока или функции open(). Чтение двоичной информации из файла осуществляется функцией read(), которая имеет следующий прототип:

istream\_typr&

read (char\_type\* s, streamsize n);

Здесь параметр s задаст буфер для считывания данных, а параметр n—число читаемых символов.

Запись двоичных данных осуществляет функция-член write ():

ostream\_type&

write(const **char\_type\* s, streamsize n);**

Эта функция получает n символов из буфера, адрес которого задан параметром s, и вставляет их в поток вывода. Рассмотрим пример:

//05.04

#include <cmath>

#include <iomanip>

#include <string.h>

#include <fstream>

#include <iostream>

using namespace std;

int main()

{

setlocale(LC\_ALL, "Rus");

int x = 255;

char str[80] ="binary input-output testing";

//"Тестирование двоичного ввода-вывода."

//Открываем файл для вывода в

//двоичном режиме

ofstream ofs("Test.dat");

if (!ofs){

cout << "Файл не открыт.";

return 1;

}

ofs.write ((char\*) &x, sizeof(int));

ofs.write ((char\*) &str, sizeof(str));

//Открываем файл для вывода в

//двоичном режиме

ifstream ifs("Test.dat");

if (!ifs){

cout << "Файл не открыт.";

return 1;

}

ifs.read((char\*)&x, sizeof(int));

ifs.read((char\*)&str, sizeof(str));

cout << x << '\n' << str << '\n';

getchar(); getchar();

return 0;

}



В большинстве рассмотренных до сих пор программ использовались стандартные текстовые файлы, или потоки — как их более правильно называть. Это не удивительно, поскольку потоки первоначально разрабатывались для текстов и текстовый режим применяется по умолчанию при вводе/выводе.

Стандартные текстовые файлы или потоки состоят из последовательности символов, включающей символы возврата каретки и перевода строки. В текстовом режиме не требуется, чтобы отдельные символы оставались неизменными — такими, какими они записывались или читались из файла. В некоторых приложениях из-за этого могут возникнуть проблемы. Например, ASCII-КОД символа перевода строки равен десятичному числу 10. Однако, он может быть также записан как 8-разрядное шестнадцатеричное число 0А. В программах С/С++ это значение рассматривается как отдельная символьная константа '\n'.

Оказывается, что в системе MSDOS новая строка физически представляется двумя символами: возврат каретки (десятичное 13) и перевод строки (десятичное 10). Обычно это не представляет проблемы, поскольку при вводе программа автоматически преобразует эту пару символов в один символ новой строки и выполняет обратное преобразование при выводе. Проблема в том, что символ новой строки занимает один байт, а пара символов CR/LF занимает два байта памяти.

Двоичные файлы или потоки содержат последовательность байтов, полностью соответствующих их представлению на внешнем устройстве (диске, ленте или терминале). В двоичном файле никаких преобразований символов не выполняется, и поэтому число прочитанных или записанных байтов совпадает с числом байтов, располагающихся в файле на внешнем устройстве.

Если в программе необходимо считывать исполняемый файл, то это нужно делать в двоичном режиме. Двоичные файлы также используются при считывании и записи чисто информационных файлов, например баз данных. Это гарантирует, что не выполняется никаких преобразований данных, за исключением тех, которые осуществляются непосредственно в приложении.

Следующая программа идентична программе 17OSTRM.СРР, описанной ранее; отличие заключается в том, что выходной файл открывается не в текстовом режиме, а в режиме ios::binary:

// 05.17BINARY.CPP

// применение класса ofstream

// вывод двойчного файла

// допустимые методы класса ofstream

// ofstream::open ofstream::rdbuf

// допустимые методы класса ostream

// ostream::flush ostream::ostream

// ostream::put ostream::seekp

// ostream::tellp ostream::write

//

#include "stdafx.h"

#include <iostream>

#include <fstream>

#include <math.h>

#include <strstream>

using namespace std;

#define iSTRING\_MAX 40

void main(void)

{

int i = 0;

long ltellp;

char pszString[iSTRING\_MAX] = "Simple test string\n"; /\* 6+1+4+1+6=18+1(‘\n’)=19 \*/

