Тест с отчетом 2

inline

1) Каким ключевым словом обозначаются встраиваемые функции?

2) Что будет выведено на экран в результате компиляции и выполнения следующего кода?

```
#include <iostream>
using namespace std;
struct A {
   A() { cout << "A"; }
  ~A() { cout << "~A"; }
  virtual void operator()() =
0;
};
struct B : A {
  B() { cout << "B"; }
  ~B() { cout << "~B"; }
  void operator()() { cout <<
"b"; }
};
int main(void) {
  B b;              //AB
  A &a = b;         //B
  a();            //b
  b();              //B   //b
  return 0;         //~B~A
}
```

3) Какой тип имеет такой литерал: "C++" ?

(1) char[4]
(2) char[3]
(3) const char[4]
(4) Ничего из перечисленного
(5) const char[3]

(перечислите номера)

Функции позиционирования в потоках данных C++

Класс	*Фу нкция*	*Описание*
basic_istream<>	tellg()	Возвращает текущую позицию чтения
	seekg(pos)	Устанавливает абсолютную позицию чтения
	seekg(offset, rpos)	Устанавливает относительную позицию чтения
basic_ostream<>	tellp()	Возвращает текущую позицию записи
	seekp(pos)	Устанавливает абсолютную позицию записи
	seekp(offset, rpos)	Устанавливает относительную позицию записи

Позиционирование чтения и записи выполняется отдельными функциями (суффикс «**g**» означает «get», а суффикс «**p**» - «put»). Не все потоковые классы поддерживают позиционирование! Например, для потоков данных **cin**, **cout** и **cerr** позиционирование не определено. Операции файлового позиционирования определяются в базовых классах, потому что обычно используются ссылки на объекты типов **istream** и **ostream**. Функции **seekg()** и **seekp()** могут вызываться для абсолютных или относительных позиций. Функции **tellg()** и **tellp()** возвращают абсолютную позицию в виде значения типа **pos_type**. Это значение не является целым числом или индексом, задающим позицию символа, поскольку логическая позиция может отличаться от фактической.

Что такое позиция?

Что есть тип **pos_type**? Стандартная библиотека C++ определяет глобальный класс шаблона **fpos<>** для представления позиций в файлах. На базе класса **fpos< >** определяются типы streampos (для потоков данных **char**) и **wstreampos** (для потоков данных **wchar_t**). Эти типы используются для определения **pos_type** соответствующих классов трактовок. Наконец, переменная типа **pos_type** класса трактовок требуется для определения типа **pos_type** соответствующих потоковых классов. Следовательно, позиции в потоке данных также могут представляться типом **streampos**, но использовать

типы **long** и **unsigned long** было бы неправильно, потому что **streampos** не является целочисленным типом (а точнее, перестал им быть!). Пример:

```
// Сохранение текущей позиции
 std::ios::pos_type  pos = file.tellg();
// Переход к позиции. хранящейся в pos
 file.seekg(pos);
```

Следующие объявления эквивалентны:

```
 std::ios::pos_type pos;
 std::streamppos pos;
```

Что такое относительная позиция?

В версиях с относительным позиционированием смещение задается по отношению к трем позициям, которые описываются соответствующими константами, определенными в классе **ios_base** и относящихся к типу **seekdir**.

Константа	*Описание*
beg	Смещение задается относительно начала файла
cur	Смещение задается относительно текущей позиции
end	Смещение задается относительно конца файла

Смещение относится к типу **off_type**, который представляет собой косвенное определение для **streamoff**. По аналогии с **pos_type** тип **streamoff** используется для определения **off_type** в классе трактовок и в потоковых классах. Тем не менее **streamoff** является целым знаковым типом, поэтому смещение в потоке данных может задаваться целым числом. Пример:

```
file.seekg (0, std::ios::beg);          // Позиционирование в начало файла
file.seekg (20, std::ios::cur);         // Позиционирование на 20 символов вперед
```

Пример позиционирования

```
// Заголовочные файлы для ввода-вывода
#include <iostream>
#include <fstream>
void printFileTwice (const char* filename)
{
  std::ifstream file(filename);    // Открытие файла
  std::cout << file.rdbuf();       // Первый вывод содержимого
  file.seekg(0);                   // Возврат к началу файла
  std::cout << file.rdbuf();       // Второй вывод содержимого
}
void main (int argc, char* argv[])
{
// Двукратный вывод всех файлов, переданных в командной строке
  for (int i=1; i<argc; ++i) {
    printFileTwice(argv[i]);
  }
}
```

Перенаправление потоков данных

Перенаправление потока данных осуществляется назначением потокового буфера.

