Лекция 15.

Стандартная библиотека шаблонов STL. Обобщенные алгоритмы.

Адаптер очереди с приоритетом. Функции-члены класса *priority_queue*

Очередь с приоритетом является структурой данных, из которой можно удалить только наибольший элемент (т.е. элемент с наибольшим приоритетом).

В STL очередь с приоритетом реализуется на основе вектора или дека, как адаптер к контейнерам vector или deque. По умолчанию используется дек.

Определены следующие функции:

`type top();`	Значение верхнего элемента
`void push(type);`	Поместить элемент
`void pop();`	Удалить элемент
`int size();`	Количество элементов
`bool empty();`	Очередь пуста?

Пример:

```
// prqueue.cpp: Очередь с приоритетами; программа
//              демонстрирует функции-члены push, pop, empty и top.

#include <iostream>
#include <vector>
#include <functional>
#include <queue>

using namespace std;

int main()
{  priority_queue <int, vector<int>, less<int> > P;
   int x;
   P.push(123); P.push(51); P.push(1000); P.push(17);
   while (!P.empty())
   {  x = P.top();
      cout << "Retrieved element: " << x << endl;
      P.pop();
   }
   return 0;
}
```

```
Retrieved element: 1000
Retrieved element: 123
Retrieved element: 51
Retrieved element: 17
```

Пример обобщенного алгоритма

Обобщенный алгоритм find используется для поиска заданного значения в контейнере. Может использоваться с любыми контейнерами STL.

```
#include <iostream>
#include <cassert>
#include <algorithm> // Алгоритм find
using namespace std;
int main()
{
   cout << "Демонстрация работы обобщенного алгоритма "
        << "find с массивом." << endl;
   char s[] = "C++ is a better C";

   int len = strlen(s);
     // Вызываем find - ищем первое вхождение символа 'e':
   const char* where = find(&s[0], &s[len], 'e');
   assert (*where == 'e' && *(where+1) == 't');
   cout << " ======= Ok!" << endl;
   return 0;
}
```

Ниже - пример работы алгоритма find с вектором.

```
#include <iostream>
#include <cassert>
#include <vector>
#include <algorithm> // Алгоритм find
using namespace std;
<Функция make (создание контейнера символов) >

int main()
{
   cout << "Работа алгоритма find с вектором." << endl;
   vector<char> vectorl =
      make< vector<char> >("C++ is a better C");

   vector<char>::iterator where =
      find(vectorl.begin(), vectorl.end(), 'e');
   assert (*where == 'e' && *(where + 1) == 't');
   cout << " ===== Ok!" << endl;
   return 0;
}
```