//файл открывается в двоичном режиме

ofstream ofMyOutputStream("myostrm.out", ios::out|ios::binary);

//посимвольный вывод строки

//строка '\n' НЕ преобразуется в два символа

while(pszString[i] != '\0')

{

ofMyOutputStream.put(pszString[i]);

ltellp = ofMyOutputStream.tellp();

cout << "\ntellp value: " << ltellp;

i++;

}

//выод строки целиком при помощи метода класса

ltellp = ofMyOutputStream.tellp();

cout << "\ntellp's value before writing 2nd string: " << ltellp;

ofMyOutputStream.write(pszString, strlen(pszString));

ltellp = ofMyOutputStream.tellp();

cout << "\ntellp's updated value: " << ltellp << '\n';

ofMyOutputStream.close();

}

Сокращенная распечатка результатов показывает однозначное соответствие между файлом и внутренним представлением данных:

tellp value: 1

tellp value: 2

tellp value: 3

tellp value: 4

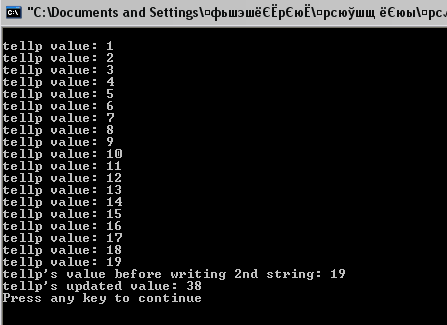
tellp value: 17

tellp value: 18

tellp value: 19

tellp's value before writing 2nd string: 19

tellp's updated value: 38



Строка pszString, занимающая 19 символов плюс null-символ '\0' в конце строки, выводится в точном соответствии с тем, как она хранится, без добавления символа конца строки '\0'. Именно поэтому функция tellp() выводит число, кратное 19, по окончанию вывода каждой строки.

## Объединение программ на С и С++. Использование спецификатора extern "С".

Ранее уже рассказывалось о том, как ключевое слово extern указывает на внешнюю связь переменной или функции. Это означает, что данная переменная или функция описаны в некотором другом исходном файле или в том же файле, но ниже.

Однако, в С/С++ можно использовать extern вместе со строкой. Эта строка указывает на то, что для описываемых идентификаторов используются соглашения о компоновке другого языка. По умолчанию для программ С++ задается строка "С++".

По умолчанию в С++ функции можно перегружать. При этом компилятор С++ каждой функции присваивает новое имя. Эту операцию можно отменить, если указать перед описанием функции extern "С". Это необходимо для того, чтобы из программы, написанной на С++, были доступны функции и данные С. Конечно же, это нужно делать только для функций с одинаковыми именами. Если не отменить эту опцию, то компоновщик обнаружит несколько глобальных функций с одним и тем же именем. В настоящее время компилятор Visual С/С++ поддерживает единственный спецификатор другого языка — "С". Синтаксис записи extern "С" выглядит следующим образом:

extern"С" **тип\_ функции имя\_ функции(тип\_параметра (тип\_параметров) параметр (параметры));**

Следующий оператор показывает, как запись extern "С" используется вместе с прототипом отдельной функции:

extern"С"int **fprintf(FILE \*stream,** char **\*format, ...);**

Для модификации группы прототипов функций нужны фигурные скобки {}:

extern"С" **{**

**….**

}

В следующем фрагменте кода модифицируются прототипы функций getc() и рutc():

extern"С" **{**

int **getc(FILE \*stream);**

int **putc(**int **с, FILE \*stream);**

}

Ниже приведена программа, показывающая, как использовать запись extern "С":

// 05.17CLINK.CPP

// связь С++ с библиотеками С

//

#include "stdafx.h"

#include <iostream>

#include <fstream>

#include <math.h>

#include <strstream>

using namespace std;

#define iMAX 9

extern "C" int imycompare(const void \*pi1, const void \*pi2);

void main(void)