В механизме назначения потоковых буферов перенаправление потоков данных находится под управлением программы, операционная система здесь не участвует.

В результате выполнения следующего фрагмента данные, отправленные в поток данных **cout**, будут передаваться не в стандартный выходной канал, а в файл **cout.txt**:

```
std::ofstream file ("cout.txt");
std::cout.rdbuf (file.rdbuf());
```

Для передачи всей форматной информации между потоками данных можно воспользоваться функцией **copyfmt()**:

```
std::ofstream file("cout.txt");
file.copyfmt (std::cout);
std::cout.rdbuf (file.rdbuf());
```

Перенаправление потоков данных - пример

Продолжение…

```
#include <iostream>
#include <fstream>
using namespace std;

void redirect(ostream&);

int main()
{
    cout << "the first row" << endl;
    redirect (cout);
    cout << "the last row" << endl;
}
```

Перенаправление потоков данных - пример

```
void redirect (ostream& strm)
{
    ofstream file("redirect.txt");
    streambuf* strm_buffer = strm.rdbuf(); // Сохранение выходного буфера потока
    strm.rdbuf (file.rdbuf());                    // Перенаправление вывода в файл
    file << "one row for the file" << endl;
    strm << "one row for the stream" << endl;
    strm.rdbuf (strm_buffer);              // Восстановление старого выходного буфера
}                                          // Автоматическое закрытие файла и буфера
```

Результат выполнения программы выглядит так:

```
the first row
the last row
```

Содержимое файла redirect.txt:

```
one row for the file
one row for the stream
```

Потоки чтения и записи

Обычно файл открывается для чтения/записи при помощи класса **fstream**:

```
std::fstream file ("example.txt", std::ios::in | std::ios::out);
```

Можно использовать два разных потоковых объекта, по одному для чтения и записи.

```
std::ofstream out("example.txt", ios::in | ios::out);
std::istream in(out.rdbuf());
```

Объявление **out** открывает файл. Объявление **in** использует потоковый буфер **out** для чтения из него. Обратите внимание: поток данных **out** должен открываться для чтения и записи. Если открыть его только для записи, чтение из потока данных приведет к непредсказуемым последствиям. Также обратите внимание на то, что **in** определяется не с типом **ifstream**, а только с типом **istream**. Файл уже открыт, и у него имеется соответствующий потоковый буфер. Все, что требуется, - это второй потоковый объект. Как и в предыдущих примерах, файл закрывается при уничтожении объекта файлового потока данных.

Потоки чтения и записи

Еще вариант - создать буфер файлового потока данных и назначить его обоим потоковым объектам. Решение выглядит так:

```
std::filebuf buffer;
std::ostream out(&buffer);
std::istream in(&buffer);
buffer.open("example.txt", std::ios::in | std::ios::out);
```

Объект **filebuf** является обычной специализацией класса **basic_filebuf<>** для типа **char**.

Класс определяет потоковый буфер, используемый файловыми потоками данных.

Далее рассмотрим пример, в котором с помощью цикла в файл выводится четыре строки.

После каждой операции вывода все содержимое файла записывается в стандартный выходной поток данных.

Потоки чтения и записи - пример

```
#include <iostream>
#include <fstream>
using namespace std;
void main()
{          // Открытие файла "example.dat" для чтения и записи
    filebuf buffer;
    ostream output(&buffer);
    istream input(&buffer);
    buffer.open ("example.dat", ios::in | ios::out | ios::trunc);
    for (int i=1; i<=4; i++) {
        output << i << ". line" << endl;   // Запись одной строки
    // Вывод всего содержимого файла
        input.seekg(0); // Позиционирование в начало
        char c;
        while (input.get(c)) {  cout.put(c);  }
    cout << endl;
    input.clear ( ); // Сброс флагов eofbit и failbit
    }
}
```

Результат:

1. line

1. line
2. Line

1. line
2. line
3. line

1. line
2. line
3. line
4. line

Потоки чтения и записи - пример

Хотя для чтения и записи используются два разных объекта потоков данных, позиции чтения и записи тесно связаны между собой. Функции **seekg()** и **seekp()** вызывают одну и ту же функцию потокового буфера!

Следовательно, для того чтобы вывести все содержимое файла, необходимо всегда устанавливать позицию чтения в начало файла. После вывода всего содержимого файла позиция чтения/записи снова перемещается в конец файла для присоединения новых строк.

