Язык программирования Си++

Иванов А.П., Князева О.С.

Семинар 6. Шаблоны функций и классов. Библиотека стандартных шаблонов (STL).

1. Шаблоны функций

Часто бывают ситуации, когда один и тот же алгоритм надо оформить в виде нескольких различных функций: в зависимости от типа данных, который этот алгоритм должен обрабатывать. Очевидный пример: функция сортировки – алгоритм сортировки опирается на две фундаментальные операции: сравнения элементов сортируемой последовательности и их перестановки (присваивания). Если эти две операции определены для сортируемого типа данных – то алгоритм сортировки записывается в их терминах универсально:

```
template <class T> void sort ( T arr[], int size )
{
  for (int i = 0; i < size; i++ )
  {
     for (int k = i+1; k < size; k++ )
     {
        if ( arr[k] < arr[i] ) {
           T tmp(arr[i]);
           arr[i]= arr[k];
           arr[k]= tmp;
        }
     }
  }
}
```

Здесь нам от неизвестного заранее класса T понадобится только наличие копирующего конструктора (для tmp), оператора «меньше» и оператора присваивания.

Для целых и вещественных чисел с этими методами нет никаких проблем, они определены, поэтому мы можем сразу воспользоваться этим алгоритмом сортировки для данных видов чисел:

```
.........
int iarr[10] = { 8, 1, 3, 4, 9, 0, 2, 7, 6, 5 };
sort( iarr, sizeof(iarr)/sizeof(iarr[0]) );
.........
double darr[10] = { 8.1, 1.2, 3.3, 4.9, 9.5, 0.4, 2.6, 7.7, 6.8, 5.0 };
sort<double>( darr, sizeof(darr)/sizeof(darr[0]) );
.........
```

Приведенные варианты записи параметра шаблона равнозначны – в первом случае он выводится из типа первого аргумента, во втором – указан явно. Приведенные фрагменты программного кода называются инстанцированием шаблона: соответствующая функция генерируется из описания шаблона в тот момент, когда компилятор встречает в программе ссылку на функцию с конкретными параметрами.

Параметров в шаблоне может быть несколько, причем некоторые из них могут быть не типами, а константами произвольного типа:

```
template <class T, int defsize>
     void sort ( T arr[], int size = defsize )
{
..................
}
```

Для строк операторы присваивания и сравнения придется определить дополнительно, однако, чтобы не «затоптать» операторы присваивания и сравнения для указателей `char*` (ведь именно они представляют строки в языке Си), то для строк лучше объявить свой собственный строковый класс, например, на основе вектора для типа `char` и уже внутри этого класса – определить операторы «<» и «=».

2. Шаблоны классов

Аналогичным образом можно создавать классы, параметризованные типами или константами, например, вектор из ранее приведенного примера:

```
template <class T, int defsize=0> class Vector
{
protected:
T *v;
int len;
public:
//==========Конструкторы объектов класса======//
// Конструктор
      Vector(int N = defsize) : v(0), len(0)
            { if (N>0) v = new T[N];
              if (v) len = N;
            }
.............
};
```

Обратите внимание на возможность задания параметров по умолчанию для аргументов шаблона класса, аналогично тому, как это может быть сделано для значений по умолчанию аргументов обычных функций.

Теперь мы можем создавать вектора любого типа:

```
Vector<double> dv(100);

Vector<char> dv(10);

Vector<char*> dv(20);
```

Шаблоны можно и нужно помещать в заголовочные файлы, однако, реализации методов шаблонных классов должны помещаться в этом же заголовочном файле, нельзя разделять описание класса и реализацию его методов, как это делается при написании обычных классов.

3. Библиотека стандартных шаблонов (STL)

Стандарт языка Си++ включает в себя описание библиотеки стандартных шаблонов (Standard template library, STL), которая включает в себя определения наиболее часто

употребляемых контейнеров (таких, как vector), алгоритмов (например, алгоритм сортировки), которые работают с любыми из этих контейнеров при помощи итераторов (которые являются обобщением понятия «указатель»).

