Контейнер **basic_string (string[char] / wstring[wchar_t])**

Его прототип:

template < class CharType**, class** Traits=**char_traits**<CharType>, **class** Allocator=**allocator<**CharType**> > class basic_string**

Здесь CharType – тип хранящихся в контейнере данных, Allocator – распределитель памяти, по умолчанию – стандартного типа **allocator.**

Конструкторы:

basic_string(); – конструктор, создающий пустую строку

basic_string(const basic_string& s, size_type pos = 0, size_type n = npos) ;
– обобщенный конструктор копирования

basic_string(const charT*) ; – конструктор, создающий строку по указателю на 0-завершенную строку

basic_string(const charT* s, size_type n); – конструирует строку из массива по заданной длине

basic_string(size_type n, charT c) – конструктор повторяющегося значения

template <class InIter> **basic_string**(InIter first, InIter last)
– конструктор, создающий очередь по диапазону

Контейнер basic_string – члены класса

Переопределяемые типы :

allocator_type, const_iterator, const_pointer, const_reference, const_reverse_iterator, difference_type, iterator, pointer, reference, reverse_iterator, size_type, value_type – как и для вектора. Кроме того:

npos – беззнаковое целое значение, которое инициализируется –1 и которое вырабатывается в ситуациях "not found" или "all remaining characters" , когда функция поиска завершается неудачей.

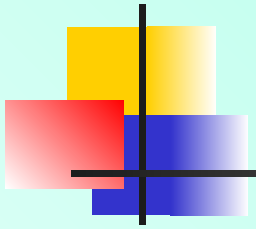
traits_type – тип трактовки символов строки.

В этом классе переопределены операторы:

operator +=	Добавляет символ в конец строки
operator =	Присваивает новое символьное значение содержимому строки
operator []	Обеспечивает ссылку на символ строки по заданному индексу

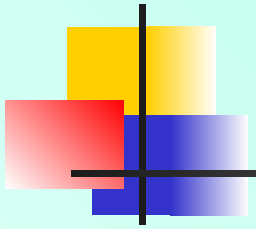
Контейнер basic_string – функции-члены класса

append	Добавляет символы в конец строки
assign	Присваивает содержимому строки специфицированное новое содержание
at	Возвращает ссылку на элемент в заданной позиции строки
begin	Возвращает итератор, адресующий первый символ строки
c_str	Преобразует содержание string в указатель char.* на строку, завершающуюся 0-кодом
capacity	Возвращает объем занятой под строку памяти
clear	Очистка строки
compare	Сравнение строк
copy	Копирует не более заданного числа символов от специфицированной позиции строки в массив символов (устарело. Лучше использовать basic_string::_Copy_s)
_Copy_s	Копирует не более заданного числа символов от специфицированной позиции строки в массив символов
data	Конвертирует содержимое строки в массив символов
empty	Проверяет, пуста ли строка или нет
end	Возвращает итератор на точку за концом строки.



Контейнер basic_string – функции-члены класса

erase	Удаляет элемент или диапазон элементов с заданной позиции
find	Находит в строке первое вхождение указанной подстроки, описываемой заданным образцом
find_first_not_of	Находит первое вхождение в строку символа, не указанного в заданном перечислении
find_first_of	Находит первое вхождение в строку одного из перечисленных символов
find_last_not_of	Находит последнее вхождение в строку символа, не указанного в заданном перечислении
find_last_of	Находит последнее вхождение в строку одного из перечисленных символов
get_allocator	Возвращает объект класса **allocator,** используемый строкой
insert	Вставляет в заданную позицию строки один или несколько символов или специфицированный диапазон символов
length	Возвращает текущее число элементов строки
max_size	Возвращает максимальный размер строки



Контейнер basic_string – функции-члены класса

push_back	Добавляет в конец строки новый элемент
rbegin	Возвращает итератор, который указывает на первый символ инвертированной строки
rend	Возвращает итератор, который указывает непосредственно за последним символом инвертированной строки
replace	Заменяет элементы строки по заданной позиции указанными символами или другой строкой, или С-строкой, или указанным диапазоном
reserve	Устанавливает объем памяти, занятый строкой не более, чем указываемое значение
resize	Устанавливает новый размер строки
rfind	Ищет в строке (в обратном направлении – от конца к началу) подстроку, описываемую указанным образцом
size	Возвращает число символов в строке
substr	Копирует подстроку заданной длины от заданной позиции исходной строки
swap	Обмен содержимого пары строк