Операции чтения и записи с одним файлом должны разделяться операцией позиционирования (кроме выхода за конец файла во время чтения). Пропуск операции позиционирования приведет к искажению содержимого файла или еще более фатальным ошибкам.

Вместо последовательной обработки символов все содержимое файла можно вывести одной командой, для чего оператору << передается указатель на потоковый буфер:

```
std::cout << input.rdbuf();
```

Потоковые классы для работы со строками

Механизм потоковых классов также может использоваться для чтения или записи в строки. У строковых потоков данных имеется буфер, но нет канала ввода-вывода. Для работы с буфером/строкой используются специальные функции.

Основная область применения строковых потоков данных - обработка вводимых/выводимых данных независимо от фактического механизма ввода-вывода. Например, выводимый текст можно отформатировать в строке и передать в выходной канал позднее. Другой вариант - ввод данных по строкам и обработка строк с использованием строковых потоков данных.

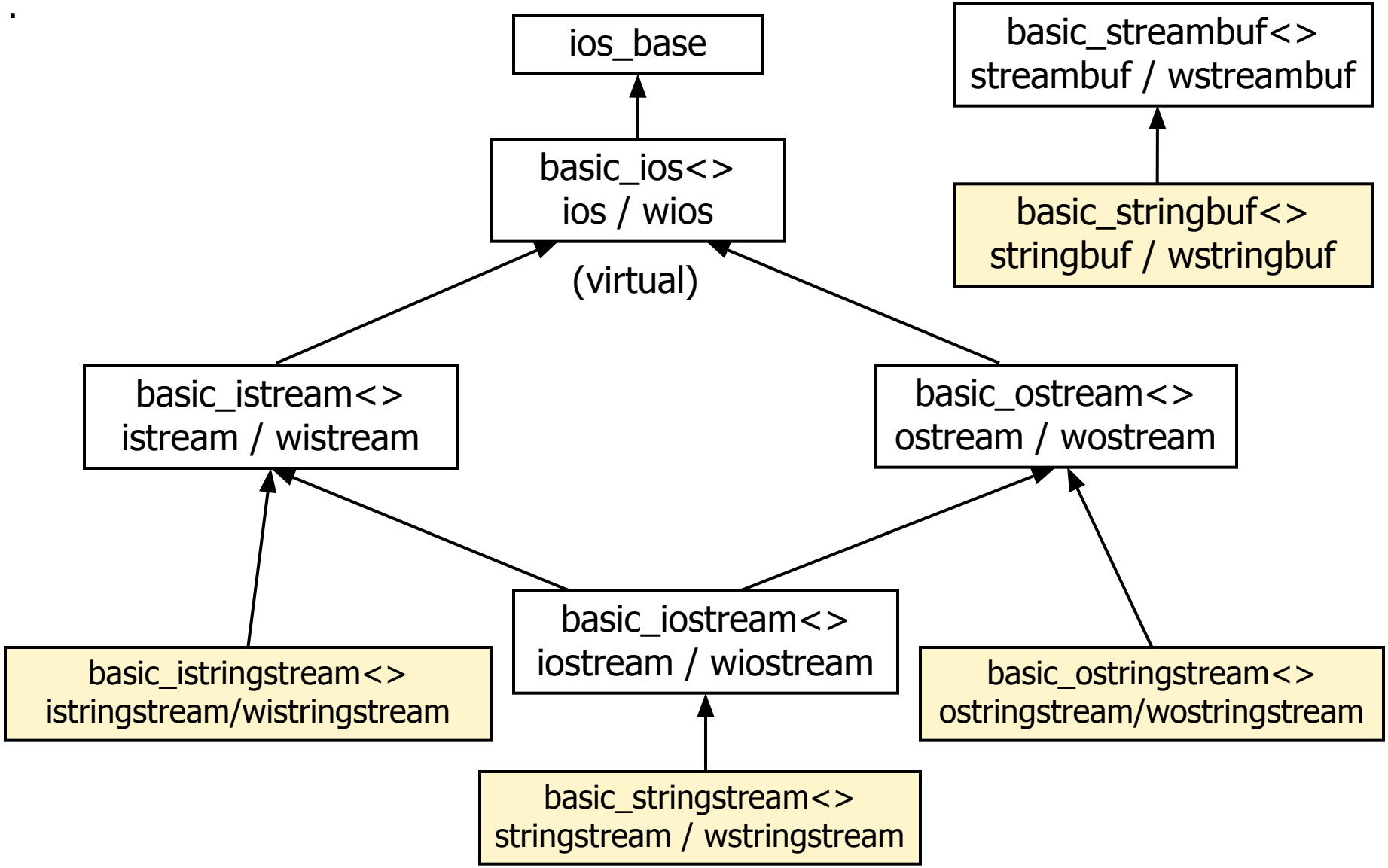
Раньше в классах строковых потоков данных для представления строк использовался тип **char***. В современной библиотеке для представления строк используется тип **string** (или в общем случае - **basic_string<>**.

