Стандартная библиотека шаблонов

Функциональные объекты

Пример

```
class  Fun {
. . .
public:
   Fun() {   }
   int operator() ( int p1, int p2){
      return p1+p2;
   }
. . .
};
Fun f1; // функциональный объект
```

Функциональные объекты (функторы)

- Рассмотрим простейшую задачу – поиск наибольшего элемента в массиве.
- Положим, что массив представлен в виде объекта класса **`vector <typename T>`**
- Дополнительно потребуем, чтобы для элементов массива была определена функция **`operator<`**

Шаблон функции

```
template <typename T>
const T& findMax(const vector<T> &a)
{ int maxIndex=0;
  for (int i=1; i<a.size();i++)
        if( a[maxIndex]<a[i])
              maxIndex=i;
  return a[maxIndex];
}
```

Недостаток

- Работает только с массивом объектов, для которых определена функция **`operator<`**
- Не может для одних и тех же объектов использовать разные варианты сравнений, например, прямоугольники можно сравнивать по площади, ширине, высоте

Решение

- Передавать в **`findMax`** в качестве второго параметра функцию сравнения для элементов массива
- C и C++ допускают передачу указателя на функцию в качестве параметра, но это допустимо не во всех объектно-ориентированных языках
- Использование функционального объекта (функтора) и передача его вторым параметром в функцию **`findMax`**

Функция с функциональным объектом

```
template <typename Object, typename Comparator>
const Object & findMax
     ( const vector<Object> & a, Comparator comp )
{
   int maxIndex = 0;
   for( int i = 1; i < a.size( ); i++ )
   if(comp.isLessThan(a[maxIndex],a[i]))
         maxIndex = i;
   return a[ maxIndex ];
}
```

Функтор для класса **`Rectangle`**

```
class LessThanByLength
{
  public:
    bool isLessThan(const Rectangle &lhs,
                    const Rectangle & rhs ) const
  {
      return lhs.getLength()< rhs.getLength();
   }
};
```

Пример использования

```
int main(){
  vector <Rectangle> a;
  . . .
  cout<<findMax(a,LessThanByLength())
        << endl;
  . . .
}
```

Дополнительная возможность C++

- В функциональном объекте перегружается оператор вызова функции **`operator()`**
- Это позволяет улучшить вид функции **`findMax`**

Другой вариант функтора (в стиле C++)

```
class LessThanByArea
{
  public:
    bool operator()(const Rectangle & lhs,          const Rectangle & rhs ) const
      {
        return  lhs.getArea( ) < rhs.getArea( );
      }
};
```

Измененная функция с функциональным объектом

```
template <typename Object, typename Comparator>
const Object& findMax1
            (const vector<Object> &a,
                  Comparator   isLessThan )
{
   int maxIndex = 0;
   for( int i = 1; i < a.size( ); i++ )
      if( isLessThan(a[maxIndex], a[i]))   maxIndex = i;
   return a[ maxIndex ];
}
```

```
int main( )
{
   vector<Rectangle> a;
. . .
cout << "Largest length:\n\t"
    << findMax( a, LessThanByLength( ) )   << endl;
cout << "Largest area:\n\t"
    << findMax1( a, LessThanByArea( ) )   << endl;
   return 0;
}
```

Соглашение STL

- Функциональный объект (функтор) – это экземпляр объекта класса, в котором перегружена операции вызова функции

  operator()

- Функторы могут быть параметрами алгоритмов вместо указателей на функции

Пример

```
class LessThan50
{ public:
   bool operator() (int x) const {
        return x<50;}
};
vector  <int> v;
vector <int>::iterator it;
it = find_if (v.begin(), v.end(), LessThan50());
```

Типы функторов

- Каждый алгоритм может предъявлять определенные требования к виду функтора, который может с ним использоваться
  - *Генератор* – функтор, который вызывается без параметров
  - *Унарная функция* – функтор, имеющий один параметр
  - *Бинарная функция* –функтор, имеющий два аргумента
  - *Предикат* (бывает унарным и бинарным) – функтор, возвращающий значение типа *bool*

Стандартные функциональные объекты

- Эквиваленты всех встроенных операций
- Можно использовать со встроенными типами и с любым пользовательским типом, в котором перегружена соответствующая операция

negate <T> ()	-p
plus <T> ()	p1 + p2
minus <T> ()	p1 - p2
multiplies <T> ()	p1 * p2
divides <T> ()	p1 / p2
modulus <T> ()	p1 % p2
equal_to <T> ()	p1 == p2
not_equal_to <T> ()	p1 != p2
less <T> ()	p1 < p2
greater <T> ()	p1 > p2
less_equal <T> ()	p1 <= p2
greater_equal <T> ()	p1 >= p2
logical_not <T> ()	!p
logical_and <T> ()	p1 && p2
logical_or <T> ()	p1 || p2

