Классы ввода-вывода на С++

Классы ввода-вывода являются важнейшими классами стандартной библиотеки С++, - программа, которая не вводит и не выводит данные, вряд ли принесет много пользы.

Классы ввода-вывода стандартной библиотеки С++ не ограничиваются операциями с файлами, экраном и клавиатурой. Они создают расширяемую архитектуру для форматирования произвольных данных и работы с произвольными *внешними представлениями*.

Библиотека IOStream (как называется совокупность классов ввода-вывода) единственная часть стандартной библиотеки С++, которая широко использовалась до стандартизации С++.

В ранние поставки систем С++ включались классы, разработанные в AT&T и ставшие фактическим стандартом ввода-вывода. Хотя в дальнейшем эти классы адаптировались для интеграции со стандартной библиотекой С++ и выполнения некоторых новых функций, базовые принципы, заложенные в классы библиотеки IOStream, остались неизменными.

Классы ввода-вывода на C++ - новое

Основные принципы потоковых классов ввода-вывода остались неизменными, но были добавлены некоторые важные новшества, расширяющие возможности настройки и адаптации. В частности:

- Классы библиотеки IOStream преобразованы в шаблоны, что позволяет поддерживать разные представления символов.
- Потоковые классы для символьных массивов типа **char*** были заменены классами, использующими строковые типы стандартной библиотеки C++. Старые классы по-прежнему поддерживаются для обеспечения совместимости, но они считаются устаревшими!
- Обработка ошибок была интегрирована с обработкой исключений.

Все символические имена библиотеки IOStream, как и в остальных компонентах стандартной библиотеки C++, объявляются в пространстве имен **std**.

Потоки

В C++ операции ввода-вывода выполняются при помощи *потоков данных*.

Согласно принципам объектно-ориентированного программирования, поток данных представляет собой объект, свойства которого определяются классом.

Вывод интерпретируется как запись данных в поток, а ввод - как чтение данных из потока.

Для стандартных каналов ввода-вывода существуют стандартные глобальные объекты.

Специализированные разновидности ввода-вывода (ввод, вывод, операции с файлами) представлены в библиотеке разными классами. Среди потоковых классов центральное место занимают следующие:

- класс istream - входной поток, используемый для чтения данных;
- класс ostream - выходной поток, используемый для записи данных.

Оба класса представляют собой специализации шаблонов **basic_istream<>** и **basic ostream<>** для типа символов **char**. Библиотека IOStream не зависит от конкретного типа символов - для большинства классов библиотеки этот тип передается в аргументе шаблона.

Потоки - преимущества

Одним из аргументов в пользу потоков является простота использования: нет необходимости задумываться о форматировании – каждый объект сам знает, как он должен выглядеть. Это избавляет программиста от одного из основных источников ошибок.

Другим преимуществом является возможность перегрузки операторов и функций вставки и извлечения потоковых данных при работе с собственными классами. Это позволяет работать с ними как со стандартными типами, что опять же делает программирование проще, не говоря уже о сокращении возможных ошибок и удобстве работы.

Но нужны ли потоки в условиях работы в графической среде, подобной Windows?

Оказывается, что да. Потому что это лучший способ записывать данные в файл, лучший способ организации данных в памяти для последующего использования при вводе-выводе текста в окошках и других элементах графического интерфейса среды.

В библиотеке IOStream определено несколько глобальных объектов типов **istream** и **ostream**. Эти объекты соответствуют стандартным каналам ввода-вывода.

- Объект cin (класс **istream**) представляет стандартный входной канал, используемый для ввода пользовательских данных. Он соответствует потоку данных **stdin** в языке С. Обычно операционная система связывает этот канал с клавиатурой.
- Объект cout (класс **ostream**) представляет стандартный выходной канал, предназначенный для вывода результатов работы программы. Он соответствует потоку данных **stdout** в языке С. Обычно операционная система связывает этот канал с монитором.
- Объект cerr (класс **ostream**) представляет стандартный канал, предназна-ченный для вывода всевозможных сообщений об ошибках. Он соответствует потоку данных **stderr** в языке С. Обычно операционная система также связы-вает этот канал с монитором. По умолчанию вывод в cerr не буферизуется.
- Объект clog (класс **ostream**) представляет стандартный канал для регистрации данных и не имеет аналогов в языке С. По умолчанию этот поток данных связывается с тем же приемником, что и cerr, но вывод в clog буферизуется.