{

int iarray[iMAX]={1, 9, 2, 8, 3, 7, 4, 6, 5};

for(int i=0; i < iMAX; i++)

cout << iarray[i] << " ";

cout << endl;

qsort(iarray,iMAX, sizeof(int), imycompare);

for(i = 0; i < iMAX; i++)

cout << iarray[i] << " ";

cout << endl;

}

extern "C" int imycompare(const void \*pi1, const void \*pi2)

{

return(\*(int \*)pi1 - \*(int \*)pi2);

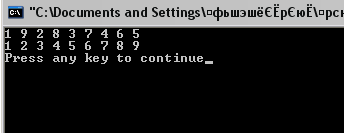
}

Результат работы программы

1 9 2 8 3 7 4 6 5

1 2 3 4 5 6 7 8 9

Press any key to continue



Во всех include-файлах Visual С используется спецификация extern "С". Благодаря этому, в программах, написанных на С++, можно вызывать библиотечные функции времени исполнения языка С. Вместо того, чтобы для каждого определения повторять спецификатор extern "С", во всех заголовочных файлах С используется пара следующих условных операторов, внутри которых располагаются описания:

// Операторы, расположенные в начале заголовочного файла

#ifdef **\_\_cplusplus**

extern"С" **{**

#endif

// Операторы, расположенные в конце заголовочного файла

#ifdef **\_\_cplusplus**

}

#endif

При компиляции программы на С++ транслятор автоматически определяет имя \_\_cplusplus, которое, в свою очередь, управляет видимостью спецификатора extern "С" и фигурных скобок, обрамляющих описания.

## Манипуляторы.

**Манипулятором** называется глобальная функция, которая изменяет поток. Манипуляторы предназначены для включения в последовательность операторов ввода или вывода и могут изменять состояние потока, управляя его флагами. Манипуляторы подразделяются на две группы: простые манипуляторы и параметризованные манипуляторы.

Простые манипуляторы не требуют аргументов. Ниже перечислены простые манипуляторы:

endl помещает в выходной поток символ ‘\n’ и вызывает манипулятор flush,

ends помещает в поток символ ‘\0’,

flush освобождает поток,

dec устанавливает 10 c/c,

hex устанавливает 16 c/c,

oct устанавливает 8 c/c,

ws игнорирует при вводе ведущие пробелы.

Манипуляторы с аргументами называются параметризованными. Ниже перечислены параметризованные манипуляторы, которые объявлены в заголовочном файле <iomanip>:

resetiosflags сбрасывает флаги форматирование,

setiosflags устанавливает флаги форматирования,

setbase устанавливает систему счисления,

setfill устанавливает символ заполнитель,

setprecision устанавливает точность для плавающих чисел,

setw устанавливает ширину поля для символа заполнителя.

Можно написать свой манипулятор. Например, следующая функция является манипулятором, который выполняет переход на новый абзац.

//#include <iostream.h>

#include<iostream>

usingnamespace **std;**

ostream& tabl(ostream& out)

{

**out <<** '\n' **<<** '\t';

return **out;**

}

int **main()**

{

**cout <<** "This is a test for the manipulator tabl"

**<< tabl**

**<<** "The end of the test" **<< endl;**

**system(**"pause");

return **0;**

}



Ранее описывалась концепция манипуляторов потоков. Манипуляторы используются вместе с операциями вставки << и выделения >> аналогично выходным данным или входным переменным. Однако, в соответствии со своим названием манипуляторы могут выполнять различные действия над входным и выходным потоками.

В некоторых примерах программ использовались встроенные манипуляторы dec, hex, oct, setw и setprecision. Теперь вы познакомитесь с тем, как создавать пользовательские манипуляторы. По мере знакомства с синтаксическими конструкциями, необходимыми для создания специальных манипуляторов, примеры усложняются; сначала рассматриваются простейшие манипуляторы без параметров, а затем — с параметрами.