Классы строковых потоков данных

Для строк определены следующие потоковые классы, которые являются аналогами соответствующих классов файловых потоков данных:

- шаблон **basic_istringstream** со специализациями **istringstream** и **wistringstream** для чтения из строк («строковыЙ входной поток данных»);
- шаблон **basic_ostringstream** со специализациями **ostringstream** и **wostringstream** для записи в строки («строковый выходной поток данных»);
- шаблон **basic_stringstream** со специализациями **stringstream** и **wstringstream** для чтения и записи в строки;
- шаблон **basic_stringbuf<>** со специализациями **stringbuf** и **wstringbuf** используется другими классами строковых потоков данных для реальных чтения и записи символов.

Иерархия классов строковых потоков данных



Основные операции со строковыми потоками данных

Основная операция реализуется функциями:

str() - возвращает буфер в виде строки;

str(string) - присваивает **string** содержимое буфера.

Например

cout << os.str() << endl;

Строковые входные потоки данных в основном используются для форматированного чтения из существующих строк. часто бывает проще читать данные по строкам и затем анализировать каждую строку в отдельности.

В следующем фрагменте из строки s читается целое число x, равное 3, и вещественное число f, равное 0.7:

```
int x;
float f;
std::string s = "3.7";
std::istringstream is(s);
is >> x >> f;
```

Операции со строковыми потоками данных

При создании строковых потоков данных могут задаваться флаги режима открытия файла и/или существующие строки. С флагами **ios::app** и **ios::ate** символы, записанные в строковый поток данных, присоединяются к существующей строке:

```
#include <sstream>
std::string s;
std::ostringstream os (s, ios::ate);
os << 77 << std::hex << 77;
```

Однако строка, возвращаемая функцией **str()**, представляет собой копию строки **s** с присоединенными значениями 77 в десятичном и шестнадцатеричном виде.

При этом сама строка **s** остается неизменной.

Правила построения пользовательских операторов ввода-вывода

Правила, которые должны соблюдаться в пользовательских реализациях операторов ввода-вывода. Они продиктованы типичным поведением стандартных операторов.

- Выходной формат должен допускать определение оператора ввода, читающего данные без потери информации. В некоторых случаях - особенно для строк эта задача практически невыполнима из-за проблем с пробелами. Пробел внутри строки невозможно отличить от пробела, разделяющего две строки.
- При вводе-выводе должна учитываться текущая форматная спецификация потока данных. Прежде всего это относится к ширине поля при выводе.
- При возникновении ошибок должен быть установлен соответствующий флаг состояния.
- Ошибки не должны изменять состояние объекта. Если оператор читает несколько объектов данных, промежуточные результаты сохраняются во временных объектах до окончательного принятия значения.
- Вывод не должен завершаться символом новой строки, в основном из-за того, что это не позволит вывести другие объекты в той же строке.
- При обнаружении ошибки форматирования следует по возможности прекратить чтение.
- Даже слишком большие данные должны читаться полностью. После чтения следует установить соответствующий флаг ошибки, а возвращаемое значение должно

Классы потоковых буферов – вывод символов

Потоки данных не выполняют непосредственные операции чтения и записи, а поручают их потоковым буферам.

С точки зрения пользователя потокового буфера, класс **basic_streambuf** представляет собой нечто, принимающее и отправляющее символы, где **traits_type** определение типа в классе **basic_streambuf**.

Открытые функции вывода символов:

sputc(c) - выводит символ с в потоковый буфер;

sputn(s, n) - выводит **n** символов из последовательности **s** в потоковый буфер.

Функция **sputc()** возвращает **traits_type::eof()** в случае ошибки.

Функция **sputn()** выводит количество символов, заданное вторым аргументом, если только выводу не помешают недостаточные размеры строкового буфера. Символы завершения строк при выводе не учитываются. Функция возвращает количество выведенных символов.