Контейнеры, итераторы и алгоритмы

Контейнеры хранят данные, итераторы позволяют адресовать хранимые в контейнерах данные, а алгоритмы (набор из множества шаблонных функций, определенных в заголовочном файле `<algorithm>`) позволяют при помощи итераторов произвести массу различных действий с любыми контейнерами: сортировки, поиска и т.п.

Кроме шаблонных функций заголовочный файл `<algorithm>` содержит массу определений, так называемых функторов: это шаблонные классы, которые, однако, могут вести себя как шаблонные функции за счет того, что в них переопределен `operator()`.

Динамический массив

Динамический массив позволяет сохранить элементы любого типа в непрерывном участке памяти и адресоваться к этим элементам по целочисленному индексу.

Взятие элемента по индексу занимает константное время, поиск элемента по значению – линейное (по размеру массива) время, вставка нового элемента в середину массива (и удаление элемента из середины) тоже занимает линейное время, а вот добавление нового элемента в конец массива (или удаление последнего элемента) в подавляющем большинстве случаев занимает константное время и лишь изредка – линейное (за счет хранения запаса аллоцированных, но не инициализированных элементов).

```
#include <iostream>
#include <vector>
.................
vector<char> v(100);

v.resize(4);
v[0] = 'a';  v[1] = 'b';
v[2] = 'c';  v[3] = '\0';
cout << &v[0] << endl << flush;

v[3] = 'd';
v.push_back('e'); v.push_back('f');
v.push_back('\0');
cout << &v[0] << endl << flush;

v.erase(v.begin()+2, v.begin()+4);
v.insert(v.begin()+4, 'g');
cout << "size=" << v.size() << ": \"" << &v[0] << "\"" << endl << flush;
```

Дека

Дека, в отличие от массива, позволяет вставлять элементы за константное время не только в конец, но и в начало:

```
#include <deque>
.................
deque<double> d(10);
d.push_front(3.14159);
```

В остальном дека практически идентична массиву. Как правило, деки используются для организации разного рода очередей.

Константное время вставки в начало может быть достигнуто, например, сохранением неинициализированного «запаса» не только в хвосте массива, но и в его начале. Реальное устройство деки гораздо сложнее, но для минимальной реализации достаточно вектора элементов с указаниями: где в нем начинается инициализированная часть и где эта инициализированная часть заканчивается.

Список

Двусвязный список представляет из себя структуру данных, которая помимо хранимого элемента заданного типа хранит еще два указателя: на предыдущий и последующий элементы:

```
struct mylist {
  mytype x;
  mylist *prev, *next;
};
```

Такая конструкция позволяет вставлять (и удалять) элементы в произвольном месте контейнера за константное время (все операции сводятся к замене значений нескольких указателей). Поиск элемента по значению занимает линейное время, доступ к i-тому по порядку элементу – тоже линейное время (в отличие от вектора или деки). К тому же, на каждый элемент приходятся немалые накладные расходы: представим себе, что элементом списка является символ (один байт), в то же время, для хранения двух указателей мы вынуждены в каждом элементе выделять еще, минимум, 8 байт.

```
#include <list>
..............
list<const char*> l(10);
list<const char*>::iterator it;

for( it = l.begin(); it != l.end(); it++ ) {
  *it = "abc";
}

l.insert((l.begin()++)++,"def");
```

Операция вставки выполняется за константное время (т.е. не зависит от количества элементов в списке), однако, операция позиционирования на два элемента вперед, как мы видим, будет линейной, т.е. нам физически придется отсчитать все начальные элементы по очереди, для того, чтобы получить итератор для точки вставки.

Множество

Множество (`set`) позволяет быстро находить хранимые элементы по их значению. «Быстро» означает – не за линейное, а за логарифмическое (по размеру контейнера) время.