Контейнер basic_string – *append* – примеры использования

```
#include <string>
#include <iostream>
using namespace std ;
void main()
{
    string str1("012");
    string str2("345");
    cout << "str1 = " << str1.c_str() << endl;
    str1.append(str2);                    // append str2 to str1
    cout << "str1 = " << str1.c_str() << endl;

    str2 = "567";                        // append the last 2 items in str2 to str1
    str1.append(str2, 1, 2);             // begin at pos 1, append 2 elements
    cout << "str1 = " << str1.c_str() << endl;
```

Контейнер basic_string – *append* – примеры использования

```
                // append the first 2 items from an array of the element type
    char achTest[] = {'8', '9', 'A'};
    str1.append(achTest, 2);
    cout << "str1 = " << str1.c_str() << endl;

    char szTest[] = "ABC";              // append all of a string literal to str1
    str1.append(szTest);
    cout << "str1 = " << str1.c_str() << endl;

    str1.append(1, 'D');                // append one item of the element type
    cout << "str1 = " << str1.c_str() << endl;

    str2 = "EF";                        // append str2 to str1 using iterators
    str1.append (str2.begin(), str2.end());
    cout << "str1 = " << str1.c_str() << endl;
}
```

STL – контейнер *rope* - аналог *string* для очень больших строк

В документации SGI контейнер **rope** описывается так:

Контейнер **rope** представляет собой масштабированную разновидность **string**: он предназначен для эффективного выполнения операций со строками в целом. Затраты времени на такие операции, как *присваивание, конкатенация и выделение подстроки,* практически не зависят от длины строки. В отличие от строк C, контейнер **rope** обеспечивает разумное представление для очень длинных строк (например, содержимого буфера текстового редактора или сообщений электронной почты).

Во внутреннем представлении контейнер **rope** реализуется в виде дерева подстрок с подсчетом ссылок, при этом каждая строка хранится в виде массива **char**. Одна из интересных особенностей интерфейса **rope** заключается в том, что функции **begin** и **end** всегда возвращают тип **const_iterator**. Это сделано для предотвращения операций, изменяющих отдельные символы. Такие операции обходятся слишком дорого. Контейнер **rope** оптимизирован для операций с текстом в целом или большими фрагментами (присваивание, конкатенация и выделение подстроки); операции с отдельными символами выполняются неэффективно.

STL – *ассоциативные контейнеры*

- **set** – множество. Каждый элемент множества является собственным ключом, причем все они уникальны. Поэтому два различных элемента множества не могут совпадать. Например, множество может состоять из следующих элементов: 123, 124, 800, 950.

- **multiset** – мультимножество или множество с дубликатами. Оно отличается от обычного лишь тем, что может содержать в себе повторяющиеся элементы. Например: 123, 123, 800, 950.

- **map** – отображение (словарь). Каждый из элементов словаря имеет несколько членов, один из которых является ключом. В словаре не может быть двух одинаковых ключей. Пример словаря с числовым ключом и сопутствующими символьными данными: (123,Ваня), (124,Маня), (800, Саня), (950, Гена).

- **multimap** – мультиотображение (словарь с дубликатами) – допускается наличие элементов с одинаковыми ключами). Например, допустимым будет набор: (123,Ваня), (123,Маня), (800, Саня), (950, Гена).

В отличие от последовательных контейнеров ассоциативные конвейеры хранят свои элементы отсортированными, вне зависимости от того, каким образом они были помещены в контейнер.

Контейнер set

Это ассоциативный контейнер. Его прототип:

template <**class** Key,**class** Traits= **less** <Key>,**class** Allocator= **allocator**<Key> > **class set**

Здесь Key – элемент данных, сохраняемый в контейнере; Traits – тип, который обеспечивает функциональный объект, который сравнивает значения двух элементов в качестве сортируемых ключей для обеспечения упорядочивания на множестве; Allocator – распределитель памяти, по умолчанию – стандартного типа **allocator.**

Конструкторы:

set(); – конструктор, создающий пустое множество

set(const key_compare& **comp**); – конструктор, создающий множество поддерживающее порядок, задаваемый **comp**

set(const set&); – конструктор копирования

template <**class** InputIterator> **set** (InIterBegin, InIterEnd); – конструктор, создающий множество по диапазону

template <**class** InputIterator> **set** (InItBegin, InIterEnd,

Контейнер set – члены класса

Переопределяемые типы :

allocator_type, const_iterator, const_pointer, const_reference, const_reverse_iterator, difference_type, iterator, pointer, reference, reverse_iterator, size_type, value_type – как и для вектора.

