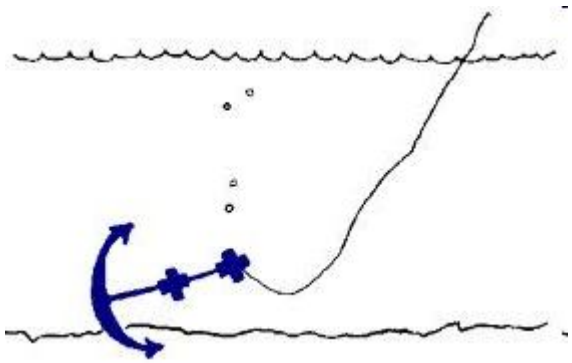
Карпов В.Э.

Объектно-ориентированное программирование

C++. Лекция 9



Библиотека STL

Standard Template Library (STL, Александр Степанов и Менг Ли, Hewlett-Packard Lab) - надстройка над C++.

Задачи:

- Упростить работу с C++
- Сделать ее «комфортной».

Главная идея **STL** - уменьшение зависимости от стандартных библиотек **C++**. Основная проблема стандартных библиотек - их тесная связь с данными, что делает эти библиотеки неудобными для работы с типами данных пользователя. **STL** позволяет работать с любыми типами данных и производить над ними операции.

- **STL** отделяет структуры данных от алгоритмов, которые с ними работают.

С 1994 года STL стала частью официального стандарта языка C++.

Возможности STL

- классы *string и wstring* реализующих динамические строки (с однобайтовыми и двубайтовыми символами);
- класс complex реализующий комплексные числа;
- классы по локализации приложений;
- потоки ввода/вывода для файлов, консоли и строк;
- классы обработки исключений;
- *итераторы* - сходные по функциональности с указателями объекты, используемые для обработки элементов контейнерных типов;
- контейнерные классы - классы по управлению множеством элементов одного типа, как
  - *vector* - динамический массив;
  - *list* - список;
  - *queue,deque* - очередь;
  - *stack* - стек;
  - *map, multimap* - отображения (ассоциативные массивы);
  - *set* - множество;
- *алгоритмы* - шаблоны функций для обработки элементов массивов и контейнерных классов;
- различные вспомогательные классы
  - *функциональные объекты* - классы для которых перегружена операция (), используется в алгоритмах;
  - *pair* - класс реализующий пару значений, используемый с отображениями;
  - *auto_ptr* - простой "умный" указатель.

Исключения

Механизм исключений позволяет легко отследить различные ошибки в программе.

Операторы:

- *try* - определяет блок, в котором необходимо отследить исключения.
- *throw* - вызывает исключение указанного типа.
- *catch()* - определяет блок обработки исключения указанного типа. Подобных блоков может быть несколько для каждого типа. Если же тип исключения не важен или нужно обработать исключения по умолчанию, то в качестве аргумента используется троеточие.

Пример

```
1.  #include <iostream>
2.  using namespace std;
3.  float divfunc(float a, float b)
4.  {
5.    if(b==0) throw 1;
6.    return a/b;
7.  }
8.  void main(void)
9.  {
10.   float a,b,c;
11.   cout<<"input a, b: ";
12.   cin>>a>>b;
13.   try
14.   {
15.     c=divfunc(a,b);
16.     // если b==0, то следующая строка не выполнится
17.     cout<<"c="<<c<<endl;
18.   }
19.   catch(...)
20.   {
21.     cout<<"exception"<<endl;
22.   }
23. }
```

Атрибут throw

- В некоторых компиляторах можно явно указать, что функция или метод могут вызвать исключение (другие компиляторы могут просто игнорировать такую конструкцию, не сообщая об ошибке синтаксиса).
- void func1(int a) **throw**(...) {}
- void func1(int a) **throw**() {}
- void func3(int a) **throw**(int) { if(a>5) **throw** 2; }

STL исключения

В STL определено несколько классов исключений. Например, в файле stdexcept объявлены стандартные исключения:

- **invalid_argument** - вызывается при передаче неправильного аргумента;
- **length_error** - вызывается при превышении размера данных;
- **out_of_range** - вызывается при выходе за допустимые границы;
- **overflow_error** - вызывается при переполнении.