### Манипуляторы без параметров.

Пользовательский манипулятор можно создать в любой момент, когда необходимо несколько раз выводить одну и ту же последовательность символов в выходной поток. Например: в приложении необходимо отмечать для пользователя важные данные. Возможно даже выдавать звуковой сигнал для привлечения внимания оператора к экрану. Без специальных манипуляторов операторы вывода будут выглядеть так:

**cout <<** '\а' **<<** "\n\n\t\tImportant data: " **<< fcritical\_mass << endl;** // Важные данные

Каждый раз для привлечения внимания пользователя нужно повторять символ звукового сигнала '\а' и строку "... Important data: ". Более простой вариант — определить манипулятор, названный beep, который будет автоматически подставлять нужную последовательность символов. Этот манипулятор упростит чтение оператора вывода:

**cout << beep << fcritical\_mass << endl;**

В следующей программе показано, как описать и использовать функцию beep():

// 05.17BEEP.CPP

// манипультор без параметров

//

#include "stdafx.h"

#include <iostream>

#include <fstream>

#include <math.h>

#include <strstream>

using namespace std;

ostream& beep(ostream& os)

{

return os << '\a' << "\n\n\t\t\tImportant data: ";

}

void main(void)

{

double fcritical\_mass = 12459876.12;

cout << beep << fcritical\_mass << endl;

}

В глобально описанной функции beep() используется формальный параметр ostream&; он же является возвращаемым значением. При работе эта функция автоматически связывается с потоковой операцией <<. Для того, чтобы принимать подобную функцию, операция вставки в поток перегружается; для этого используется следующий формат встраиваемой функции:

inline **ostream& ostream::**operator **<< (ostream& (\*f)(ostream&)) {**

**(\*f)(\***this);

return **\***this;

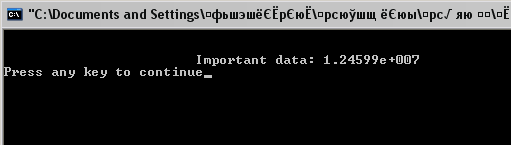
}

Встраиваемая функция связывает операцию << с пользовательским манипулятором; для этого она принимает указатель на переданный функции тип ostream& и затем возвращает его. Это полностью совпадает с прототипом функции beep(). После этого, когда совместно используются операция << и функция beep(), компилятор выполняет разыменование перегруженной операции, находит функцию beep(), а затем выполняет ее. Перегруженная операция возвращает ссылку на исходный поток ostream. Благодаря этому, можно комбинировать манипуляторы, строки и другие данные и операции <<.

Результат работы программы:

Important data: 1.24599e+007

Press any key to continue



### Манипуляторы с одним параметром.

В библиотеке классов Microsoft iostream, имеющей прототипы в файле iomanip.h, описан набор специальных макросов, предназначенных для создания параметризованных макроопределений. Простейший параметризованный макрос может принимать один аргумент типа int или long.

В следующем листинге описан прототип для такого манипулятора (fс). Этот пример показывает, как создать пользовательский манипулятор с одним параметром:

// 05.17MANIP.CPP VC6.0

// манипультор с одним параметром

#include <iostream.h>

#include <iomanip.h>

#include <string.h>

#define iSCREEN\_WIDTH 80

ostream& fc(ostream& os, int istring\_width)

{

os << '\n';

for(int i = 0; i < ((iSCREEN\_WIDTH - istring\_width)/2); i++)

os << ' ';

return (os);

}

OMANIP(int) center(int istring\_width)

{

return OMANIP(int) (fc, istring\_width);

}

void main(void)

{

char \*psz="This is auto-centered text!"; // автоцентрирующийся текст

cout << center(strlen(psz)) << psz << endl;

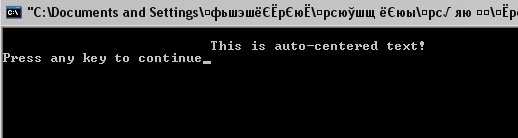
}

Пользовательский параметризованный манипулятор center принимает одно значение, strlen(psz), представляющее собой длину строки. В файле iomanip.h описывается макрос OMANIP(int), расширяемый в класс \_\_OMANIP\_int. Описание этого класса включает конструктор и перегруженную операцию вставки в поток ostream. Когда функция center() включается в поток, она вызывает конструктор, который создает и возвращает объект \_\_OMANIP\_int. Затем конструктор объектов вызывает функцию fc().