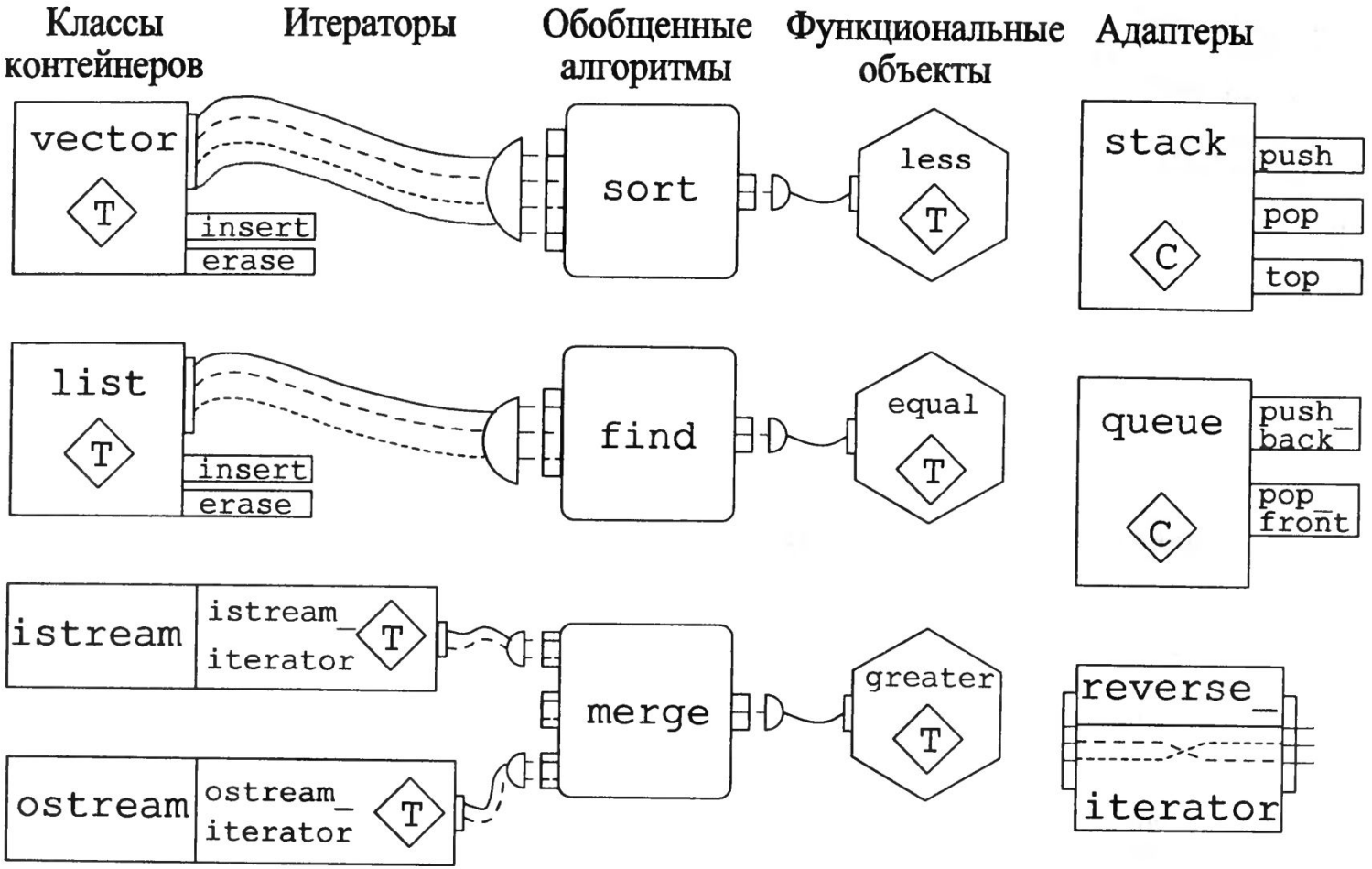
Итераторы

Итераторы представляют собой указателеподобные объекты, которые используются для обхода последовательности элементов.

Если элементы хранятся в массиве (например, char), итераторы **являются** указателями C++ (типа char*). Если элементы хранятся в контейнере STL (таком как vector), тогда итератор не эквивалентен указателю.

Каждый контейнер STL типа C определяет тип C::iterator, который может использоваться с контейнерами этого типа (vector::iterator, list::iterator, deque::iterator и т.д.).

Понятие итератора является ключевым в STL. Алгоритмы STL написаны с применением итераторов в качестве параметров, а контейнеры STL предоставляют итераторы, которые затем могут "включаться" в алгоритмы.



Алгоритм find.

Синтаксис вызова обобщенного алгоритма find

```
where = find(first, last, value);
```

Параметры:

• first - итератор, указывающий на позицию начала поиска;

• last - итератор, указывающий на позицию завершения поиска.

• value - искомое значение.

Возвращаемое значение:

итератор where, указывающий на позицию, в которой был обнаружен элемент со значением value. Возвращается 1-й по порядку элемент. Если ни одного элемента со значением value не найдено, find возвращает итератор, который указывает на значение "за концом" контейнера (last).

Обобщенный алгоритм find и связанный список. Отличие от примера с вектором - для списка list не определена операция +, но определена ++.

```
#include <iostream>
#include <cassert>
#include <list>
#include <algorithm>
using namespace std;

<Функция make (создание контейнера символов)>

int main()
{
   cout << "Работа алгоритма find со списком." << endl;
   list<char> listl = make<list<char>>("C++ is a better C");

   list<char>::iterator where =
      find(listl.begin(), listl.end(), 'e');
   list<char>::iterator next = where;
   ++next;
   assert(*where == 'e' && *next == 't');
   cout << " ===== Ok!" << endl;
   return 0;
}
```

Использование алгоритма find с деком.

```
#include <iostream>
#include <cassert>
#include <deque>
#include <algorithm>
using namespace std;
<Функция make (создание контейнера символов)>

int main()
{
   cout << "Работа алгоритма find с деком." << endl;
   deque<char> dequel =
      make<deque<char>>("C++ is a better C");

   deque<char>::iterator where =
       find(dequel.begin(), dequel.end(), 'e');
   assert(*where == 'e' && *(where + 1) == 't');
   cout << " ===== Ok!" << endl;
   return 0;
}
```

Обобщенный алгоритм merge.

Алгоритм merge используется для объединения двух отсортированных последовательностей в единую (отсортированную) последовательность.

Синтаксис вызова

```
merge(first1, last1, first2, last2, result);
```

Параметры

- first1 и last1 - итераторы начала и конца первой входной последовательности элементов некоторого типа T.
- first2 и last2 - итераторы начала и конца второй входной последовательности элементов того же типа T.
- result - итератор начала последовательности, в которой будет сохранен результат слияния входных последовательностей.