Эти и другие классы являются потомками класса **exception**. Через виртуальный метод **what** можно получить дополнительную информацию об исключении.

```
class logic_error: public exception
{
  public:
    logic_error (const string& what_arg): str_(what_arg) { ; }
    virtual ~logic_error ();
    virtual const char * what () { return str_.data(); }
  private:
    string str_;
};
class invalid_argument: public logic_error
{
  public:
    invalid_argument (const string& what_arg): logic_error(what_arg) {;}
    virtual ~invalid_argument ();
};
```

Пример

```
1.  #include <iostream>
2.  #include <stdexcept>
3.  using namespace std;
4.  class Range
5.  {
6.    int r1,r2,pos;
7.    public:
8.      Range(int rr1,int rr2, int ppos) { r1=rr1; r2=rr2; pos=ppos; }
9.      void setPos(int ppos)
10.     {
11.       if(ppos<r1 ||ppos>r2)
12.         throw std::invalid_argument("setPos illegal argument");
13.       pos=ppos;
14.     }
15. };
16. void main(void)
17. {
18.   Range r(10,20,12);
19.   try
20.   {
21.     r.setPos(25);
22.   }
23.   catch(exception &e) { cout<<e.what(); }
24.   catch(...) { cout<<"unknown exception"<<endl; }
25. }
```

try без catch не работает (компилятор ругается)

Класс auto_ptr

Ограничения класса auto_ptr (простой "умный" указатель) :

- объектом может владеть только один указатель,
- объектом не может быть массив,
- нельзя использовать адресную арифметику.

Единственное назначение этого класса - *автоматизировать уничтожение* выделенной ранее памяти.

Данный класс используется, когда время существование выделенного объекта можно ограничить определенным блоком.

Делая код более безопасным, данные классы не наносят ущерб размеру или скорости программы.

Пример

```
1. #include <memory> // объявление шаблона класса auto_ptr
2. #include <iostream>
3. using namespace std;

4. // Внутри функции мы выделяем память для объекта типа int
5. // но не освобождаем ее явно оператором delete.
6. // Это делается автоматически.
7. void main(void)
8. {
9.   auto_ptr<int> aptr(new int(20));
10.  auto_ptr<int> aptr2;
11.  cout<<"*aptr="<<*aptr<<endl;
12.  aptr2=aptr; // теперь aptr не владеет никаким объектом
13.  cout<<"*aptr2="<<*aptr2<<endl;
14.}
```

```
~auto_ptr() { delete this->get(); }
```

Итераторы библиотеки STL

- Итераторы - удобная обертка для указателей, а выполнены они как шаблоны классов.
- "Обычный" указатель тоже можно считать итератором (очень примитивным).
- Удобства итераторов: автоматическое отслеживание размера типа, на который указывает итератор, автоматизированные операции инкремента и декремента для перехода от элемента к элементу и т.д.
- 2 важных правила работы с итераторами: получения итераторов и отслеживания значения "за пределом".
  - Метод begin() - возвращает итератор, указывающий на первый элемент данных
  - Метод end() - возврат значения "за пределом" (past-the-end).
- Итераторы: основные и вспомогательные.

Фрагменты определений

```
1.  template <class _Tp, class _Alloc>  class _Vector_base {
2.  public:
3.    ...
4.    _Vector_base(const _Alloc& __a)
5.      : _M_start(0), _M_finish(0), _M_end_of_storage(__a, 0) { }
6.    _Vector_base(size_t __n, const _Alloc& __a)
7.      : _M_start(0), _M_finish(0), _M_end_of_storage(__a, 0)
8.    {
9.      _M_start = _M_end_of_storage.allocate(__n);
10.     _M_finish = _M_start;
11.     _M_end_of_storage._M_data = _M_start + __n;
12.   }
13.   ~_Vector_base() { ... }
14. protected:
15.   _Tp* _M_start;
16.   _Tp* _M_finish;
17.   ...
18. };
19. template <class _Tp> class vector : public _Vector_base<_Tp, _Alloc>  {
20.   ...
21. typedef _Vector_base<_Tp, _Alloc> _Base;
22.   ...
23. public:
24.   iterator begin() { return this->_M_start; }
25.   iterator end()  { return this->_M_finish; }
26.   ...
27. }
```

Итераторы ввода

- Наиболее простые из всех итераторов STL, и доступны они только для чтения.
- Оператор разыменовывания (*) для прочтения содержимого объекта, на который итератор указывает.
- Оператор (++) - перемещение от первого элемента, на который указывает итератор ввода, к следующему.
- Итераторы ввода возвращает только шаблонный класс istream_iterator.