Кроме того:

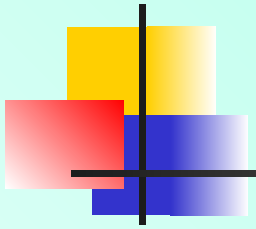
key_compare – тип, который обеспечивает функциональный объект, сравнивающий значения двух ключей для упорядочивания множества.

key_type – тип описывающий сохраняемый в множестве объект.

value_compare – тип, который обеспечивает функциональный объект, сравнивающий значения двух элементов для упорядочивания множества.

Операторы:

bool operator==(const set&, const set&)	Сравнивает пару множеств между собой
bool operator<(const set&, const set&)	Сравнивает пару множеств в лексикографическом порядке



Контейнер set – функции-члены класса

begin	Возвращает итератор, адресующий первый элемент множества
clear	Очистка множества
count	Возвращает число элементов множества, обладающих параметрически специфицированным ключом
empty	Проверяет, пусто ли множество
end	Возвращает итератор элемент за последним элементом множества
equal_range	Возвращает пару итераторов, определяющих специфицированный диапазон значений ключа множества
erase	Удаляет из множества заданный диапазон значений
find	Возвращает итератор, определяющий позицию с искомым ключом
get_allocator	Возвращает объект класса **allocator,** используемый при построении множества
insert	Вставляет в множество один или группу элементов

Контейнер set – функции-члены класса

key_comp	Возвращает копию объекта, используемого для проведения сравнений ключей при упорядочивании элементов множества
lower_bound	Возвращает итератор первого элемента множества, обладающего ключом большим или равным заданному
max_size	Возвращает максимальный размер множества
rbegin	Возвращает итератор, который указывает на первый элемент «обратного» множества
rend	Возвращает итератор, который указывает на следующий за последним элемент «обратного» множества
size	Возвращает число элементов в множестве
swap	Меняет содержание для пары множеств
upper_bound	Возвращает итератор первого элемента множества, обладающего ключом большим заданного
value_comp	Возвращает копию объекта, используемого для проведения сравнений значений элементов при упорядочивании элементов множества

STL – контейнер *set* (пример)

```
#include <set>
#include <iostream>
using namespace std ;
void main()
{ set <int, less<int> > S,T;
  S.insert(1); S.insert(2); S.insert(3); S.insert(1);
  T.insert(2); T.insert(3); T.insert(1);
  if (S==T) cout << "Equal sets, containing:\n";
  for (set <int, less<int> >::iterator i = T.begin();
      i!=T.end(); i++)
    cout << *i << " ";
    cout << endl;
}
```

```
Equal sets, containing:
1 2 3
```

Порядок внесения элементов несущественен, второй раз тот же элемент не вносится. Предикат **less<int>** требуется, чтобы определить упорядочение элементов множества – определить правила вычисления k1 < k2.
Для итераторов множеств допустимы операции ++ и --, но не арифметика.

Структура pair

Эта структура дает возможность работать с двумя объектами, как с единым целым. (#include <**utility**>)

```
template<class Type1, class Type2>
struct pair {
    typedef Type1 first_type;
    typedef Type2 second_type;
    Type1 first;
    Type2 second;
     pair( );    // 1
     pair( const Type1& __Val1, const Type2& __Val2 ); // 2

     template<class Other1, class Other2>                // 3
         pair( const pair<Other1, Other2>& _Right );
     void swap(pair<Type1, Type2>& _Right);
};
```

Структура pair – логические операторы и функция make_pair

Для этой структуры переопределены операторы:

```
template<class _T1, class _T2> inline
   bool operator==(
      const pair<_T1, _T2>& _X,
      const pair<_T1, _T2>& _Y
   ) {return (_X.first == _Y.first && _X.second == _Y.second
);}

template<class _T1, class _T2> inline
   bool operator<(
      const pair<_T1, _T2>& _X,
      const pair<_T1, _T2>& _Y
   ) { return (_X.first < _Y.first || !( _Y.first < _X.first)
               && _X.second < _Y.second ); }
```

```
template <class T1, class T2>
  pair<T1,T2> make_pair(const T1&,const T2&)
```

pair<T1,T2>(x,y)