Потоковые операторы

Операторы сдвига >> и << были перегружены для потоковых классов и означают соответственно ввод и вывод. При помощи этих операторов можно выполнять каскадные операции ввода-вывода. Например, следующий цикл при каждой итерации читает из стандартного входного потока данных два целых числа (пока вводятся только целые числа) и записывает их в стандартный выходной поток данных:

```
int a, b:
// Пока операции ввода a и b проходят успешно
while (std::cin >> a >> b {
// Вывод a и b
std::cout << "a: " << a << "b: " << b << std::endl;
```

Манипуляторы

В конце большинства команд потокового ввода-вывода записывается так называемый манипулятор:

```
std::cout << std::endl
```

Манипуляторы - специальные объекты, предназначенные для управления потоком данных. Часто манипуляторы изменяют только режим интерпретации ввода или форматирования вывода (например, манипуляторы выбора системы счисления **dec**, **hex** и **oct**). Это означает, что манипуляторы потока данных *ostream* не всегда создают выходные данные, а манипуляторы потока данных *istream* не всегда интерпретируют ввод. Однако некоторые манипуляторы выполняют непосредственные действия - очистку выходного буфера, переключение в режим игнорирования пропусков при вводе и т. д.

Манипулятор **endl** обозначает -«конец строки», а при его выводе выполняются

две операции.

1. Отправка признака новой строки (то есть символа **\n**) в выходной поток данных.

2. Очистка выходного буфера (принудительный вывод всех буферизованных данных методом «**flush**».

Манипуляторы - 2

В классах потокового ввода-вывода *istream* и *ostream* определен (кроме рассмотренного **endl**) следующий набор манипуляторов:

Манипулятор	*Класс*	*Описание*
flush	basic_ostream	Принудительный вывод выходного буфера на устройство
ends	basic_ostream	Запись символа завершения строки в буфер
ws	basic_istream	Чтение с игнорированием пропусков

Кроме перечисленных существуют и другие, в частности используемые для проведения форматирования потока