Пример

```
vector <double> v1,v2;

. . .

ostream_iterator <double, char> out(cout,”  “);

transform(v1.begin(), v1.end(),

            v2.begin(), out, plus<double>());
```

Адаптеры функциональных объектов

- Если интерфейс функционального объекта не соответствует требуемому в алгоритме интерфейсу, можно воспользоваться специальными адаптерам
- Имеются специальные адаптеры, для приспособления бинарных функций к унарному интерфейсу

Адаптеры функциональных объектов

bind1st() позволяет задать постоянное значение для первого аргумента бинарного функтора, превращая его в унарный

bind2nd() выполняет аналогичное преобразование, но полагая второй аргумент постоянным

Пример

- Пусть необходимо заменить все значения вектора на противоположные

```
transform (v.begin(), v.end(), out,
                    negate<double>());
```

В этой версии алгоритма требуется одноместная функция

Если нужно умножить все элементы вектора на одно и то же значение, то функтор multiplies не подходит под интерфейс алгоритма, изменим интерфейс

Пример

```
transform (v.begin(), v.end(), out,
                bind1st(multiplies<double>(), 2.5));
```

В данном случае не имеет значения, какой адаптер использовать – bind1st() или bind2nd()

- Для несимметричных функторов

```
find_if(v.begin(), v.end(),
                bind2nd(greater<int>(), 5))
```

Находит первый, больше 5

```
find_if(v.begin(), v.end(),
                bind1st(greater<int>(), 5))
```

Находит первый, меньше 5

Адаптеры - отрицатели

- Адаптеры

not1(op) !op(param1)

not2(op) !op(param1, param2)

Применяются к одноместным и двуместным предикатам

```
int array [4] = { 4, 9, 7, 1 };
sort (array, array + 4, not2 (greater<int> ()));
```

Стандартная библиотека шаблонов

Алгоритмы

Соглашение STL

- Функции, не являющиеся элементами классов, но предназначенные для работы с контейнерами
- Описаны в заголовке <algorithm>. Некоторые численные алгоритмы — в <numeric>
- В <algorithm> также описаны вспомогательные функции min(), max() и swap() — для двух значений произвольного типа

Соглашение STL

- Все алгоритмы отделены от деталей реализации структур данных и используют в качестве параметров типы итераторов
- Поэтому они могут работать с определяемыми пользователем структурами данных, когда эти структуры данных имеют типы итераторов, удовлетворяющие предположениям в алгоритмах

Правила именования

- Для типов итераторов используются имена, которые указывают какой итератор требуется в алгоритме
- Для некоторых алгоритмов предусмотрены и оперативные и копирующие версии
- Когда такая версия предусмотрена для какого-то алгоритма *algorithm*, он называется *algorithm_copy*
- Алгоритмы, которые используют предикаты, оканчиваются суффиксом *_if*

Категории алгоритмов

- Не модифицирующие контейнер
- Модифицирующие контейнер
- Сортировки и подобные им операции
- Обобщенные числовые операции

Алгоритмы не модифицирующие контейнер

- Эти алгоритмы выполняются над каждым элементом диапазона, оставляя их неизменными
- Например,

int count (InputIterator first, InputIterator last, const T& val)

int count_if (InputIterator first, InputIterator last, UPredicate pred)

Алгоритмы не модифицирующие контейнер

```
InputIterator find (InputIterator first,
                    InputIterator last, const T& value);
InputIterator find_if(InputIterator first,
                    InputIterator last, UPredicate pred);
InputIterator find_if_not(InputIterator first,
                    InputIterator last, UPredicate pred);
bool equal ( InputIterator1 first1,
                    InputIterator1 last1,
```

Алгоритмы не модифицирующие контейнер

Function for_each(InputIterator first,

InputIterator last,

Function f)

Другие алгоритмы

find_first_of ()	find_end()
search()	search_n()
mismatch()	all_of()
any_of()	none_of()

Пример

```
int numbers[10];
int* location = find (numbers, numbers + 10, 25);


vector <int> v;
bool div_3 (int a_) { return a_ % 3 ? 0 : 1; }
vector<int>::iterator iter;
iter = find_if (v.begin (), v.end (), div_3);
```

Пример

```
vector <int> v1, v2;

iter = search (v1.begin (), v1.end (),

                    v2.begin (), v2.end ());

if (iter == v1.end ())

    cout << "v2 not contained in v1" << endl;

else
```

Модифицирующие контейнер алгоритмы

- Копировать

```
OutputIterator copy (InputIterator first, InputIterator last,
        OutputIterator result);

OutputIterator copy_n (InputIterator first, Size n,

                OutputIterator result);

OutputIterator copy_if (InputIterator first, InputIterator last,

                        OutputIterator result, UPredicate pred);


BidirectionalIterator2

        copy_backward ( BidirectionalIterator1 first,
```

Модифицирующие контейнер алгоритмы

- Обменять

  swap();

  void iter_swap (ForwardIterator1 a,

  ForwardIterator2 b) ;

