Introduction to Templates and Standard Template Library

Function Templates

Function overloading allows the same function to act on different argument types:

```
void swap2Items(int &n1, int &n2){
       int temp = n2;
       n2 = n1;
       n1 = temp;
}
void swap2Items(card &n1,card &n2){
       card temp = n2;
       n2 = n1;
       n1 = temp;
}
```

Function Templates

Function written as a template:

```
template <class T>
void swap2Items(T &n1, T &n2) {
   T temp = n2;
   n2 = n1;
   n1 = temp;
}
```

Compiler generates code for us when it comes across statements:

```
swap2Items(j, k);   //i  and j  are ints
swap2Items(c1, c2); //c1 and c2 are cards
```

Function Templates

This function will work with any defined type:

```
Date d1, d2;
swap2Items(d1,d2);
```

With what condition?

Function Templates

This function will work with any defined type:

```
Date d1, d2;
swap2Items(d1,d2);
```

With what condition?

Date class must have a valid assignment operator.

```
operator=
```

Function Templates

Function templates can even be overloaded

```
template <class T, class S>
void swap2Items(T &x, S &y) {
   T temp = x;
   x = y;
   y = temp;
}
```

Which allows us to swap any object with any other object as long as we can assign between the two types. This now replaces the previous version with the special case S=T.

Class Templates

Templates can also be used with classes. We can create generic classes that handle different types of data.

A stack ADT is a FILO type.

For the stack class:

We may want stacks of integer, float, Date etc

Without templates we have to create stacks for each of them,

With templates we can create generic stacks

Class Templates

A stack ADT is a FILO type. Here is the class declaration using templates:

```
template <class T>
class StackT{
public:
   StackT(int = 10) ;
   ~StackT() { delete [] stackPtr; }
   int push(const T&);
   int pop(T&) ;
   int isEmpty()const { return top == -1; }
   int isFull() const { return top == size - 1; }
private:
   int size;
   int top;
   T* stackPtr;
};
```

Class Templates

Template class constructor:

```
template <class T>
StackT<T>::StackT(int s){
   size = s > 0 && s < 1000 ? s : 10;
   top = -1;  //initialize stack
   stackPtr = new T[size];
}
```

Class Templates

```
template <class T>
int StackT<T>::push(const T& item){
   if (!isFull()){
      stackPtr[++top] = item;
      return 1;  // push successful
   }
   return 0;     // push unsuccessful
}
template <class T>
int StackT<T>::pop(T& popValue) {
   if (!isEmpty()){
      popValue = stackPtr[top--] ;
      return 1;  // pop successful
   }
   return 0;     // pop unsuccessful
}
```

Class Templates

Code for these template classes and functions are generated when they are used in code.

```
int main() {
   StackT<float> fs(5);
   float f = 1.1;

   while(fs.push(f)) {
      cout << f << ' ';
      f += 1.1;
   }
}
```

Templates vs Macros

Macros present the possibility for side effects and avoid type checking.

```
#define SQ(A) ((A)*(A))

template<class T> T square (T x){
        return x*x;
}
```

Templates vs Macros

```
int main(){
   int a = 7;
   cout << square(a++)       << endl;
   cout << "and a is " << a << endl;
   cout << SQ(a++)           << endl;
   cout << "and a is " << a << endl;
}
```

output:

```
49
and a is 8
64
and a is 10
```

Common Errors

1. Not using template<**class** T> when defining member function for class templates.
2. Not using the generic type for parameters/ variables.
3. Not placing class, before every formal type parameter. Correct: template<**class** T, **class** S>
4. If a template is invoked with a user defined class type and that template uses operators (==, +, <=) with objects of that class type, then those **operator must be overloaded**.

Standard Template Library

A large collection of reusable components.

The key components of the STL - **containers** are data structures created using templates.

A container is an object that contain objects.

Using STL can save considerable time and effort, and result in higher quality programs.

Standard Template Library

STL is a set of general-purpose template classes, built using the template mechanism. The main ideas of STL are:

- Containers
- Iterators
- Algorithms

Standard Template Library

Containers are objects that hold other objects. There are different types of containers:

sequence containers:

```
vector, deque, list
```

associative containers for allowing efficient retrieval of values based on keys:

```
map, sets
```

Vector Class

Vectors are dynamic arrays: arrays that can grow as needed.

```
template <class T>
class vector{
  //...
};
```

When using the vector type the required header to be included is <`vector`>

Vector Class

Declaring/Defining a vector:

```
vector<int> iv;
vector<char> cv(5);
vector<char> cv(5,  'x');
vector<int> iv2(iv);
```

Vector Class

Using a vector:

```
// display original size of v
cout<<"Size = "<<v.size()<<endl;

// put values onto end of a vector
// the vector will grow as needed
for(int i = 0; i < 10; i++) {
   v.push_back(i);
}
// change contents of a vector
for(int i = 0; i < v.size(); i++) {
   v[i] = v[i] + v[i];
}
```

Vector Class

```
// access using subscripting
for(int i = 0; i < 10; i++) {
   cout << v[i] << " ";
}
cout << endl;

// access via iterator
vector<char>::iterator p = v.begin();
while(p != v.end()) {
   cout << *p << " ";
   p++;
}
```

Iterators

An Iterator is just a convenient pseudo-pointer that is set up as a type to use associated with each container class. Various operators will have been set up to use with them eg =, ==, != and +=

Iterators

```
int array[10];
for(int i = 0; i < 10; i++) {

}
```

Becomes

```
vector<int> v(10);
vector<int>::iterator it;
for(it = v.begin(); it != v.end(); it++) {

}
```

Iterators

Iterators are **objects** similar to **pointers**.