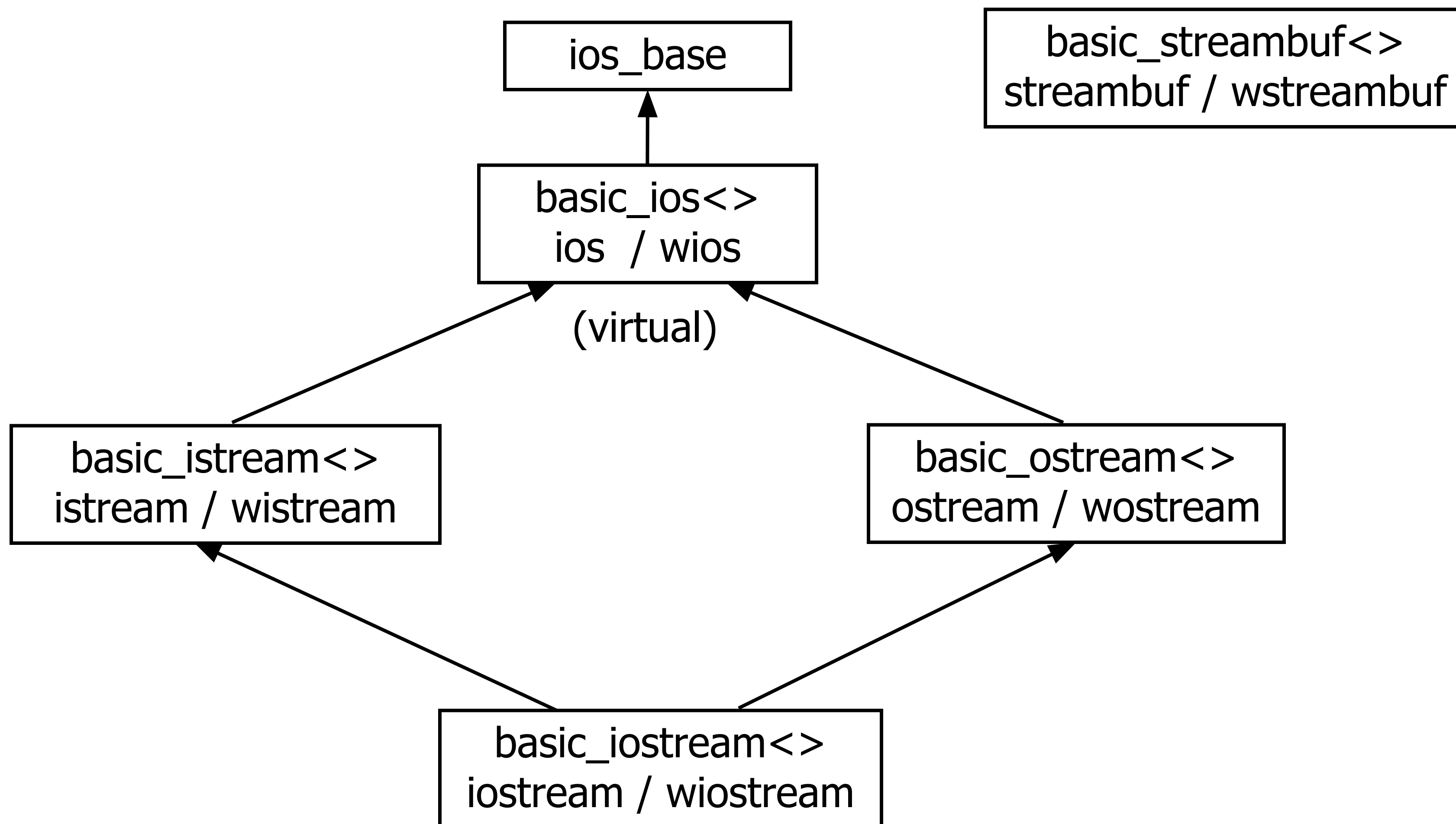
Простой пример

Программа читает два вещественных числа и выводит их произведение.

```
int f()
{
  double x, y;                 // Операнды

cout << "Multiplication of two floating point values" << endl; // Вывод заголовка
  cout << "first operand: ";                                      // Чтение первого операнда
  if (! (cin>> x)) {        /* Ошибка ввода  => вывести сообщение   *
                             * и завершить программу с кодом ошибки */
    cerr << "error while reading the first floating value" << endl;
    return EXIT_FAILURE;
  }

  cout << "second operand: ";                                     // Чтение второго операнда
  if (! (cin>> y)) {       /* Ошибка ввода => вывести сообщение   *
                            * и завершить программу с кодом ошибки */
    cerr << "error while reading the second floating value"<< endl;
    return EXIT_FAILURE;
  }

  cout<<x<<" times "<<y<<" equals "<<x * y<< endl;   // Вывод операндов и результата
  return EXIT_SUCCESS;
}
```

Иерархия основных потоковых классов

Для шаблонных классов в верхней строке указано имя шаблона,
а в нижней - имена специализаций для типов **char** и **wchar_t**



Назначение основных потоковых классов

Базовый класс **ios_base** определяет свойства всех потоковых классов, не зависящие от типа и трактовок символов. Класс в основном состоит из компонентов и функций, предназначенных для управления состоянием и флагами формата.

Шаблон класса **basic_ios<>**, производный от **ios_base**, определяет общие свойства всех потоковых классов, зависящие от типа и трактовок символов. В число этих свойств входит определение буфера, используемого потоком данных. Буфер представлен объектом класса, производным от базового класса **basic_streambuf<>**, с соответствующей специализацией. Фактически именно он выполняет операции чтения/записи.

Назначение основных потоковых классов - 2

Шаблоны **basic_istream<>** и **basic_ostream<>**, виртуально производные от **basic_ios< >**, определяют объекты, которые могут использоваться соответственно для чтения и записи. Эти классы, как и **basic_ios< >**, оформлены в виде шаблонов, параметризованных по типу и трактовкам символов. Если проблемы интернационализации несущественны, задействуются специализации этих классов для типа символов **char** (а именно **istream** и **ostream**).