  ForwardIterator2 swap_ranges (ForwardIterator1 first1,

  ForwardIterator1 last1,

  ForwardIterator2 first2)

- Другие

  transform() replace() replace_if()

  replace_copy() fill() fill_n()

Модифицирующие контейнер алгоритмы

- Генерировать

  generate( )

  получает в качестве параметра функцию-генератор

- Удалить

remove() remove_if()

remove_copy() remove_copy_if()

- Убрать повторы

  unique() unique_copy()

Пример

```
int numbers[10];
// function generator:
int    RandomNumber () {
         return std::rand() % 100 ;
}
generate (numbers, numbers + 10,
                RandomNumber);
```

Пример

```
class Fibonacci {
 public: Fibonacci () : v1 (0), v2 (1) {}
      int operator () ();
private:
      int v1;  int v2;
};
int Fibonacci::operator () () {
      int r = v1 + v2;
      v1 = v2; v2 = r;
      return v1;
```

Пример

```
int main () {
        vector <int> v1 (10);
     Fibonacci  generatorF;
        generate (v1.begin (), v1.end (), generatorF);
 ostream_iterator<int> iter (cout, "  ");
        copy (v1.begin (), v1.end (), iter);
        cout << endl;
        return 0;
}
```

Перестановки

- Расположить в обратном порядке

void reverse (BidirectionalIterator first,

BidirectionalIterator last)

OutputIterator reverse_copy (BidirectionalIterator first,

BidirectionalIterator last, OutputIterator result)

- Переместить циклически

rotate() rotate_copy()

- Перемешать

random_shuffle()

- Разделить

Сортировки

- Все операции имеют две версии: одна берёт в качестве параметра функциональный объект типа *Compare*, а другая использует *operator<*

Сортировки

- Сортировка

```
void sort (RandomAccessIterator first,
           RandomAccessIterator last);
void sort (RandomAccessIterator first,
           RandomAccessIterator last,
           Compare comp);
bool is_sorted (ForwardIterator first,
ForwardIterator last);
```

Сортировки

- Другие

stable_sort()

partial_sort()

partial_sort_copy()

- N-ный элемент

nth_element()

Сортировки

- Бинарный поиск

Работают с итераторами произвольного доступа, уменьшая число сравнений, которое будет логарифмическим для всех типов итераторов. Для итераторов не произвольного доступа они выполняют линейное число шагов

lower_bound() upper_bound()

equal_range() binary_search()

- Слияние

merge() inplace_merge()

Дополнительная информация

http://www.cplusplus.com/reference/algorithm/

Обобщенные численные операции

- <numeric>

```
T accumulate (InputIterator first,
              InputIterator last, T init);

T accumulate (InputIterator first,
              InputIterator last, T init,
               BinaryOperation binary_op);
```

Пример

- Сумма

```
int sum = accumulate (v.begin (), v.end (), 0);
```

- Произведение

```
int mult (int val, int elem) {
        return  val* elem;
}
int prod = accumulate (v.begin (), v.end (), 1
```

Пример

- Используем функциональный объект

```
int init = 100;
int numbers[] = {10,20,30};
 cout<<accumulate (numbers, numbers+3,
                   init, minus<int>());
```

Обобщенные численные операции

- Скалярное произведение

```
result = inner_product (vector1, vector1 + 5,
                        vector2, 0);

int add (int a_, int b_) { return a_ + b_; }
int mult (int a_, int b_) { return a_ * b_; }

int result = inner_product (v1.begin (), v1.end (),
v2.begin (), 1, mult, add);
```

Обобщенные численные операции

- Частичная сумма

partial_sum (v1.begin (), v1.end (), v2.begin ());

записывает во второй диапазон частичные суммы для каждой позиции в первом диапазоне:

*i1, *i1 + *i2, *i1 + *i2 + *i3

Обобщенные численные операции

- Смежные разности

```
adjacent_difference (v.begin (), v.end (),
    result.begin ());
```

записывает во второй диапазон для каждой позиции разность её и предыдущего элемента:

*i1, *i2 - *i1, *i3 - *i2, ...

Класс string

- Шаблон

basic_string

- Классы

string

u16string

u32string

wstring

Конструкторы

```
string();
string (const string& str);
string (const string& str, size_t pos,
                size_t len = npos);
string (const char* s);
string (const char* s, size_t n);
string (size_t n, char c);
template <class InputIterator>
  string (InputIterator first, InputIterator last);
```

Операции

operator=

operator[ ]

operator+=

operator+

operator<<

operator>>

и все операции сравнения

Итераторы

begin ()	cbegin()
end ()	cend()
rbegin()	crbegin()
rend()	crend()

Методы для операций со строками

size()	length()
max_size()	capacity()
resize()	reserve()
clear()	empty()
at()	
back()	front()
insert()	erase()
replace()	copy()