Классы потоковых буферов – ввод символов

При вводе иногда требуется узнать символ без его извлечения из буфера. Кроме того, желательно предусмотреть возможность возврата символов в потоковый буфер. Соответствующие функции поддерживаются классами потоковых буферов.

Функция	*Описание*
in_avail()	возвращает нижнюю границу доступных символов
sgetc()	возвращает текущий символ без его извлечения из буфера
sbumpc()	возвращает текущий символ с извлечением из буфера
snextc()	извлекает текущий символ из буфера и возвращает следующий символ
sgetn(b,n)	читает n символов и сохраняет их в буфере b
sputbackc(c)	возвращает символ c в потоковый буфер
sungetc()	возвращается на одну позицию к предыдущему символу

Потоковые буфера – ввод символов

Функция **in_avail()** проверяет минимальное количество доступных символов. Например, с ее помощью можно убедиться в том, что чтение не будет заблокировано при вводе с клавиатуры. С другой стороны, количество доступных символов может быть больше значения, возвращаемого этой функцией.

Пока потоковый буфер не достигнет конца потока данных, один из символов считается «текущим». Функция **sgetc()** используется для получения текущего символа без перемещения к следующему символу.

Функция **sbumpc()** читает текущий символ и переходит к следующему, который становится текущим.

Функция **snextc()**, переходит к следующему символу и читает новый текущий символ. Для обозначения неудачи все три функции возвращают **traits_type::eof()**.

Функция **sgetn()** читает в буфер последовательность символов, максимальная длина которой передается в аргументе. Функция возвращает количество прочитанных символов.

Функции **sputbackc()** и **sungetc()** возвращаются на одну позицию в потоке данных, вследствие чего текущим становится предыдущий символ. Функция **sputbackc()** может использоваться для замены предыдущего символа другим символом. При вызове этих функций необходимо соблюдать осторожность: нередко возврат ограничивается всего одним символом.

Потоковые буфера – другие функции

Отдельная группа функций используется для подключения объекта локального контекста, для смены позиции и управления буферизацией.

Функция	*Описание*
pubimbue(loc)	Ассоциирует потоковый буфер с локальным контекстом loc
getloc()	возвращает текущий локальный контекст
pubseekpos(pos)	Перемещает текущую позицию в заданную абсолютную позицию
pubseekpos(pos, which)	То же с указанием направления ввода-вывода
pubseekoff(offset, rpos)	Перемещает текущую позицию по отношению к другой позиции
pubseekoff(offset, rpos, which)	То же с указанием направления ввода-вывода
pubsetbuf(b,n)	Управление буферизацией

Функция **pubimbue()** подключает новый объект локального контекста к потоковому буферу и возвращает ранее установленный объект локального контекста. Функция **getloc()** возвращает текущий объект локального контекста.

Потоковые буфера – другие функции - пояснения

Функция **pubsetbuf()** позволяет в определенной степени управлять стратегией буферизации потоковых буферов.

Функции **pubseekoff()** и **pubseekpos()** используются для управления текущей позицией чтения и/или записи. Позиция зависит от последнего аргумента, который относится к типу **ios_base: :openmode** и по умолчанию равен **ios_base::in | ios_base::out**. При установленном флаге **ios_base::in** изменяется позиция чтения, а при установленном флаге **ios_base::out** - позиция записи.

Функция **pubseekpos()** перемещает поток данных в абсолютную позицию, заданную первым аргументом, тогда как функция **pubseekoff()** использует смещение, заданное по отношению к другой позиции. Смещение передается в первом аргументе. Позиция, по отношению к которой задается смещение, передается во втором аргументе и может быть равна **ios_base::cur**, **ios_base::beg** или **ios_base::end**.

Обе функции возвращают новую текущую позицию или признак недействительной позиции. Чтобы обнаружить недействительную позицию, следует сравнить результат с объектом **pos_type(off_type(-1))**.

Текущая позиция потока возвращается функцией **pubseekoff()**:

```
sbuf.pubseekoff(0, std::ios::cur);
```

Синхронизация со стандартными потоками данных С

По умолчанию восемь стандартных потоков данных C++ (четыре символьных потока с однобайтовой кодировкой **cin**, **cout**, **cerr** и **clog**, а также четыре их аналога с расширенной кодировкой) синхронизируются с соответствующими каналами из стандартной библиотеки С (**stdin**, **stdout** и **stderr**).