Шаблон **basic_iostream<>** является производным от двух шаблонов – **basic_istream<>** и **basic_ostream<>**. Он определяет объекты, которые могут использоваться как для чтения, так и для записи.

Шаблон **basic_streambuf<>** занимает центральное место в библиотеке IOStream. Он определяет интерфейс всех представлений, записываемых в потоки данных или читаемых из потоков данных, и используется другими потоковыми классами для фактического чтения или записи символов. Для получения доступа к некоторым внешним представлениям классы объявляются производными от **basic_streambuf< >**.

Назначение потоковых буферных классов

Библиотека IOStream предполагает строгое разделение обязанностей.

Классы, производные от **basic_ios**, ограничиваются *форматированием данных*. Операции чтения и записи символов выполняются потоковыми буферами, которые представлены объектами, подчиненными по отношению к классу **basic_ios**. Потоковые буферы обеспечивают выполнение чтения/записи в символьных буферах и помогают абстрагироваться от внешнего представления (например, файлов или строковых данных).

Потоковые буферы играют важную роль при выполнении ввода-вывода с новыми внешними представлениями (например, сокетами или компонентами графического интерфейса), перенаправлении потоков данных или их конвейерном объединении (например, при сжатии выходных данных перед их передачей в другой поток данных). Кроме того, потоковые буферы обеспечивают синхронизацию при одновременном вводе-выводе с одним внешним представлением.

Потоковые буферы упрощают определение новых «внешних представлений» (скажем, предназначенных для работы с новым носителем данных). Для этого требуется лишь объявить новый потоковый буферный класс, производный от **basic_streambuf<>** (или его подходящей специализации) и определить функции чтения и/или записи символов для нового внешнего представления.

Заголовочные файлы

Определения потоковых классов распределены по нескольким заголовочным файлам.

- **<iosfwd>**. Содержит опережающие объявления потоковых классов. Этот заголовочный файл необходим из-за того, что простые опережающие объявления вида *class ostream* теперь не разрешены.
- **<streambuf>**. Содержит определения базового потоков ого класса с буферизацией (*basic_streambuf<>*).
- **<istream>**. Содержит определения классов, поддерживающих только ввод (*basic_istream<>*, а также классов с поддержкой ввода и вывода (*basic_iostream<>*).
- **<ostream>**. Содержит определения потокового класса вывода (*basic_ostream<>*).
- **<iostream>**. Содержит объявления глобальных потоковых объектов (таких, как *cin* и *cout*).

Многие из этих заголовочных файлов предназначены для внутренней организации стандартной библиотеки C++. Прикладному программисту обычно достаточно включить файл **<iosfwd>** в объявление потоковых классов и **<istream>** или **<ostream>** при непосредственном использовании функций ввода или вывода.

Заголовок **<iostream>** следует включать только при использовании стандартных потоковых объектов. В некоторых реализациях в начале работы каждой единицы трансляции, включающей этот заголовок, выполняется фрагмент кода инициализации.

Само по себе выполнение этого кода обходится недорого, но при этом приходится загружать соответствующие страницы исполняемого файла, а эта операция может быть довольно дорогостоящей. Как правило, в программу следует включать только заголовки, содержащие абсолютно необходимые объявления. В частности, в заголовочные файлы должен включаться только заголовок **<iosfwd>**, а соответствующие файлы реализации включают заголовок с полным определением.

Специальные средства работы с потоками данных (параметризованные манипуляторы, файловые и строковые потоки данных) определяются в дополнительных заголовочных файлах **<iomanip>**, **<fstream>** , **<sstream>** и **<strstream>**. На первый взгляд кажется, что объявлять классы с поддержкой ввода и вывода в заголовке **<istream>** нелогично. Но так как в начале работы каждой единицы трансляции, включающей **<iostream>**, тратится время на инициализацию, объявления для ввода и вывода были выделены в файл **<istream>**.

Состояние потока данных - константы

Общее состояние потока данных определяется несколькими флагами, представленными константами типа **iostate**. Тип **iostate** определяется в классе **ios_base**. Конкретный тип констант зависит от реализации (иначе говоря, стандарт не указывает, является тип iostate перечислением, определением целочисленного типа или специализацией класса bitset).