Алгоритм for_each

```
template <class InputIterator, class Function>
Function for_each (InputIterator first, InputIterator last, Function f)
{  while (first != last) f(*first++);    return f; }
```

Или так:

```
template <class InputIter, class Function>
Function for_each(InputIter first, InputIter last, Function f)
{  for ( ; first != last; ++first)    f(*first);  return f; }
```

```
1. #include <iostream>
2. #include <algorithm>
3. using namespace std;
4. void printValue(int num) { cout << num << "\n"; }
5. main(void)
6. {  int init[] = {1, 2, 3, 4, 5};
7.    for_each(init, init + 5, printValue);
8. }
```

Итераторы вывода

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <iterator>
using namespace std;
main(void)
{  int init1[] = {1, 2, 3, 4, 5};
   int init2[] = {6, 7, 8, 9, 10};
  vector<int> v(10);
  merge(init1, init1 + 5, init2, init2 + 5, v.begin());
  copy(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));
}
```

Однонаправленные итераторы

Если соединить итераторы ввода и вывода, то получится однонаправленный итератор (forward iterator), который может перемещаться по цепочке объектов в одном направлении. Для такого перемещения в итераторе определена операция инкремента (++).

```
template <class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last, const T&
    old_value,  const T& new_value)
{ while (first != last)
   {  if (*first == old_value) *first = new_value;  ++first;  }}

template <class ForwardIter, class Tp>
void replace(ForwardIter first, ForwardIter last,  const Tp& old_value,
    const Tp& new_value)
{  for ( ; first != last; ++first)
     if (*first == old_value)  *first = new_value; }
```

Пример

```
1. #include <algorithm>
2. #include <iostream>
3. #include <vector>
4. #include <iterator>
5. using namespace std;
6. main(void)
7. {
8.   int init[] = {1, 2, 3, 4, 5};
9.   replace(init, init + 5, 0, 2);
10.  replace(init, init + 5, 1, 0);
11.  replace(init, init + 5, 2, 1);
12.  copy(init, init + 5, ostream_iterator<int>(cout, "\n"));
13.}
```

Двунаправленные итераторы

```
1. #include <algorithm>
2. #include <iostream>
3. #include <vector>
4. #include <iterator>
5. using namespace std;
6. main(void)
7. {
8.   int init[] = {1, 2, 3, 4, 5};
9.   reverse(init, init + 5);
10.  copy(init, init + 5, ostream_iterator<int>(cout, "\n"));
11.}
```

Итераторы произвольного доступа

```
1.  #include <algorithm>
2.  #include <iostream>
3.  #include <vector>
4.  using namespace std;
5.  void main(void)
6.  {
7.    const int init[] = {1, 2, 3, 4, 5};
8.    vector<int> v(5);
9.    typedef vector<int>::iterator vectItr;
10.  vectItr itr;
11.  copy(init, init + 5, itr = v.begin());
12.  cout << *( itr + 4 ) << endl;
13.  cout << *( itr += 3 ) << endl;
14.  cout << *( itr -= 1) << endl;
15.  cout << *( itr = itr - 1) << endl;
16.  cout << *( --itr ) << endl;
17.  for(int i = 0; i < (v.end() - v.begin()); i++)    cout << itr[i] << " ";
18.}
```

Итераторы потоков

```
1. #include <algorithm>
2. #include <iostream>
3. #include <vector>
4. #include <iterator>
5. using namespace std;
6. main(void)
7. {
8.   istream_iterator<int> is(cin);
9.   ostream_iterator<int> os(cout, " - last entered value\n");
10. int input;
11. while((input = *is) != 666)
12. {
13.    *os++ = input;
14.     is++ ;
15. }
16.}
```

Итераторы вставки

```
1.  #include <algorithm>
2.  #include <iostream>
3.  #include <list>
4.  using namespace std;
5.  main(void)
6.  {  int init[]  = {0, 0};  int init1[] = {3, 2, 1};
7.     int init2[] = {1, 2, 3};  int init3[] = {1, 1, 1};
8.     list<int> l(2);
9.     copy(init, init + 2, l.begin());
10.    copy(l.begin(), l.end(), ostream_iterator<int>(cout, " "));
11.    cout << " - before front_inserter" << endl;
12.    copy(init1, init1 + 3, front_inserter(l));
13.    copy(l.begin(), l.end(), ostream_iterator<int>(cout, " "));
14.    cout << " - before back_inserter" << endl;
15.    copy(init2, init2 + 3, back_inserter(l));
16.    copy(l.begin(), l.end(), ostream_iterator<int>(cout, " "));
17.    cout << " - before inserter" << endl;
18.    list<int>::iterator& itr = l.begin();
19.    advance(itr, 4);
20.    copy(init3, init3 + 3, inserter(l, itr));
21.    copy(l.begin(), l.end(), ostream_iterator<int>(cout, " "));
22. }
```