Modern C++ - STL

17. The Standard Template Library

The *Standard Template Library* (STL) is part of the C++ standard. The STL is the most important example for generic programming.

Generic programming: make more abstract routines without loosing efficiency using parameterization (both data and algorithm).

Generics

Suppose we want to find an element in an array of integers:

```
int t[] = { 1, 3, 5, ... };
const int len =  t+sizeof(t)/sizeof(t[0]);

// find the first occurance of a value
int *pi = find( t, t+len, 55);

if ( t+len != pi )
{
  *pi = 56;
}
```

For this purpose we can define a very specific implementation. We walk throu the interval and check every elements until we find the value we are looking for, or we find the end of the interval.

Consider the arguments of the **find** function: the **begin** pointer specifies the beginning of the interval, but **end** points the place *after the last* element. Thi sway the empty interval can be expressed more easily.

```
int *find( int *begin, int *end, int x)
{
  while ( begin != end )
  {
    if ( *begin == x )
    {
      return begin;
    }
    ++begin;
  }
  return 0;
}
```

This function works, but too specific. It can be applied only for arrays of integers. We will *generalize* the function: we change it to accept more types, without compromising performance. Templates solve the problem:

```
template <typename T>
T *find( T *begin, T *end, const T& x)
{
  while ( begin != end )
  {
    if ( *begin == x )
    {
      return begin;
    }
    ++begin;
  }
  return 0;
}
```

This version of find works on array of doubles or even on strings. However, we are still restricted our algorithm to arrays.

Now we generalize on *data structure*. The idea is that on every data structure we can follow the same *pseudo code*:

1. Take the first element (*begin*)
2. Chack whether we reached the end of the interval (begin != end)
3. If not, check whether this is the element we are looking for (*begin == x)
4. If not, go to the next element of the interval (++begin)

What data structure dependent is how to reach the current element and how to go to the next element of the interval. To implement these concepts we introduce a new template type, the *iterator*, which is data structure dependent, and responsible to implement these operations.

```
template <typename It, typename T>
It find( It begin, It end, const T& x)
{
  while ( begin != end )
  {
    if ( *begin == x )
    {
      return begin;
    }
    ++begin;
  }
  return end;  // not 0
}
```

Notice, that we return either an iterator referring to the element we found or **end**, the iterator value referring outside of the range. It is more general and still easy to check criteria to inform the user that the searched element was not found.

An iterator is the abstraction of the “pointer” concept, however, implemented usually as a (nested) class in a usually templated container. Now, we can use our find function in a data structure independent way:

```
// use with integer array
int t[] = { 1, 3, 5, ... };
const int len = t+sizeof(t)/sizeof(t[0]);

int *pi = find( t, t+len, 55);

if ( t+len != pi )
{
  *pi = 56;
}

// use with vector of doubles
vector<double> v(t,t+len);  // initialize from int array

vector<double>::iterator vi = find( v.begin(), v.end(), 55.0);
if ( v.end() != vi )
{
  *vi = 56.66;
}

// use with list of doubles
list<double> l(v.begin(),v.end()); // initialize from vector<double>
```

```

list<double>::iterator li = find( l.begin(), l.end(), 55.55);

if ( l.end() != li )
{
  *li = 56.66;
}
```

Iterators are also have *const* version: **const_iterator**. The const_iterator behaves like a *pointer to const*, it is not a contant itself, but the referred memory is inmutable. Do not mix **const iterator** with **const_iterator**.

```
const vector<int> cv = {1,2,3,4,5};  // initialization, C++11

vectort<int>::const_iterator cvi = find( cv.begin(), cv.end(), 55.55);

if ( cv.end() != cvi )
{
  // *cvi = 56;  // Syntax error, the referred memory is immutable
  cout << *cvi;  // ok, read
}
```

In C++11, we have a shortcut to declare iterators using automatic *type deduction*.

```
vector<int> v = {1,2,3,4,5};              // initialization, C++11
const list<double> cl(v.begin(), v.end()) // initialization

// vector<int>::iterator
auto  vi = find(  v.begin(),  v.end(), 55);

// list<double>::const_iterator
auto cli = find( cl.begin(), cl.end(), 55.55);
```

Functors

When we want to execute more complex algorithms, we get some problems. Suppose, we want to find the *third* occurance of an element *smaller* than 55. This is not trivial with the original **find** algorithm.