Результат выполнения программы:

This is auto-centered text!

Press any key to continue



### Манипуляторы с несколькими параметрами.

Следующий пример может показаться вам знакомым. Действительно, эта программа (17SQRT.СРР) описана выше и служит для демонстрации форматированного вывода цифр. Однако, программа переписана, и теперь в ней используется пользовательский манипулятор с двумя параметрами.

Во-первых, в программе описана простая структура, хранящая фактические параметры манипулятора форматирования:

struct **stwidth\_precision**

{

int **iwidth;**

int **iprecision;**

};

При создании манипуляторов, получающих аргументы, отличные от int и long, необходимо использовать макрос IOMANIPdeclare. Это макроопределение объявляет классы для нового пользовательского типа данных. Описание манипулятора format начинается с макроса OMANIP:

OMANIP(stwidth\_precision) format(int **iwidth,** int **iprecision)**

{

**stwidth\_precision stWidth\_Precision;**

**stWidth\_Precision.iwidth = iwidth;**

**stWidth\_Precision.iprecision = iprecision;**

return **OMANIP(stwidth\_precision) (ff, stWidth\_Precision);**

}

В этом примере пользовательскому манипулятору передаются два целочисленных аргумента, iwidth и iprecision. Первый из них определяет количество пробелов, используемых манипулятором format, а второй — задает число позиций после десятичной точки. После инициализации структуры stWidth\_Precision манипулятор format вызывает конструктор, создающий и возвращающий объект \_\_OMANIP. Затем конструктор объектов вызывает функцию ff(), которая устанавливает заданные параметры:

static **ostream& ff(ostream& os, stwidth\_precision stWidth\_Precision)**

{

**os.width(stWidth\_Precision.iwidth);**

**os.precision(stWidth\_Precision.iprecision);**

**os.setf(ios::fixed);**

return **os;**

}

Законченная программа приводится ниже. Для сравнения в листинге оставлен весь код, замененный манипулятором format. Обратите внимание на то, как использование этого манипулятора упрощает каждый оператор вывода.

// 05.17MANIP2.CPP VC6.0

// манипультор с двумя параметрами

#include <iostream.h>

#include <iomanip.h>

#include <math.h>

struct stwidth\_precision

{

int iwidth;

int iprecision;

};

IOMANIPdeclare(stwidth\_precision);

static ostream& ff(ostream& os, stwidth\_precision stWidth\_Precision)

{

os.width(stWidth\_Precision.iwidth);

os.precision(stWidth\_Precision.iprecision);

os.setf(ios::fixed);

return os;

}

OMANIP(stwidth\_precision) format(int iwidth, int iprecision)

{

stwidth\_precision stWidth\_Precision;

stWidth\_Precision.iwidth = iwidth;

stWidth\_Precision.iprecision = iprecision;

return OMANIP(stwidth\_precision) (ff, stWidth\_Precision);

}

void main(void)

{

double number,square,sqroot;

cout << "num\t" << "square\t\t" << "square root\n";

cout << "--------------------------------------\n";

number = 1.0;

//cout.setf(ios::ffixed);

for(int i = 1;i < 16; i++)

{

square = number \* number;

sqroot = sqrt(number);

cout.fill('0');

//cout.width(2);

//cout.precision(0);

cout << format(2, 0) << number << "\t";

//cout.width(6);

//cout.precision(1);

cout << format(6, 1) << square << "\t\t";

//cout.width(8);

//cout.precision(6);

cout << format(8, 6) << sqroot << endl;

number += 1.0;

}

}

Результат работы программы:

num square square root

--------------------------------------

01 0001.0 1.000000

02 0004.0 1.414214

03 0009.0 1.732051

04 0016.0 2.000000

05 0025.0 2.236068

06 0036.0 2.449490

07 0049.0 2.645751

08 0064.0 2.828427

09 0081.0 3.000000

10 0100.0 3.162278

11 0121.0 3.316625

12 0144.0 3.464102

13 0169.0 3.605551

14 0196.0 3.741657

15 0225.0 3.872983

Press any key to continue