По умолчанию **clog** и **wclog** используют тот же потоковый буфер, что и **cerr** и **wcerr** соответственно. Таким образом, по умолчанию они синхронизируются с **stderr**, хотя в стандартной библиотеке С у этих потоков данных нет прямых аналогов.

В зависимости от реализации синхронизация может приводить к лишним затратам. Например, если стандартные потоки данных **C++** реализованы с использованием стандартных файлов **C**, это фактически подавляет буферизацию соответствующих потоковых буферов. Однако буферизация необходима для выполнения некоторых оптимизаций, особенно для форматированного чтения. Чтобы программист мог переключиться на нужную реализацию, в классе **ios_base** определена статическая функция:

sync_with_stdio() - возвращает информацию о том, синхронизируются ли стандартные объекты потоков данных со стандартными потоками данных С;

sync_with_stdio(false) - запрещает синхронизацию потоков данных C++ и С (при условии, что функция была вызвана до первой операции ввода-вывода).

Непосредственная работа с потоковыми буферами

Все функции классов **basic_istream** и **basic_ostream**, выполняющие чтение или запись символов, работают по одной схеме: сначала конструируется соответствующий объект **sentry**, а затем выполняется операция. Конструирование объекта **sentry** приводит к очистке буферов возможных связанных объектов, игнорированию пропусков (только при вводе) и выполнению операций, специфических для конкретных реализаций, например операций блокировки файлов в средах с параллельным функционированием нескольких потоков выполнения (**threads**), то есть в многопоточных средах.

При неформатированном вводе-выводе многие операции потоков данных все равно бесполезны, разве что операция блокировки может пригодиться при работе с потоками в средах с параллельным функционированием нескольких потоков выполнения (поскольку в C++ проблемы многопоточности не решаются).

Следовательно, при неформатированном вводе-выводе прямая работа с потоковыми буферами обычно более эффективна.

Непосредственная работа с потоковыми буферами (<<)

Для этого можно определить для потоковых буферов операторы **<<** и **>>**.

(1) При получении указателя на потоковый буфер оператор **<<** выводит все накопленные данные. Это самый быстрый способ копирования файлов с использованием потоков данных C++.

Пример:

```
#include <iostream>
int main ()
{    // Копирование стандартного ввода в стандартный вывод
   std::cout << std::cin.rdbuf();
}
```

В этом фрагменте функция rdbuf() возвращает буфер потока cin. Следовательно, программа копирует весь стандартный входной поток данных в стандартный выходной поток.

Непосредственная работа с потоковыми буферами (>>)

При получении указателя на потоковый буфер оператор **>>** выполняет прямое чтение данных. Например, копирование стандартного входного потока данных в стандартный выходной поток также может выполняться следующим образом:

Пример:

```
#include <iostream>
int main ()
{
 // Копирование стандартного ввода в стандартный вывод
 std::cin >> std::noskipws >> std::cout.rdbuf();
}
```

Обратите внимание на сброс флага **skipws**. В противном случае будут проигнорированы начальные пропуски во входных данных.

Интернационализация

С развитием глобального рынка интернационализация стала играть более важную роль в разработке программного обеспечения. По этой причине в стандартную библиотеку C++ были включены средства написания локализованного кода. В основном они связаны с вводом-выводом и обработкой строк.

Стандартная библиотека C++ предоставляет общие средства поддержки национальных стандартов без привязки к конкретным системам и правилам.

При интернационализации программ следует учесть 2 момента:

- Разные кодировки символов обладают разными свойствами. Для работы с ними необходимы гибкие решения практических вопросов.
- Пользователь рассчитывает, что применяемая им программа соответствует национальным и культурным стандартам.

Основной механизм интернационализации основан на использовании *объекта локального контекста*. Локальный контекст (**locale**) отражает расширяемый набор правил, адаптируемых для конкретных национальных стандартов.

Поддержка разных кодировок символов

Существуют два основных принципа определения кодировок, содержащих более 256 символов: *многобайтовое* и *расширенное* представления.

- В многобайтовых кодировках символы представляются переменным количеством байтов. За однобайтовым символом (например, символом из кодировки ISO Latin-1) может следовать трехбайтовый символ (японский иероглиф).
- В расширенных кодировках символ всегда представляется постоянным количеством байтов независимо от его типа. В типичных кодировках символ представляется величиной 2 или 4 байта.