Константа	*Описание*
goodbit	Нормальное состояние потока данных, другие биты не установлены
eofbit	Обнаружен признак конца файла
failbit	Ошибка; операция ввода-вывода завершилась неудачно
badbit	Фатальная ошибка, неопределенное состояние потока данных

Бит **goodbit** по определению равен 0. Таким образом, установка флага **goodbit** означает, что все остальные биты также равны 0. Имя **goodbit** выбрано не совсем удачно, поскольку нормальное состояние потока данных обозначается не установкой отдельного бита, а сбросом всех остальных битов.

Состояние потока данных – константы 2

Основное отличие флагов **failbit** и **badbit** состоит в том, что последний обозначает более серьезную ошибку.

- **failbit** устанавливается в том случае, если операция завершилась неудачно, но состояние потока данных позволяет продолжить работу. Обычно этот флаг устанавливается при ошибках форматирования в процессе чтения - например, если программа пытается прочитать целое число, а следующий символ является буквой.
- **badbit** указывает на неработоспособность потока данных или потерю данных, например, при установке указателя в файловом потоке данных перед началом файла.
- **eofbit** обычно устанавливается вместе с **failbit**, поскольку признак конца файла проверяется и обнаруживается при попытке чтения за концом файла. После чтения последнего символа флаг **eofbit** не устанавливается, он устанавливается вместе с флагом **failbit** при следующей попытке чтения символа.

Константы флагов определяются не глобально, а в классе **ios_base**. Это означает, что они всегда должны использоваться с указанием области видимости или с конкретным объектом. Пример: **std::ios_base::eofbit**. Или от производных классов: **std::ios::eofbit**.

Потоковые буферы состояния не имеют, они могут использоваться несколькими потоками.

Функции для работы с состоянием потока данных

Основное отличие флагов **failbit** и **badbit** состоит в том, что последний обозначает более серьезную ошибку.

Функция	*Описание*
good()	Возвращает true, если поток данных находится в нормальном состоянии (то есть при установке флага goodbit)
eof()	Возвращает true при обнаружении признака конца файла (установка флага eofbit)
fail()	Возвращает true при обнаружении ошибки (установка флага failbit или badbit)
bad()	Возвращает true при обнаружении фатальной ошибки (установка флага badbit)
rdstate()	Возвращает установленные флаги
clear()	Сбрасывает все флаги
clear(state)	Сбрасывает все флаги и устанавливает флаги, содержащиеся в state
setstate(state)	Устанавливает флаги, содержащиеся в state

Пример работы с функциями

*Как установить и сбросить флаг **failbit**:*

```
         // Проверка установки флага failbit
if (strm.rdstate() & std::ios::failbit) {
  std::cout << "failbit was set" << std::endl;
         // Сброс только флага failbit
  strm.clear (strm.rdstate() & ~std::iOs::failbit);
}
```

Здесь используются поразрядные операторы **&** и **~**. Оператор **~** возвращает поразрядное дополнение своего аргумента. Следовательно, показанное ниже выражение возвращает временное значение, в котором установлены все биты, кроме **failbit**: **~ios::failbit**.

Оператор **&** возвращает результат поразрядного пересечения своих операндов. В результате операции устанавливаются только биты, установленные в обоих операндах. При поразрядном объединении всех текущих установленных флагов (**rdstate()**) со всеми установленными битами, кроме **failbit**, бит **failbit** сбрасывается, а значения всех остальных битов сохраняются.

Класс *istream*

Класс **istream** предназначен для извлечения данных из потока ввода. К этому классу относится объект **cin**.

Для этого класса перегружен оператор сдвига >>, который извлекает из потока ввода данные любого стандартного типа, разделенные пробельными символами (к ним относятся пробелы, знаки табуляции, конца строки, конца страницы). Кроме того в нем определены следующие функции для чтения последовательностей символов:

Функция	Признак конца чтения	Количество символов	Присоединение завершителя	Возвращаемый тип
get(s, num)	Новая строка (без включения) или конец файла	До num-1	Да	istream
get(s, num, t)	t (без включения) или конец файла	До num-1	Да	istream
getline(s, num)	Новая строка (с включением) или конец файла	До num-1	Да	istream
getline(s, num, t)	t (с включением) или конец файла	До num-1	Да	istream
read(s, num)	Конец файла	num	Нет	istream
readsome(s, num)	Конец файла	До num	Нет	streamsize

Класс *istream* - функции ввода символа

int *istream::get* ()

1) Читает следующий символ.