There is an other *find* version: **find_if** which searches not for a concrete element, but the first element where a *predicate is true*:

```
template <typename It, typename Pred>
It find_if( It begin, It end, Pred p)
{
  while ( begin != end )
  {
    if ( p(*begin) )
    {
      return begin;
    }
    ++begin;
  }
  return end;
}
```

To use **find_if** we first have to define a *predicate*, a function taking one parameter and return **true** when this parameter satisfy the required criteria.

The most simple (but far from perfect) implementation for the predicate is a simple function, which returns true when called the third time with an argument less than 55.

```
// Pred1: not too good
bool less55_3rd( int x)
{
  static int cnt = 0;
  if ( x < 55 )
    ++cnt;
  return 3 == cnt;
}
```

It seems working when we call first:

```
vector<int> v = { 3, 99, 56, 44, 2, 8, 1, 5, 88, 6, 9};
auto i = find_if( v.begin(), v.end(), less55_3rd);
// *i == 2
```

But in real situations this is a wrong implementation:

```
vector<int> v = { 3, 99, 56, 44, 2, 8, 1, 5, 88, 6, 9};
auto i = find_if( v.begin(), v.end(), less55_3rd);
// *i == 2
i = find_if( v.begin(), v.end(), less55_3rd);
```

```
// i == v.end()
```

The problem is, that **less55_3rd** uses a local static variable **cnt** for counting, for functions this is the only feasible solution to remember the previous value of a local variable. Unfortunately, a static variable will keep its value between the separate calls of **find_if** too.

A further problem of static variables – locals or globals – is that they are also shared between parallel execution threads in a possible multithreaded application.

We need a solution in which we can allocate a separate memory for each applications of the function, separated from all – concurrent, or different in in time – applications of the same function. *Classes* do exactly this: data members provide separate memory space for each objects, while member functions provide the functionality.

```
struct less55_3rd
{
  less55_3rd() : cnt(0) { }
  bool operator()(int x)
  {
    if ( x < 55 )
      ++cnt;
    return 3 == cnt;
  }
private:
  int cnt;
};
```

The constructor initializes the **cnt** data member which conts the hits. The function call operator provides the actual checks – called on every data items on the range by the **find_if** algorithm. For each application of **less55_3rd** we should create a new object. Instead of declare a variable with name, it is enough to create a temporary variable.

```
vector<int> v = { 3, 99, 56, 44, 2, 8, 1, 5, 88, 6, 9};
auto i = find_if( v.begin(), v.end(), less55_3rd);
// *i == 2
i = find_if( v.begin(), v.end(), less55_3rd);
// *i == 2
i = find_if( ++i, v.end(), less55_3rd);
// *i == 5
i = find_if( ++i, v.end(), less55_3rd);
// i == v.end()      // no more findings
```

Functors are flexible way to define arbitrary search, sort or other criteria by the user.

Remark: there is not always a good idea to create a functor with state. Some algorithms may copy or assign the functor parameter, therefore the state may changed differently as one suppose it.

One can generalize the **less55_3rd** functor to make it more generally usable. We can parameterize both the 55 and the 3 numeric parameters: instread of “wired-in” to the code they can be constructor parameters. Also we can generalize the types the functor works on.

```
template <typename T>
struct less_nth
{
  less_nth( const T& t, int n) : t_(t), n_(n), cnt_(0) { }
  bool operator()(const T& t)
  {
    if ( t < t_ )
      ++cnt;
    return n_ == cnt;
  }
private:
  T   t_;
  int n_;
  int cnt_;
};

vector<int> v = { ... };
auto i = find_if(v.begin(),v.end(),less_nth<int>(55,3));
```

Further generalization can be replace the **operator<** used in the function call operator with a functor parameter used to compare two elements with the *less* criteria.

Defining functors is a bit heavy weight solution since we have to write some boiler-plate code. The *lambda functions* can be used to replace functors with a much less syntactical overhead.