Многобайтовое представление более компактно по сравнению с расширенным, поэтому для хранения данных вне программ обычно применяется многобайтовое представление. И наоборот, с символами фиксированного размера гораздо удобнее работать, поэтому в программах обычно используется расширенное представление.

Преобразование между кодировками символов осуществляется при помощи шаблона **codecvt<>**. Этот класс используется в основном классом **basic_filebuf<>** для преобразования между внутренними и внешними представлениями.

Трактовки символов

Различия в кодировках существенны для обработки строк и ввода-вывода.

Строковые и потоковые классы специализируются встроенными типами **char** и **wchar_t**. Интерфейс встроенных типов должен оставаться неизменным, поэтому информация о разных аспектах представления символов выделяется в отдельный класс - так называемый *класс трактовок символов*. Он передается строковым и потоковым классам в аргументе шаблона.

По умолчанию в этом аргументе передается класс **char_traits**, параметризованный по аргументу, определяющему тип символов строки или потока данных.

Трактовки символов указываются в классе **char_traits<>**, который определяется в заголовочном файле **<string>** и параметризуется по конкретному типу символов:

```
namespace std {
      template <class charT>
      struct char_traits {
      . . .
      };
}
```

Классы трактовок определяют все основные свойства типа символов и операции, необходимые для реализации строк и потоков данных как статических компонентов.

Классы трактовки символов - функции

<cstring>

Выражение	*Описание*
char_type	Тип символов (то есть аргумент шаблона **char_traits**)
int_type	Тип, размеры которого достаточны для представления дополнительного, не используемого для других целей признака конца файла
pos_type	Тип, используемый для представления позиций в потоке
off_type	Тип, используемый для представления смещений между позициями в потоке
state_type	Тип, используемый для представления текущего состояния в многобайтовых потоках
assign(c1, c2)	Присваивает **c1** символ **c2**
eq(c1, (2)	Проверяет равенство символов **c1** и **c2**
lt(c1, c2)	Проверяет условие «символ **c1** меньше символа **c2**»
length(s)	Возвращает длину строки **s**
compare(s1, s2, n)	Сравнивает до **n** символов строк **s1** и **s2**
copy(s1, s2, n)	Копирует **n** символов строки **s2** в **s1**

Классы трактовки символов – функции (дальше)

Выражение	*Описание*
move(s1, s2, n)	Копирует **n** символов строки **s2** в **s1**, причем строки **s1** и **s2** могут перекрываться
assign(s, n, c)	Присваивает символ **c n** символам строки **s**
find(s, n, c)	Возвращает указатель на первый символ строки **s**, равный **c**; если среди первых **n** символов такой символ отсутствует, возвращает **0**
eof()	Возвращает признак конца файла
to_int_type(c)	Преобразует символ **c** в соответствующее представление типа **int_type**
to_char_type(i)	Преобразует представление **i** типа **int_type** в символ (результат преобразования EOF не определен)
not_eof(i)	Возвращает значение **i**, если **i** не является представлением EOF; в этом случае возвращается значение, определяемое реализацией (и отличное от EOF)
eq_int_type(i1,i2)	Проверяет равенство двух символов **i1** и **i2**, представленных в виде типа **int_type** (иначе говоря, аргументы могут быть равны EOF)

Концепция локального контекста

Распространенный подход к интернационализации основан на использовании специальных сред, называемых локальными контекстами (locale) и инкапсулирующих национальные или культурные стандарты.

В отношении интернационализации локальный контекст представляет собой набор параметров и функций, обеспечивающих поддержку национальных или культурных стандартов. В зависимости от локального контекста выбираются разные форматы вещественных чисел, дат, денежных сумм и т. д.

Обычно локальный контекст определяется строкой в формате

`язык[зона[.код]]`

Здесь **`язык`** - обозначение языка (например, английский или немецкий), а **`зона`** - страна, географический регион или культура, в которой используется этот язык. Квалификатор **`код`** определяет кодировку символов.

locale langLocale(""); - создание локального контекста.