2) Возвращает прочитанный символ или EOF.

3) В общем случае возвращаемое значение относится к типу **traits::int_type**, а EOF величина, возвращаемая при вызове **traits::eof()**. Для *istream* это соответственно тип *int* и константа EOF. Следовательно, для *istream* эта функция соответствует функциям **getchar()** и **getc()** языка С.

4) Возвращаемое значение не обязательно относится к типу символов потока данных; оно также может относиться к типу с более широким диапазоном значений. Без этого было бы невозможно отличить EOF от символа с соответствующим значением.

***istream& istream::*get *(char&* c)**

1) Присваивает следующий символ аргументу с.

2) Возвращает объект потока данных, по состоянию которого можно проверить, успешно ли выполнено чтение.

Класс *istream* - функции get

istream& istream::*get *(char* str,* streamsize *count)

istream& istream::*get *(char* str,* streamsize *count, char delim)

- Обе формы читают до *count-1* символов в строку *str.*
  - Первая форма завершает чтение, если следующий читаемый символ является символом новой строки соответствующей кодировки. Для istream это символ \n.
  - Вторая форма завершает чтение, если следующий читаемый символ является разделителем *delim.*
- Обе формы возвращают объект потока данных, по состоянию которого можно проверить, успешно ли выполнено чтение.
- Завершающий символ *(delim)* не читается.
- Символ завершения строки прерывает чтение.
- Перед вызовом необходимо убедиться в том, что размер *str* достаточен для хранения *count* символов.

Класс *istream* - функции getline и read

istream& istream::*getline *(char* str,* streamsize *count)

istream& istream::*getline *(char* str,* streamsize *count, char delim)

Обе формы идентичны предыдущим функциям get() со следующими исключениями:

- чтение завершается не перед символом новой строки или *delim* соответственно, а включая этот символ, то есть символ новой строки или *delim* будет прочитан, если он встречается среди count-1 символов, но он не сохраняется в *str,*
- если прочитанная строка содержит более count-1 символов, функции устанавливают флаг failbit.

istream& istream::*read *(char* str,* streamsize *count)

- Читает *count* символов в строку *str.*
- Возвращает объект потока данных, по состоянию которого можно проверить, успешно ли выполнено чтение.
- Строка *str* не завершается автоматически символом завершения строки.
- Перед вызовом необходимо убедиться в том, что размер *str* достаточен для хранения *count* символов.
- Обнаружение признака конца файла в процессе чтения считается ошибкой, для которой устанавливается бит failbit (вдобавок к флагу eofbit).

Класс *istream* - функции readsome и gcount

streamsize *istream*::readsome (*char* str*, streamsize *count*)

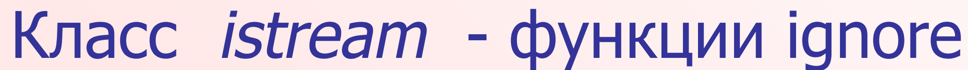
- Читает до *count* символов в строку *str.*
- Возвращает количество прочитанных символов.
- Строка *str* не завершается автоматически символом завершения строки.
- Перед вызовом необходимо убедиться в том, что размер *str* достаточен для хранения *count* символов.
- В отличие от функции *read()* функция *readsome()* читает из потокового буфера все доступные символы (при помощи функции *in_avail()* класса буфера).

Например, она может использоваться в ситуациях, когда ввод поступает с клавиатуры или от других процессов, поэтому ожидание нежелательно. Обнаружение конца файла не считается ошибкой, а биты *eofbit* и *failbit* не устанавливаются

---

streamsize *istream*::gcount () const

- Возвращает количество символов, прочитанных последней операцией неформатированного ввода.