Структура pair –пример

```
#include <utility>
#include <iomanip>
#include <iostream>
using namespace std ;
void main()
{
  pair <int, double> p1 ( 10, 1.1e-2 );
  pair <int, double> p2;
  p2 = make_pair ( 10, 2.22e-1 );
  pair <int, double> p3 ( p1 );
  cout.precision ( 3 );
  cout << "The pair p1 is: ( " << p1.first << ", "  << p1.second << " )." << endl;
  cout << "The pair p2 is: ( " << p2.first << ", "   << p2.second << " )." << endl;
  cout << "The pair p3 is: ( " << p3.first << ", "   << p3.second << " )." << endl;
}
```

```
The pair p1 is: ( 10, 0.011 ).
The pair p2 is: ( 10, 0.222 ).
The pair p3 is: ( 10, 0.011 ).
```

Контейнер map

Это ассоциативный контейнер. Его прототип:

template < **class** Key, **class** Type, **class** Traits = **less**<Key>,
class Allocator=**allocator**<**pair** <**const** Key, Type> > **> class map**

Здесь Key – элемент данных, сохраняемый в контейнере; Type – тип сохраняемых в контейнере данных, Traits – тип, который обеспечивает функциональный объект, который сравнивает значения двух ключей для обеспечения упорядочивания на множестве; Allocator – распределитель памяти, по умолчанию – стандартного типа **allocator.**

Конструкторы:

map(); – конструктор, создающий пустое отображение

map(const key_compare& **comp**); – конструктор, создающий отображение поддерживающее порядок, задаваемый **comp**

map(const set&); – конструктор копирования

template <**class** InputIterator> **map** (InIterBegin, InIterEnd); – конструктор, создающий отображение по диапазону

template <**class** InputIterator> **map** (InItBegin, InIterEnd, const key_compare& **comp**); –

Контейнер map – функции-члены класса

allocator_type	const_iterator	const_pointer
const_reference	const_reverse_iterator	difference_type
iterator	key_compare	key_type
mapped_type	pointer	reference
reverse_iterator	size_type	value_type

begin	clear	count	empty
end	equal_range	erase	find
get_allocator	insert	key_comp	lower_bound
max_size	rbegin	rend	size
swap	upper_bound	value_comp	

operator [ ]

STL – контейнер *map* (пример)

```
#include <map>
#include <iostream>

void  main( )
{
   using namespace std;
   typedef pair <const int, double> IntDbl;
   map <int, double> Map1;
   map <int, double> :: key_type key1;
   map <int, double> :: mapped_type mapped1;
   map <int, double> :: iterator pI;
            // value_type нужно использовать
            //  для описания корректного приведения типов
   Map1.insert ( map <int, double>::value_type ( 54, 1.1 ) );
          // Другие способы вресения информации в отображение
   Map1.insert ( IntDbl ( 2, 2.78 ) );
   Map1[ 44 ] = 3.1415;
```

STL – контейнер *map* (пример)

```
   key1 = ( Map1.begin( ) -> first );          // Доступ к ключу и данным
   mapped1 = ( Map1.begin( ) -> second );
   cout << "key = " << key1 << " ---> ";
   cout << "value = " << mapped1 << endl;

   cout  << "Keys:";
   for ( pI = Map1.begin( ) ; pI != Map1.end( ) ; pI++ )
      cout << "\t" << pI -> first;
   cout << endl;

   cout  << "Data:";
   for ( pI = Map1.begin( ) ; pI != Map1.end( ) ; pI++ )
      cout << "\t" << pI -> second;
   cout << endl;
}
```

```
key = 2 ---> value = 2.78
Keys:   2       44      54
Data:   2.78    3.1415  1.1
```

STL – контейнер *multiset* (пример)

```
#include <set>
#include <iostream>
using namespace std;
  struct C {
     int x,y;
     C(int xx=0, int yy=0) {x = xx; y = yy;}
  };
  struct LessC : public std::binary_function<C, C, bool>
  {
    bool operator ()(const C& l, const C& r) const
    {  return (l.x - r.x < 0);   }
  };

void main( )
{
   multiset <C, LessC> Set1;
   multiset <C, LessC>::iterator S_AI, S_RI;
```