В предположении, что все хранимые элементы множества (будем называть их ключами) уникальны – можно построить простейшую реализацию множества при помощи отсортированного массива, в котором поиск производится методом дихотомии (деления пополам).

В настоящей реализации множества используется сбалансированное двоичное дерево поиска, в котором в каждом узле, помимо самого хранимого элемента нужного типа,

сохраняются еще два указателя: на левое и правое поддеревья. При этом все ключи, расположенные в левом поддереве данного узла обязаны быть строго меньше ключа, хранимого в текущем узле, а все ключи, расположенные в правом поддереве обязаны быть строго больше него.

Вставка и удаление элементов в такое дерево занимают тоже логарифмическое время.

```
#include <set>
..............
set<double> s;
s.insert(10.0);
..............
set<double>::const_iterator sit = s.find(2.71828);

multiset<double> ms;
ms.insert(10.0);
..............
multiset<double>::const_iterator msbegin = ms.lower_bound(2.71828);
multiset<double>::const_iterator msend   = ms.upper_bound(2.71828);
```

`multiset` отличается от `set` только тем, что оно допускает хранение произвольного количества равных ключей. Поиск в таком мультимножестве выдает не один итератор найденного элемента, а пару итераторов, первый из которых указывает на начало подпоследовательности, содержащую ключи, равные заданному, а второй итератор – указывает «за последний элемент» этой подпоследовательности.

Ассоциативный контейнер

Ассоциативный контейнер устроен практически так же, как и множество, однако, каждый его элемент содержит пару ключ-значение (а не только ключ), то есть, устанавливает ассоциативную связь от ключа к некоторому значению. Ключ и значение могут быть разными типами и поэтому указываются в параметрах шаблона явно:

```
#include <map>
..............
map<double,int> m;
m[3.14159] = 314;
map<double,int>::const_iterator mit = m.find(3.14159);
cout << "Key: " << mit->first << "Value: " << mit->second << endl << flush;

multimap<double,int> mm;
..............
multimap<double,int>::const_iterator mmit = mm.lower_bound(2.71828);
multimap<double,int>::const_iterator mmend   = mm.upper_bound(2.71828);
for ( ; mmit != mmend; mmit++ ) {
  cout << mmit->first << "," << mmit->second << endl << flush;
}
```

Аналогично мультимножеству `multimap` позволяет сохранять в ассоциативном контейнере элементы (пары ключ-значения) с неуникальными, повторяющимися ключами.

1. Вариант

Создать шаблонную функцию вывода для любого STL-контейнера, параметр шаблона – тип контейнера. Инстанцировать для своего класса, представляющего собой строку, изготовленную из `vector<char>`.

2. Вариант

Создать шаблонную функцию сортировки для STL контейнеров типа `vector`, параметр шаблонной функции передается в параметры шаблона vector, ссылка на который является аргументом этой функции. Инстанцировать `vector` и функцию его сортировки для массива строк (где строка – это свой класс, изготовленный из `vector<char>` с переопределенными операциями необходимыми для работы функции сортировки, т.е. операциями сравнения).

3. Вариант

Создать шаблонную функцию поиска в STL-контейнерах, параметры шаблона – тип итератора (итераторы в параметрах функции указывают на начало и конец контейнера) и тип искомой величины. Инстанцировать для массива строк (где строка – это свой класс, изготовленный из `vector<char>` с переопределенными операциями необходимыми для работы функции поиска, в частности, оператор проверки на равенство).

4. Вариант

Создать шаблонную функцию случайного (`rand`) перемешивания элементов в STL-контейнерах типа `vector`, параметр шаблонной функции передается в параметры шаблона `vector`, ссылка на который является аргументом этой функции. Инстанцировать для массива строк (где строка – это свой класс, изготовленный из `vector<char>`).