Класс *istream* - функции ignore

***istream& istream*::ignore ()**

istream& istream*::*ignore* (streamsize *count)

istream& istream*::ignore (streamsize *count, int delim)

- Все формы извлекают символы из потока данных и игнорируют их.
- Первая форма игнорирует один символ.
- Вторая форма игнорирует до *count* символов.
- Третья форма игнорирует до *count* символов и прекращает работу тогда, когда будет извлечен и проигнорирован символ *delim.*
- Если значение *count* равно *std::numeric_limits<std::streamsize>::max()*, то есть максимальному значению типа *std::streamsize*, функция игнорирует все символы до тех пор, пока не будет обнаружен ограничитель *delim* или конец файла.
- Все формы возвращают объект потока данных.

Примеры:

```
// игнорирование остатка строки:
cin.ignore(numeric_limits<std::streamsize>::max(), '\n');
// игнорирование всех оставшихся символов cin:
cin.ignore(numeric_limits<std::streamsize>::max());
```

Класс *istream* - функции unget, putback и peek

***istream& istream*::unget ()**

***istream& istream*::putback (*char* c)**

- Обе функции возвращают в поток данных последний считанный символ, чтобы он был считан следующей операцией чтения (если не изменится позиция ввода).
- Различия между функциями *unget()* и *putback()* заключаются в том, что *putback()* проверяет, был ли передаваемый символ ***c*** последним считанным символом.
- Если символ не удается вернуть или функция *putback()* пытается вернуть другой символ, устанавливается флаг *badbit*, что может привести к выдаче соответствующего исключения.
- Максимальное количество символов, которые могут быть возвращены в поток данных этими функциями, зависит от реализации. По стандарту гарантированно работает только один вызов этих функций между двумя операциями чтения.

***int istream:* :peek ()**

- Возвращает следующий считываемый из потока данных символ без его извлечения. Символ считывается следующей операцией чтения (если не изменится позиция ввода).
- Если дальнейшее чтение невозможно, возвращает EOF.
- EOF - значение, возвращаемое traits::eof().

Класс *istream* - функции - резюме

При чтении С-строк описанные здесь функции безопаснее оператора >>, поскольку они требуют явной передачи максимального размера читаемой строки.

Хотя количество читаемых символов можно ограничить и при использовании оператора >>, об этом часто забывают.

Вместо использования потоковых функций ввода часто удобнее работать с потоковым буфером напрямую. Функции потоковых буферов позволяют эффективно читать отдельные символы или последовательности символов без затрат на конструирование объектов sentry.

Также можно воспользоваться шаблонным классом istreambuf_iterator, предоставляющим итераторный интерфейс к потоковому буферу.

Функции tellg() и seekg() предназначены для изменения текущей позиции чтения. В основном они используются при работе с файлами.

Класс *ostream* – функции

***ostream& ostream*::put (*char* c)**

- Записывает аргумент **c** в поток данных.
- Возвращает объект потока данных, по состоянию которого можно проверить, успешно ли выполнена запись.

***ostream& ostream*::write (const *char* str*, streamsize *count*)**

- Записывает *count* символов строки *str* в поток данных.
- Возвращает объект потока данных, по состоянию которого можно проверить, успешно ли выполнена запись.
- Символ завершения строки не останавливает запись и выводится вместе с остальными символами.
- Перед вызовом необходимо убедиться в том, что *str* содержит не менее *count* символов, иначе вызов приводит к непредсказуемым последствиям.

***ostream& ostream*::flush ()**

- Очищает потоковые буферы (принудительная запись всех буферизованных данных на устройство или в канал ввода-вывода, с которым связан буфер).

Пример

Классический фильтр - вывод в выходной поток все прочитанные символы :

```
#include <iostream>
using namespace std;
int main()
{ char c;
 while (cin.get(c)) { // Пока удается прочитать символ
 cout.put(c); // Вывести прочитанный символ в выходной поток
}
```

При каждом вызове cin.get(c) следующий символ присваивается переменной **c**, которая передается по ссылке. Функция get() возвращает объект потока данных; таким образом, условие while остается истинным до тех пор, пока поток данных находится в нормальном состоянии.

Чтобы повысить эффективность программы, можно работать напрямую с потоковыми буферами. Такой интерфейс лучше классического интерфейса фильтров в языке C. В C вам пришлось бы использовать функцию getchar() или getc(), которая возвращает как следующий символ, так и признак конца файла. В этом случае возвращаемое значение пришлось бы обрабатывать как тип int, чтобы отличить символ от признака конца файла.

