SFU CMPT-212 2008-1 Topic: Function Objects

Ján Maňuch

E-mail: jmanuch@sfu.ca

Friday 14th March, 2008

Function objects (functors)

Types and their concepts:

- a *generator* — a function object that can be called with no arguments
- a *unary function* — a function object that can be called with one argument
- a *binary function* — . . . with two arguments
- a *unary predicate* — a unary function object returning `bool`
- a *binary predicate* — a binary function object returning `bool`

Models:

- pointer to function
- object that can be used with `operator()`

Example of use:

- *generator:* can be used in the generic algorithm `generate`:

```
1 void generate(ForwardIterator b, ForwardIterator e,
2                Generator g);
```

  for each element in the interval `[b,e)`, it will call `g()` and assign the returned value to the element

  Example: generator.cpp

```
1 vector<long> v(10);
2 generate(v.begin(),v.end(),rand);
3 // fill v with 10 random values
```

- *unary function:*

  Recall:

  – without returned value, can be used with generic algorithm `for_each` (foreach.cpp)

  – with returned value, can be use with `transform` (transform.cpp)

- *binary function:* can be used with another form of `transform` algorithm

```
1 transform(InputIterator1 b1, InputIterator1 e1,
2           InputIterator2 b2, OutputIterator result,
3           BinaryOperation op);
```

which will read elements from interval `[b1,e1)` and at the same time elements starting from `b2`, apply `op` on them and write the result starting from `result`

Example:

```
1 int add(int x,int y) { return x+y; }
2 ...
3 int a[]={1,2,3,4,5,6,7,8,9,10};
4 int b[5];
5 transform(a,a+5,a+5,b,add);
6 // b: [7,9,11,13,15]
```

- *unary predicate:* can be used with the generic algorithm `remove_if`:

```
1 remove_if(Forward Iterator b, Forward Iterator e,
2           Predicate p)
```

— removes all elements in interval `[b,e)` which satisfy the predicate `p`

- *binary predicate:* can be used with extended version of `sort`:

```
1 sort(RandomAccessIterator b, RandomAccessIterator e,
2      BinPredicate compare);
```

— sorts the elements in interval `[b,e)` by uses predicate `compare` instead of `operator<`

Predefined functors

- it might be troublesome to declare a new function or a function object class each time we need to use some different functor in generic algorithms
- STL provides elementary function objects for basic arithmetic operations (as template classes) in the header file `<functional>`

  Example: `multiplies<double>()` is a binary functor multiplying two doubles:

```
1 cout << multiplies<double>()(3.3,3.0); // prints 9.9
```

- *Q.* why is there an extra pair of parenthesis `()`?
- *A.* `multiplies` is a template class, `multiplies<double>` is an instantiation of the class and `multiplies<double>()` is a *function object*

Example of implementation:

```
1  template <typename T>
2  class multiplies {
3   public:
4    T operator()(const T& x, const T& y) const
5    { return x*y; }
6  };
```

Example of usage:

```
1  list<double> v;
2  // fill data ...
3  double product=accumulate(v.begin(),v.end(),
4                            1.0,multiplies<double>());
```

— computes product of all elements in `v`

Remarks:

- `multiplies` works fine with built-in types or any user-defined types which overload `operator*()`

- if your class has overloaded a *non-member* `operator*()`, you can use it directly:

```
1 Point product=accumulate(v.begin(),v.end(),
2                          Point(1,0),operator*);
```

  Example: point.cpp

- the list of predefined function objects:
  - *unary functions:* `negate`
  - *binary functions:* `plus`, `minus`, `multiplies`, `divides`, `modulus`
  - *unary predicates:* `logical_not`
  - *binary predicates:* `equal_to`, `not_equal_to`, `greater`, `less`, `greater_equal`, `less_equal`, `logical_and`, `logical_or`
- *Problem:* How to multiply each element of the container by `2` or increment it by `1`?

- *Solution:* we need to fix one parameter in the predefined function objects
- STL provides `binder1st` and `binder2nd` template classes doing that:

```
1 binder1st<multiplies<double> > times_two(multiplies<int>(),2);
2 transform(v.begin(),v.end(),v.begin(),times_two);
```

  - a template parameter to `binder1st` class is the name of the functor class (`multiplies<double>`)
  - `times_two` is the name of the function object constructed with two parameters: function object of the class passed as parameter (`multiplies<double>()`) and the value of the first parameter to be fixed

- easier to use template functions `bind1st()` and `bind2nd()` which will guess the template parameter

```
1 transform(v.begin(),v.end(),v.begin(),bind2nd(plus<int>(),1));
```

  Example: bind.cpp, pointbind.cpp

Sample questions

- What types of function objects there are? What are the models for function objects? Show an example for each.
- What is the following code doing?

```
1 int add(int x,int y) { return x+y; }
2 ...
3 int a[]={1,2,3,4,5,6,7,8,9,10};
4 transform(a,a+5,a+5,a+2,add);
```

  What are the values of array `a` after execution of the above fragment?
- Given a `vector<int> v1`, construct another `vector<int> v2` with the same size an elements multiplied by `2` and incremented by `1`. For example, if `v1` contains elements `1,3,2`, at the end `v2` should contain `3,7,5`.
- What is the difference between `binder1st` and `bind1st`?