5. Вариант

Создать свою функцию случайного (`rand`) перемешивания для STL контейнеров типа `list`, параметр шаблонной функции передается в параметры шаблона `list`, ссылка на который является аргументом этой функции. Инстанцировать для списка из строк (где строки – это свой класс, изготовленный из `vector<char>`).

6. Вариант

Создать свою шаблонную функцию обращения (`reverse`) для STL-контейнеров типа `list` и `vector`. Инстанцировать для обоих контейнеров, содержащих строки (где строка – это свой класс, изготовленный из `vector<char>`).

7. Вариант

Создать свою шаблонную функцию, убирающую повторяющиеся значения в STL-контейнерах типа `vector`, `list`, `deque`. Инстанцировать для контейнера, содержащего строки (где строка – это свой класс, изготовленный из `vector<char>` с переопределенными операциями необходимыми для работы функции удаления, в частности, оператор сравнения на равенство).

8. Вариант

Создать шаблонную очередь `mydeque` с использованием STL-контейнера `vector` (например, из двух векторов сделать одну очередь, так, что вставка в начало очереди – это вставка в конец первого вектора, а вставка в конец очереди – вставка в конец второго вектора). Определить в нем свои функции вставки в начало и в конец. Инстанцировать очередь для вещественных чисел.

9. Вариант

Создать шаблонную очередь `mydeque` с использованием STL-контейнера `vector` (например из двух векторов сделать одну очередь, так, что вставка в начало очереди – это вставка в конец первого вектора, а вставка в конец очереди – вставка в конец второго вектора). Определить в нем свои функции вставки в начало и в конец.
Инстанцировать для своего класса, представляющего строку, изготовленную из `vector<char>`.

10. Вариант

Создать шаблонную очередь `mydeque` с использованием стандартного STL-контейнера vector, таким образом, чтобы добавление в конец и в начало занимало одинаковое количество времени (например, при создании вектора выделяется память с запасом, содержимое вектора хранится в середине массива, добавление в начало очереди это запись в свободные ячейки массива, при исчерпании свободных ячеек память реаллоцируется). Определить в нем свои функции вставки в начало и в конец. Инстанцировать очередь для вещественных чисел.

11. Вариант

Создать шаблонную очередь `mydeque` с использованием STL-контейнера `vector`, таким образом, чтобы добавление в конец и в начало занимало одинаковое количество времени (например при создании вектора выделяется память с запасом, содержимое вектора храниться в середине массива, добавление в начало очереди это запись в свободные ячейки массива, при исчерпании свободных ячеек память реаллоцируется). Определить в нем свои функции вставки в начало и в конец. Инстанцировать для своего класса, представляющего строку, изготовленную из `vector<char>`.

12. Вариант

Создать шаблонную очередь `mydeque` с использованием STL-контейнера `vector` (например, из двух векторов сделать одну очередь, таким образом, что вставка в начало очереди – это вставка в конец первого вектора, а вставка в конец очереди – вставка в конец второго вектора). Определить в нем свои функции вставки в начало и в конец. Инстанцировать очередь для комплексных чисел.

13. Вариант

Создать шаблонную очередь mydeque с использованием STL-контейнера `vector`, так чтобы добавление в конец и в начало занимало одинаковое количество времени (например, при создании вектора выделяется память с запасом, содержимое вектора хранится в середине массива, добавление в начало очереди – это запись в свободные ячейки массива, при исчерпании свободных ячеек память реаллоцируется). Определить в этой шаблонной очереди свои функции вставки и удаления в начало и в конец. Инстанцировать очередь для комплексных чисел.

14. Вариант

Создать шаблонный контейнер ключей `myset` из отсортированного STL-контейнера `vector`. Определить в нем свою функцию поиска по значению и вставки (с учетом того, что ключи должны быть уникальны, и вставка должна происходить таким образом, чтобы контейнер оставался отсортированным). Инстанцировать для вещественных чисел.