STL – контейнер *multiset* (пример)

```
The first element of multiset with a key of 30 is: 15
The 1-t element with a key matching
that of the last element is: (70,30).
This is the last element of multiset.
```

```
C c(30,15);          Set1.insert( c );
c.x = 70, c.y = 30;  Set1.insert( c );
c.x = 30, c.y = 40;  Set1.insert( c );
S_RI = Set1.find (30);
cout<<"The first element of multiset with a key of 30 is: "<<(*S_RI).y<<endl;

                    // The element at a specific location in the multiset can be
                    // found using a dereferenced iterator addressing the location
S_AI = Set1.end( );    S_AI--;    S_RI = Set1.find( *S_AI );
cout<<"The 1-t element with a key matching"<<endl<<
     "that of the last element is:("<<(*S_RI).x<<","<<(*S_RI).y<<")."<<endl;

                        // Note that the first element with a key equal to
                        // the key of the last element is not the last element
if ( S_RI == --Set1.end( ) )
      cout << "This is the last element of multiset." << endl;
else  cout << "This is not the last element of multiset." << endl;
}
```

Какой контейнер выбрать?

Нужна ли возможность вставки нового элемента в произвольной позиции контейнера?

Если да, выбирайте последовательный контейнер; ассоциативные контейнеры не подходят.

Важен ли порядок хранения элементов в контейнере?

Если порядок следования элементов не важен, хэшированные контейнеры попадают в число возможных кандидатов. В противном случае придется обойтись без них.

Должен ли контейнер входить в число стандартных контейнеров C++?

Если порядок следования элементов не важен, хэшированные контейнеры попадают в число возможных кандидатов. В противном случае придется обойтись без них.

К какой категории должны относиться итераторы?

С технической точки зрения итераторы произвольного доступа ограничивают ваш выбор контейнерами **vector**, **deque** и **string**, хотя, в принципе, можно рассмотреть и возможность применения **rope**. Если нужны двусторонние итераторы, исключается класс **slist** и распространенная SGI-реализация хэшированных контейнеров.

Какой контейнер выбрать? (продолжение 1)

Нужно ли предотвратить перемещение существующих элементов при вставке или удалении?

Если да, воздержитесь от использования блоковых контейнеров.

Должна ли структура памяти контейнера соответствовать правилам языка C?

Если должна, остается лишь использовать **vector**.

Насколько критична скорость поиска?

Если скорость поиска критична, рассмотрите хэшированные контейнеры, сортированные векторы и стандартные ассоциативные контейнеры — вероятно, именно в таком порядке.

Может ли в контейнере использоваться подсчет ссылок?

Если подсчет ссылок вас не устраивает, держитесь подальше от **string**, поскольку многие реализации **string** построены на этом механизме. Также следует избегать контейнера **rope**. Конечно, средства для представления строк вам все же понадобятся — попробуйте использовать **vector<char>**.

Какой контейнер выбрать? (продолжение 2)

Потребуется ли обеспечить надежную отмену вставок и удалений?

Если да, понадобится использовать узловой контейнер.

Нужно ли свести к минимуму количество недействительных итераторов, указателей и ссылок?

Если нужно — выбирайте узловые контейнеры, поскольку в них операции вставки и удаления никогда не приводят к появлению недействительных итераторов, указателей и ссылок (если они не относятся к удаляемым элементам). В общем случае операции вставки и удаления в блоковых контейнерах могут привести к тому, что все итераторы, указатели и ссылки станут недействительными.

Не подойдет ли вам последовательный контейнер с итераторами произвольного доступа, в котором указатели и ссылки на данные всегда остаются действительными, если из контейнера ничего не удаляется, а вставка производится только в конце?

Ситуация весьма специфическая, но если вы с ней столкнетесь — выбирайте **deque**.

STL – итераторы

В настоящее время в стандартной библиотеке определено пять категорий итераторов:

- итератор ввода (input – InIter),
- итератор вывода (output – OutIter),
- однонаправленный итератор (forward – ForIter),
- двунаправленный итератор (bidirectional – BiIter),
- итератор с произвольным доступом (random access – RandIter).