Тесты …

Вопрос: **Что будет выведено на экран?**

```
#include <iostream>

class A {
   int count;

public:
   A() : count(0) { }
   A(A & a) : count(a.count+1) {}

   A & operator = (A & a) {
      return count *= 10, *this;
   }

   void print() { std::cout << count;  }
};

int main(int argc, char * argv[]) {
   A a, b = a, c = b = b, d = c = c = c;
   c.print();
}
```

Варианты ответа:

(1) 1101
(2) Ошибка компиляции
(3) 1100
(4) 100

Тесты …

Вопрос: **Что будет выведено на экран?**

```
#include <iostream>
 class A {
   int count;
public:
   A() : count(0) { }
   A(A & a) : count(a.count+1) {}
   A & operator = (A & a) {
      return count *= 10, *this;
   }
   void print() { std::cout << count;  }
};
int main(int argc, char * argv[]) {
   A a, b = a, c = b = b, d = c = c = c;
   c.print();
}
```

Создание "a", конструктор по умолчанию, a.count = 0.

Создание "b", конструктор копирования, b.count = a.count + 1 = 1.

Создание "c" надо рассматривать как c = (b = b), т.е. до инициализации "c" надо вычислить выражение в правой части оператора = :

сначала срабатывает оператор присваивания b = b, после чего b.count = 10. А затем уже конструктор копирования для c = b (с новым count), т.е. c.count = b.count + 1 = 10 + 1 = 11.

Создание "d". По аналогии с пред. пунктом сначала вычисляется c = c = c, т.е. два оператора присваивания, где получаем c.count = c.count*10*10 = 11*10*10 = 1100. "d" можно дальше не вычислять.

Варианты ответа:

(1) 1101
(2) Ошибка компиляции
(3) 1100
(4) 100

Тесты …

Что будет выведено на консоль в результате выполнения программы?

```
#include <iostream>
struct A {
    char foo() { return 'A';}
};
template<class T> struct B : public T {
    virtual char foo() {return 'B';}
};
template<class T> struct C : public T {
    virtual char foo() {return 'C';}
};

int main(int argc, char* argv[])
{
    A* a = new A();
    A* b = new B< A >();
    A* c = new C< A >();
    A* d = new B< C< A > >();

    std::cout << a->foo() << b->foo() << c->foo() << d->foo();
    return 0;
}
```

Тесты …

Что будет выведено на консоль в результате выполнения программы?

```
#include <iostream>
struct A {
    char foo() { return 'A';}
};
template<class T> struct B : public T {
    virtual char foo() {return 'B';}
};
template<class T> struct C : public T {
    virtual char foo() {return 'C';}
};

int main(int argc, char* argv[])
{
    A* a = new A();
    A* b = new B< A >();
    A* c = new C< A >();
    A* d = new B< C< A > >();

    std::cout << a->foo() << b->foo() << c->foo() << d->foo();
    return 0;
}
```

AAAA

Параметры шаблонов B и C определяют родителя генерируемого класса.
Так как метод foo() базового класса (класса A) **невиртуальный**, то в данном случае позднее связывание использовано не будет и во всех случаях будет вызван метод foo() класса A.

Тесты …

Вопрос: В каких строках (укажите номера) допущены ошибки:

```
class cl
{
  static int x;
public :
  static int y;
  static void f2();
  void f1();
};

int cl::x=1;
int cl::y=2;
void cl::f1()
{  x++; y++; }                  // 1

void cl::f2()
{  x++; y++; }                  //2

void f3()
{  cl::x++; cl::y++; }      // 3
```

Тесты …

Вопрос: В каких строках (укажите номера) допущены ошибки:

```
class cl
{
  static int x;
public :
  static int y;
  static void f2();
  void f1();
};

int cl::x=1;
int cl::y=2;
void cl::f1()
{  x++; y++; }                //  1

void cl::f2()
{  x++; y++; }                //2

void f3()
{  cl::x++; cl::y++; }     // 3
```

в 3 т.к функция вне класса имеет доступ только к public переменным класса.