15. Вариант

Создать шаблонный класс контейнер ключей myset из отсортированного STL-контейнера `list`. Определить в нем свою функцию поиска по значению и вставки (с учетом того, что ключи должны быть уникальны, и вставка должна происходить таким образом, чтобы контейнер оставался отсортированным). Инстанцировать для вещественных чисел.

16. Вариант

Создать шаблонный класс – ассоциативный массив `mymap` с использованием STL-контейнера `vector`. Ключ и знание должны быть разнотипными. Определить в нем свою функцию вставки пары ключ-значение (с учетом того, что ключи должны быть уникальны) и поиска значения по ключу. Инстанцировать для ключей, являющихся строками символов и значений – вещественных чисел.

17. Вариант

Создать шаблонный класс – ассоциативный массив `mymap` с использованием STL-контейнера `list`. Ключ и знание должны быть разнотипными. Определить в нем свою функцию вставки пары ключ-значение (с учетом того, что ключи должны быть уникальны) и поиска значения по ключу. Инстанцировать для ключей, являющихся строками символов и значений – вещественных чисел.

18. Вариант

Создать свой шаблонный класс – динамический массив (`myvector`). Создать методы этого класса – `size`, `reverse`, `clear`, `push_back`, `pop_back`, `swap`, `sort`. Инстанцировать для своего класса, представляющего собой строку, изготовленную из STL-контейнера `vector<char>` (с определением в нем операций, необходимых для сортировки, в частности, операторов сравнения).

19. Вариант

Создать свой шаблонный класс – динамический массив (`myvector`). Создать методы класса – `size`, `resize`, `clear`, `insert`, `sort`. Инстанцировать для своего класса, представляющего строку, изготовленную из STL-контейнера `vector<char>` (с определением в нем операций, необходимых для сортировки, в частности, операторов сравнения).

20. Вариант

Создать свой шаблонный класс – двусвязный список (`mylist`). Поля класса: поле значений, два указателя на следующий и предыдущий элементы списка. Создать методы класса – `size`, `resize`, `reverse`, `clear`, `insert`, `erase`, `unique`. Инстанцировать для вещественных чисел.

21. Вариант

Создать свой шаблонный класс – односвязный список (`mylist`). Поля класса: поле значений, указатель на следующий элемент списка. Создать методы класса – `size`, `resize`, `clear`, `insert`, `erase`. Инстанцировать для комплексных чисел.

22. Вариант

Создать свой шаблонный класс – двусвязный список (`mylist`). Поля класса: поле значений, два указателя на следующий и предыдущий элементы списка. Создать методы класса – `size`, `resize`, `clear`, `pop_back`, `pop_front`, `push_back`, `push_front`, `merge`, `split`. Инстанцировать для вещественных чисел.

23. Вариант

Создать свой шаблонный класс – дерево (`mytree`). Поля класса: поле значений, указатель на левое поддерево, указатель на правое поддерево. При заполнении дерева нужно обеспечивать, чтобы все ключи в левом поддереве были строго меньше ключа в данном узле, а все ключи в правом поддереве – строго больше. Создать методы класса – `find`, `insert`, `size`. Инстанцировать для вещественных чисел.

24. Вариант

Создать свой шаблонный класс – динамический массив (myvector) указателей на произвольный тип, который является параметром шаблона. Сделать реализацию на основе массива указателей `void*` с приведением хранимых указателей к нужному типу. Создать методы класса – `add`, `remove`, `size`, `resize`, `[]`. Инстанцировать для указателей на вещественные числа.

25. Вариант

Создать свой шаблонный класс – очередь с приоритетом (`myqueue`) на основе отсортированного по убыванию значений vector. Создать методы класса – `size`, `clear`, `push`, `pop`. Метод `push` должен помещать заданное значение в очередь, сохраняя сортировку, метод `pop` –удалять из очереди первый (то есть – наиболее приоритетный) элемент и возвращать его. Инстанцировать для вещественных чисел.