They are design to give us the ability to cycle through the content of a container.

There are five iterator types:

- input
- output
- forward
- bidirectional
- random access

Iterators

Container classes and algorithms dictate the category of iterator available or needed.

- **vector** containers allow random-access iterators
- **lists** do not;

- **sorting** requires a random-access iterators
- **find** requires an input iterator.

Iterators

Iterators can be adapted to provide backward traversal.

Example that uses a reverse iterator to traverse a sequence.

```
template <class ForwIter>
void print(ForwIter first, ForwIter last,
           const char* title){
   cout << title << endl;
   while ( first != last)
      cout << *first++ << '\t';
   cout << endl;
}
```

Iterators

```
int main(){
   int data[3] = { 9, 10, 11};

   vector<int> d(data, data + 3);
   vector<int>::reverse_iterator p = d.rbegin();
   print(p, d.rend(), "Reverse");
   //...
}
```

STL Standard Components

STL relies upon several other standard components for support:

allocators: to manage memory allocation for a container

predicates: special functions returning true/false results

comparison functions, etc.

STL Algorithms

Algorithms act on the contents of containers.

They include capabilities for:

- initializing
- sorting
- searching and
- transforming the contents of containers.

All algorithms are template functions.

To access them: `#include <algorithms>`

STL Algorithms

```
string words[5] = {"my", "hop", "mop", "hope”, "cope"};

string* where;
where = find(words, words + 5, "hop");

cout << *++where << endl; //mop
```

STL Algorithms

```
vector<int> v;
int i;

for(i = 0; i < 10; ++i) v.push_back(i);

cout << "Initial: ";
for(i = 0; i < v.size(); ++i)
      cout << v[i] << " ";
cout << endl;

reverse(v.begin(), v.end());
```

Transform Reverse Algorithms

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

double reciprocal( double d){  return 1.0 / d; }

template <class T>
void printVector( vector<T> v ){
  for(int j=0;j<v.size();j++){
     cout << v[j] << " ";
  }
  cout << endl;
}

int main(){
  vector<double> vals;
  for(int i=1;i<10;i++) vals.push_back((double)i);

  printVector( vals );
  transform( vals.begin(), vals.end(),
             vals.begin(), reciprocal );
  printVector( vals );
  reverse( vals.begin(), vals.end() );
  printVector( vals );
  return 0;
}
```

Output

```
1 2 3 4 5 6 7 8 9
1 0.5 0.333333 0.25 0.2 0.166667 0.142857 0.125 0.111111
0.111111 0.125 0.142857 0.166667 0.2 0.25 0.333333 0.5 1
```

Note use of function name identifier as an argument to transform

transform takes arguments: start-iter, end-iter, result-iter, func

Note use of template mechanism to write a generalised printVector function

Merge Algorithm

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

template <class T>
void printVector( vector<T> v ){
  for(int j=0;j<v.size();j++){
    cout << v[j] << " ";
  }
  cout << endl;
}
int main(){
  vector<int> v1;
  for(int i=1;i<20;i+=2) v1.push_back( i );
  vector<int> v2;
  for(int i=0;i<20;i+=2) v2.push_back( i );
  printVector( v1 );
  printVector( v2 );
  vector<int> v3( v1.size() + v2.size() );
  printVector( v3 );
  merge( v1.begin(), v1.end(), v2.begin(), v2.end(), v3.begin() );
  printVector( v3 );
  return 0;
}
```

Output

```
1 3 5 7 9 11 13 15 17 19
0 2 4 6 8 10 12 14 16 18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19
```

Note arguments:

- start1, end1, start2, end2, result

Returns end-iter of result (not used in this example)

STL Algorithms

```
  merge(v1.begin(),v1.end(),v2.begin(),v2.end(),v3.begin());
  printVector( v3 );
  random_shuffle( v3.begin(), v3.end() );
  printVector( v3 );
  cout << "end  - begin = " << v3.end() - v3.begin() << endl;
  cout << "Max is " << *max_element(v3.begin(), v3.end()) << endl;
  cout << "Min is " << *min_element(v3.begin(), v3.end()) << endl;
  sort(v3.begin(), v3.end());
  printVector( v3 );
  for_each(v3.begin(), v3.end(), sum);
  cout << "Sum  was " << total << endl;
  vector<int> v4;
  v4.push_back(11);
  v4.push_back(12);
  v4.push_back(13);
  cout << "subsequence  included is " <<
           includes(v3.begin(),v3.end(),v4.begin(),v4.end()) << endl;
  v4[1] = 10;
  cout << "subsequence  included is " <<
           includes(v3.begin(),v3.end(),v4.begin(),v4.end()) << endl;
  return 0;
}
```

STL Algorithms

adjacent_find
binary_search
copy
copy_backward
count
count_if
equal
equal_range
fill and fill_n
find
find_end
find_first_of
find_if
for_each
generate and generate_n
includes
inplace_merge
iter_swap
lexicographical_compare
lower_bound
make_heap
max
max_element
merge
min
min_element
mismatch
next_permutation

STL Algorithms

nth_element
partial_sort
partial_sort_copy
partition
pop_heap
prev_permutation
push_heap
random_shuffle
remove and remove_if…
replace and replace_if…
reverse and reverse_copy
rotate and rotate_copy
search
search_n
set_difference
set_intersection
set_symmetric_difference
set_union
sort
sort_heap
stable_partition
stable_sort
swap
swap_ranges
transform
unique and unique_copy
upper_bound

STL Summary

Algorithms in the STL are just template functions

There are some useful ones that may save you reinventing the wheel.

Library functions have the great advantage - someone else has tested them!