Пустая строка вместо имени локального контекста имеет особый смысл: она

обозначает локальный контекст по умолчанию для окружения пользователя . Этот локальный контекст связывается со стандартным входным потоком данных командой: **cout.imbue(langLocale);**

Фацеты

На функциональном уровне локальный контекст делится на несколько специальных объектов. Объект, обеспечивающий работу некоторого аспекта интернационализации, называется *фацетом*. Это означает, что объект локального контекста может рассматриваться как контейнер для различных фацетов. Для обращения к некоторому аспекту локального контекста тип соответствующего фацета передается в аргументе шаблонной функции use_facet(). Например, следующее выражение обращается к фацету типа numpunct для типа символов char объекта локального контекста loc:

```
std::use_facet<std::numpunct<char> >(loc)
```

Каждый тип фацета определяется в виде класса, предоставляющего определенные сервисы. Например, тип фацета numpunct предоставляет сервис форматирования числовых и логических величин. Так, следующее выражение возвращает строку, используемую для представления **true** в локальном контексте loc:

```
std::use_facet<std::numpunct<char> >(loc).truename()
```

Фацеты делятся на категории, используемые некоторыми конструкторами для создания новых объектов локальных контекстов на основании комбинации других объектов.

Категории фацетов стандартной библиотеки C++

Категории	*Тип фацета*	*Использование*
numeric	num_get<>()	Ввод числовых данных
	num_put<>()	Вывод числовых данных
	numpunct<>()	Символы, используемые при числовом вводе-выводе
time	time_get<>()	Ввод даты и времени
	time_put<>()	Вывод даты и времени
monetary	money_get<>()	Ввод денежных величин
	money_put<>()	Вывод денежных величин
	moneypunct< >()	Символы, используемые при вводе-выводе денежных величин
ctype	ctype<>()	Информация о символах (toupper(), isupper())
	codecvt<>()	Преобразование между кодировками
collate	collate<>()	Контекстное сравнение строк
messages	messages<>()	Чтение строковых сообщений

Фацеты - использование

Определение собственных версий фацетов позволяет создавать специализированные локальные контексты. В следующем примере показано, как определяется фацет для использования русских строковых представлений логических величин:

```
class RusBoolNames : public std::numpunct_byname<char> {
public:
    RusBoolNames (const char *name):
std::numpunct_byname<char>(name) {}
protected:
    virtual std::string do_truename () const {return "истина"; }
    virtual std::string do_falsename () const {return "ложь"; }
};
```

Чтобы использовать этот фацет в локальном контексте, необходимо создать новый объект локального контекста при помощи специального конструктора класса **locale**. В первом аргументе этого конструктора передается объект локального контекста, а во втором - указатель на фацет. Созданный локальный контекст идентичен первому аргументу во всем, кроме фацета, переданного во втором аргументе. Заданный фацет устанавливается в созданном объекте контекста после копирования первого аргумента:

```
std::locale loc (std::locale(" " ). new RusBoolNames(" " );
```

Тесты …

Вопрос: **Какой вывод будет у этой программы?**

```
#include <iostream>

int main()
{
   std::cout << ++static_cast<int&>(*(new int(5)));
}
```

Варианты ответа:

(1) 6

(2) 5

(3) 0

(4)Результат не определён

(5)Ошибка компиляции

Тесты …

Вопрос: **Какой вывод будет у этой программы?**

```
#include <iostream>

int main()
{
   std::cout << ++static_cast<int&>(*(new int(5)));
}
```

Варианты ответа:

(1) 6

(2) 5

(3) 0

(4)Результат не определён

(5)Ошибка компиляции

Тесты …

Вопрос: **Что произойдет при попытке скомпилировать и исполнить нижеприведенный код?**

```
class CBase {
public:
    /*...*/
};

typedef CBase * LPBASE;

void process( const LPBASE pBase)
{
    /*...*/    }

class CDerived : public CBase
{
    /*...*/   };

int main() {
    CDerived d;
    const CDerived* pd = &d;
    process(pd);
    return 0;
}
```

Варианты ответа:

(1)Ошибка компиляции

(2)Ошибка исполнения

(3) Undefined behaviour

(4)Код не содержит ошибок

Тесты …

Вопрос: **Что произойдет при попытке скомпилировать и исполнить нижеприведенный код?**

```
class CBase {
public:
    /*...*/
};

typedef CBase * LPBASE;

void process( const LPBASE pBase)
{
    /*...*/    }

class CDerived : public CBase
{
    /*...*/   };

int main() {
    CDerived d;
    const CDerived* pd = &d;
    process(pd);
    return 0;
}
```

Варианты ответа:

(1)Ошибка компиляции

(2)Ошибка исполнения

(3) Undefined behaviour

(4)Код не содержит ошибок

error C2664: 'process' : cannot convert parameter 1 from 'const CDerived *' to 'const LPBASE'

```
void process( const CBase * pBase)
```

const LPBASE != const CBase*
const будет применен к типу LPBASE
в итоге получим CBase* const