22 - Lecture - Generic programming: templates and STL

Recommended reading

Lippman, 5th Ed.:

- 3.1 - 3.4: string, vector, iterators
- 16.1 - 16.3: templates
- 9.1 - 9.4: sequential containers
- 11.1 - 11.3: associative containers: map

Lippman, 4th Ed.:

- 3.1 - 3.4: string, vector, iterators
- 16.1 - 16.4: templates
- 9.1 - 9.5: sequential containers
- 10.1 - 10.3: associative container: map

Temlate function

Consider max() function:

```
int max(int x, int y) { if (x<y) return y; else return x; }
```

But what about max(1.1, 1.2) or max("abc", "def")?
Do we need to write max(double, double) and max(MyString, MyString)?
Can we get away with writing just one function?
Yes, we write a template function:

```
template <typename T>
T max(T x, T y)
{
    if (x<y) return y; else return x;
}
```

Or even better:

```
template <typename T>
const T& max(const T& x, const T& y)
{
    if (x<y) return y; else return x;
}
```

(You should know why this is better if you did the MyString lab.)

The compiler uses the template definition to generate the "typed instances" of the template function as needed. For example, if the compiler encounters max(1.1, 1.2), it'll generate the following code:

```
    double max(double x, double y)
    {
        if (x<y) return y; else return x;
    }
```

Template class

Consider a class implementing a dynamic array of integers that grows as you put elements into it:

```
    class IntArray {
      public:

        ......

        void push_back(int x) {
          if (++size > capacity) {
            // allocate new memory with 2 * capacity
            // copy over the old array
            // delete the old array
          }
          // assign x at the end of the array
        }

        ......

      private:

        int *a;
        size_t size;
        size_t capacity;
    };
```

Just as we wrote max() as a template function to apply the same logic to various different types, we can turn IntArray into a template class:

```
    template <typename T>
    class TArray {
      public:

        ......

        void push_back(const T& x);

        ......

      private:
        T *a;
        size_t size;
        size_t capacity;
    };
```

And here is how you use it:

```
    TArray<double> a;

    a.push_back(1.5);

    ......
```

In fact, you have exactly this in the standard library, and it's called "vector".

Before we move on to the standard library, let's compare template with void*:

```
    template                    void*
    ------------------------------------------------------------
    generic programming         typeless programming

    type-safe                   type-unsafe
      - compiler catches          - seg-fault catches
        type mismatches             type mismatches

    enables value semantics     forced to use pointer semantics
                                  - need to manage object life-time
```

"vector" class in the standard library

Basic usage:

```
    vector<MyString> v;

    v.push_back("hello");
    v.push_back(" ");
    v.push_back("world");

    cout << v[0] << v[1] << v[2] << endl;
```

You can have a vector of vectors:

```
    vector<vector<double> > matrix;
```

Note:

- It's just an array under the hood.
- vector<T> contains objects of type T "by value", that is, when you push_back(x), a copy of x is constructed (by calling the copy constructor of T) and held in the vector. The elements are destroyed along with the vector when the vector gets destroyed.
- Provides efficient random access (operator[]) and adding/removing elements from the end of the vector (push_back()/pop_back()).
- Supports item additions & removals at positions other than the end of the vector (insert()/remove()), but they are inefficient because other items need to be moved.

"string" class in the standard library

- Has all the features of our MyString class, plus much, much more.

- Highly optimized implementation, but MyString is a good mental model for it.

- "string" is a template too! It's a typedef of something like:

```
std::basic_string<char, std::char_traits<char>, std::allocator<char> >
```

  (You'll see a lot of these crazy-looking template instances in the compiler error messages.)

Other sequential containers: deque & list

deque:

- short for double-ended queue, pronounced "deck"

- similar to vector, but unlike vector, deque supports efficient addition & removal of elements from the beginning of the deque (push_front()/pop_front()).

- slower and/or wastes more memory than vector

list:

- doubly linked list

- efficient insertion & removal of elements anywhere in the list

- no random access

All sequential containers follow a common interface to the extent possible.