Выделяют три разновидности итераторных адаптеров:

- **итераторы вставки;**

  back_inserter(контейнер) **front_inserter**(контейнер) **inserter**(контейнер, позиция)

- **потоковые итераторы;**
  - **istream_iterator и ostream_iterator**
- **обратные итераторы.**

STL – класс iterator

```
template<class Category,class Type,class Distance = ptrdiff_t,
         class Pointer = Type*, class Reference = Type&>
    struct iterator {
        typedef Category iterator_category;
        typedef Type value_type;
        typedef Distance difference_type;
        typedef Distance distance_type;
        typedef Pointer pointer;
        typedef Reference reference;
      };
```

STL – класс iterator

Члены – функции:

advance	Увеличивает итератор на заданное число позиций
back_inserter	Создает итератор, вставляющий новые элементы в конце контейнера
distance	Определяет число приращений, между позициями, адресующими пару итераторов
front_inserter	Создает итератор, вставляющий новые элементы в начало заданного контейнера
inserter	Итераторный адаптер, добавляющий новые элементы в специфицированную позицию контейнера

STL – класс iterator

Операторы:

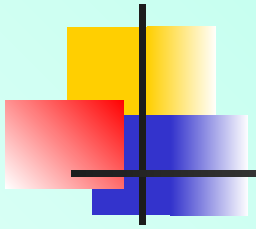
operator!=	Проверка неравенства
operator==	Проверка равенства
operator <	Проверка отношения «меньше»
operator<=	Проверка отношения «меньше или равно»
operator>	Проверка отношения «больше»
operator>=	Проверка отношения «больше или равно»
operator+	Добавляет смещение к итератору и возвращает новый **reverse_iterator,** адресующий вставленный элемент на новой позиции смещения
operator-	Вычитает один итератор из другого, возвращая расстояние между ними

STL – класс istream_iterator

istream_iterator<T, Distance>

istream_iterator – это Input Iterator (итератор ввода), который исполняет форматированный ввод объектов типа T с некоторого отдельного потока. Когда достигается конец потока, формируется специальное «конечное» значение итератора. Итератор ввода перемещается только вперед и поддерживает только чтение (то есть возвращает значения элементов в порядке их перебора).

Выражение	*Описание*
istream_iterator<int>(istream)	Создание итератора потока
istream_iterator<int>()	Создание итератора конца потока
*iter	Обращение к элементу
iter->member	Обращение к переменной или функции элемента
++iter	Смещение вперед (возвращает новую позицию)
iter++	Смещение вперед (возвращает старую позицию)
iterl == iter2	Проверка двух итераторов на равенство
Iterl !=iter2	Проверка двух итераторов на неравенство
TYPE(iter)	Копирование итератора (копирующий конструктор)



STL – класс istream_iterator

Пример –

заполнение вектора данными со стандартного входного потока:

```
vector<int> V;
copy(istream_iterator<int>(cin),
     istream_iterator<int>(),
     back_inserter(V) );
```

STL – класс итератора вывода

Итераторы вывода составляют пару с итераторами ввода. Они тоже перемещаются только вперед, но выполняют запись. Присваивание новых значений выполняется только для отдельных элементов. Итератор вывода не может использоваться для повторного перебора интервала. Запись производится в некую абстрактную "черную дыру"; если вы повторно записываете данные в той же позиции в исходную "черную дыру", ничто не гарантирует, что они будут записаны поверх предыдущих данных.

Операции итераторов вывода:

Выражение	*Описание*
*iter=value	Записывает value в позицию, определяемую итератором
++iter	Смещение вперед (возвращает новую позицию)
iter++	Смещение вперед (возвращает старую позицию)
TYPE(iter)	Копирование итератора (копирующий конструктор)

Для итераторов вывода операции сравнения не нужны!

STL – класс итератора вывода

Итератор ostream_iterator является адаптером итератора вывода. Потоковые итераторы вывода записывают присваиваемые значения в выходной поток данных. Это позволяет напрямую выводить результаты работы алгоритмов в потоки данных через стандартный интерфейс итераторов. Реализация потоковых итераторов вывода основана на тех же принципах, что и реализация итераторов вставки. Единственное различие заключается в том, что присваивание нового значения преобразуется в операцию вывода оператором <<.

Выражение	*Описание*
ostream_iterator<T> (ostream)	Создание потокового итератора вывода для потока данных **ostream**
ostream_iterator<T> (ostream, delim)	Создание потокового итератора вывода для потока данных **ostream** с разделением выводимых значений строкой **delim** (параметр **delim** относится к типу const char*)

При создании потокового итератора вывода необходимо указать выходной поток данных, в который должны записываться данные. В необязательном аргументе можно передать строку, которая должна выводиться между отдельными значениями. Без разделителя элементы будут выводиться непосредственно друг за другом.

Тест с отчетом 4

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

```
#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;
}
```