Результат

Предполагается, что две входные последовательности упорядочены в возрастающем порядке в соответствии с оператором < для типа T. Результат работы алгоритма будет содержать все элементы входных последовательностей, причем они также будут отсортированы в порядке возрастания.

Интерфейс шаблона функции merge достаточно гибок. Входная и выходная последовательности могут находиться в контейнерах разного типа.

```
#include <...>
using namespace std;

int main()
{
  cout << "merge с массивом, списком и деком" << endl;
  char s[] = "aeiou";
  int len = strlen(s);
  list<char> list1 = make<list<char>>("bcdfghjklmnpqrstvwxyz");
    // Инициализация deque1 26 символами x
  deque<char> deque1(26, 'x');
    // слияние s и list1, размещение результата в deque1
  merge(&s[0],&s[len],list1.begin(),list1.end(),
       deque1.begin());
  assert(deque1 ==
     make<deque<char>>("abcdefghijklmnopqrstuvwxyz"));
  cout << " ===== Ok!" << endl;
  return 0;
}
```

Алгоритм merge может использоваться для объединения отдельных частей последовательностей.

```
#include <...>
using namespace std;
int main()
{
  cout << "Объединение частей массива и дека"
       << " с размещением результата в списке" << endl;
  char s[] = "acegikm";
  deque<char> deque1 =
      make<deque<char>>("bdfhjlnopqrstuvwxyz");
    // Инициализация списка 26 символами x:
  list<char> list1(26, 'x');
    // Слияние первых 5 символов массива s с первыми
    // 10 символами из deque1, с размещением результата
    // в списке list1:
  merge(&s[0], &s[5], deque1.begin(), deque1.begin()+10,
      list1.begin());
  assert(list1 ==
      make<list<char>>("abcdefghijlnopqxxxxxxxxxxx"));
  cout << " ===== Ok!" << endl;
  return 0;
}
```

Алгоритм accumulate

При вызове `accumulate(first, last, init)`, где first и last - итераторы, а init - некоторое значение, алгоритм accumulate добавляет к init значения в позициях от first до last (не включая значение в последней позиции) и возвращает получившуюся сумму.

Пример: программа расчета суммы элементов вектора.

```
#include <...>
#include <numeric> // Алгоритм accumulate
using namespace std;

int main()
{
   cout << "Демонстрация функции accumulate." << endl;
   int x[5] = {2, 3, 5, 7, 11};
      // Инициализация вектора элементами от x[0] до x[4]:
   vector<int> v1(&x[0], &x[5]);
   int sum = accumulate(v1.begin(), v1.end(), 0);
   assert (sum == 28);
   cout << " ===== Ok!" << endl;
   return 0;
}
```

Рассмотрим, как функция accumulate использует итераторы.

```
template <typename InputIterator, typename T>
T accumulate(InputIterator first, InputIterator last, T init)
{
  while(first != last)
  {
    init = init + *first;
    ++first;
  }
  return init;
}
```

Типы итераторов STL и поддерживаемые ими операции

Функциональные объекты

Из определения шаблона accumulate в предыдущем примере видно, что для элементов контейнера задан **оператор +** (в выражении init = init + *first). Однако абстрактное понятие **накопления (accumulation)** применимо не только к суммированию; можно так же накапливать, к примеру, произведение значений элементов. Потому в STL определена еще один шаблон для функции accumulate

```
template <typename Inputlterator, typename T,
          typename BinaryOperation>
T accumulate(InputIterator first, Inputlterator last,
          T init, BinaryOperation binary_op)
{
   while(first != last)
   {
     init = binary_op(init, *first);
     ++first;
   }
   return init;
}
```

Пример. Использование accumulate для расчета произведения элементов

```
#include <iostream>
#include <vector>
#include <cassert>
#include <numeric>
using namespace std;

int mult(int x, int y) { return x*y; };

int main()
{
   cout << "Расчет произведения элементов вектора" << endl;
   int x[5] = {2, 3, 5, 7, 11};
     // Инициализация вектора элементами от x[0] до x[4]:
   vector<int> v1(&x[0], &x[5]);
   int product = accumulate(v1.begin(), v1.end(), 1, mult);
   assert(product == 2310);
   cout << " ===== Ok!" << endl;
   return 0;
}
```

Пример. Расчет произведения с применением функционального объекта

```
class multiply
{
  public:
    int operator()(int x, int y) const { return x*y; }
};

int main()
{
   cout << "Использование алгоритма accumulate и "
        << "функционального объекта multiply для "
        << " вычисления произведения." << endl;
   int x[5] = {2, 3, 5, 7, 11};
     // Инициализация вектора элементами от x[0] до x[4]:
   vector<int> v1(&x[0], &x[5]);
   int p = accumulate(v1.begin(), v1.end(), 1, multiply());
   assert(p == 2310);
   cout << " ===== Ok!" << endl;
   return 0;
}
```