Iterators

Generalization of pointers for standard containers.

vector example:

```
for (vector<string>::iterator it = v.begin(); it != v.end(); ++it)
    *it = "";
```

const version:

```
for (vector<string>::const_iterator it = v.begin(); it != v.end(); ++it)
    cout << *it << endl;
```

Note:

- "iterator" and "const_iterator" is a "type member" (i.e., a typedef defined inside a class)

- An iterator behaves like a pointer: ++it, *it. Sometimes it's actully a pointer (vector<T>::iterator is just a typedef of T*), but more often it's not. (Think about what a list<T>::iterator would have to do when you increment it.)

- Dereferencing an iterator gives you a T&, and dereferencing a const_iterator gives you a const T&.

- v.begin() returns an iterator that points to the first element in the vector.

- v.end() returns an iterator that points to ONE PAST THE LAST ELEMENT, not the last element. It is used as a marker for the end of the container, and it cannot be dereferenced.

- Note that we use "!=" rather than "<" in the for loop. This is recommended (especially if you're writing template code) because not all iterators define operator<(). (list iterator does not, for example.)

- Note that we use ++it rather than it++. This is recommended because when the iterator is a class object rather than a native pointer, it's usually less costly to call ++it than it++.

Iterators and generic algorithms

The standard C++ library provides many useful template functions that implement common algorithms such as sorting, finding an element, reversing container contents, etc. (See A.2 in Lippman.)

They usually take as parameters two iterators, b and e, that denote a range of elements [b, e). That is, they denote a range starting from (and including) *b, and up to (but NOT including) *e. Note that this is consistent with the definition of v.begin() and v.end() above.

For example, reverse(b,e) algorithm function reverses all elements in [b,e). You can call it with vector iterators as well as with list iterators. In fact, any iterators that can go back and forth, which are called "bidirectional iterators".

That brings us to iterator categories. Iterators are grouped into 5 categories depending on the operations they support (see Lippman p416-418):

- Input iterators
- Output iterators
- Forward iterators
- Bidirectional iterators
- Random-access iterators

The later ones implement more operations and include all operations of the previuos ones (except the Input and Output iterators, which are just different.) list iterators are bidirectional iterators, and vector or deque iterators are random-access iterators.

Concrete example

```
/*
 * stack.h
 */

#ifndef __STACK_H__
#define __STACK_H__

using namespace std;
#include <deque>
#include <algorithm>
#include <iostream>

template <typename T>
class Stack;
template <typename T>
ostream& operator<<(ostream& os, const Stack<T>& rhs);

template <typename T>
class Stack
{
    public:

        bool empty() const { return q.empty(); }

        void push(const T& t) { q.push_back(t); }

        T pop();

        void reverse();

        friend ostream& operator<< <T>(ostream& os, const Stack<T>& rhs);

    private:

        deque<T> q;
};

template <typename T>
T Stack<T>::pop()
{
    T t = q.back();
    q.pop_back();
    return t;
}

template <typename T>
void Stack<T>::reverse()
{
    ::reverse(q.begin(), q.end());
}

template <typename T>
ostream& operator<<(ostream& os, const Stack<T>& rhs)
```

```
{
    os << "[ ";
    typename deque<T>::const_iterator i;
    for (i = rhs.q.begin(); i != rhs.q.end(); ++i)
        os << *i << " ";
    os << "<";
    return os;
}

#endif


/*
 * test1.cpp - test driver for stack.h
 */

using namespace std;
#include <string>
#include "stack.h"

int main()
{
    Stack<string> s;

    s.push("one");
    s.push("two");
    s.push("three");
    s.push("four");

    while (!s.empty()) {
        cout << s << endl;
        cout << "popped " << s.pop() << endl;
        s.reverse();
        cout << "reversed the stack: " << s << endl;
        cout << endl;
    }

    return 0;
}


Associative container in C++ standard library: map
--------------------------------------------------

"pair" type:

    template <class T1, class T2>
    struct pair {
        T1 first;
        T2 second;
    };

    pair<string,int>  prez("obama", 1961);
    pair<string,int>  veep("biden", 1942);

map:

  - a map is a collection of key-value pairs.
```

- example: counting the occurences of each word read from stdin

```
map<string,int>  word_count;
string word;
while (cin >> word)
    word_count[word]++;

map<string,int>::iterator it;
for (it = word_count.begin(); it != word_count.end(); ++it) {
    cout << it->first << " occurs ";
    cout << it->second << " times." << endl;
}
```