NWEN 241
Systems Programming

Sue Chard

suechard@ecs.vuw.ac.nz

Content

- Midterm
- Templates C++
- Vectors C++

Templates and Vectors

- C++ only
- Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type
- A template is a blueprint or formula for creating a generic function or a generic class
- In 1990, Alex Stepanov and Meng Lee of Hewlett Packard Laboratories extended C++ with a library of class and function templates which has come to be known as the Standard Template Library (STL)
- In 1994, STL was adopted as part of ANSI/ISO Standard C++

- Templates are not built into C, the nearest construct to template functions in C is Macros
- Vectors are an example of a template class that is not built into the standard C library
- Vectors can be replicated in C but you will need to write code to implement them (or find a library that already does this)

Standard template library

STL started with three basic components:

- **Containers**
  - Containers are used to manage collections of objects of a certain kind
  - Generic class templates for storing collection of data
- **Algorithms**
  - Algorithms act on containers.
  - They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.
  - Generic function templates for operating on containers
- **Iterators**
  - Generalized 'smart' pointers that facilitate use of containers
  - Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers
  - They provide an interface that is needed for STL algorithms to operate on STL containers

- **String abstraction** was added during standardization

Algorithms - Function Templates

- A ***function template*** behaves like any other function except that the template can have arguments of many different types

- A function template represents a group of functions
- The format for declaring a function template with type parameters is:

```
template <typename identifier> function_declaration;
```

- A type parameter may be a basic /fundamental / primitive / built-in type such as int or double or a user defined type such as class or structure
- For example, the C++ Standard Library contains the function template max(x, y) which returns the larger of x and y. That function template could be defined like this:

```
template <typename T>
inline T max(T a, T b) {
  return a > b ? a : b;
}
```

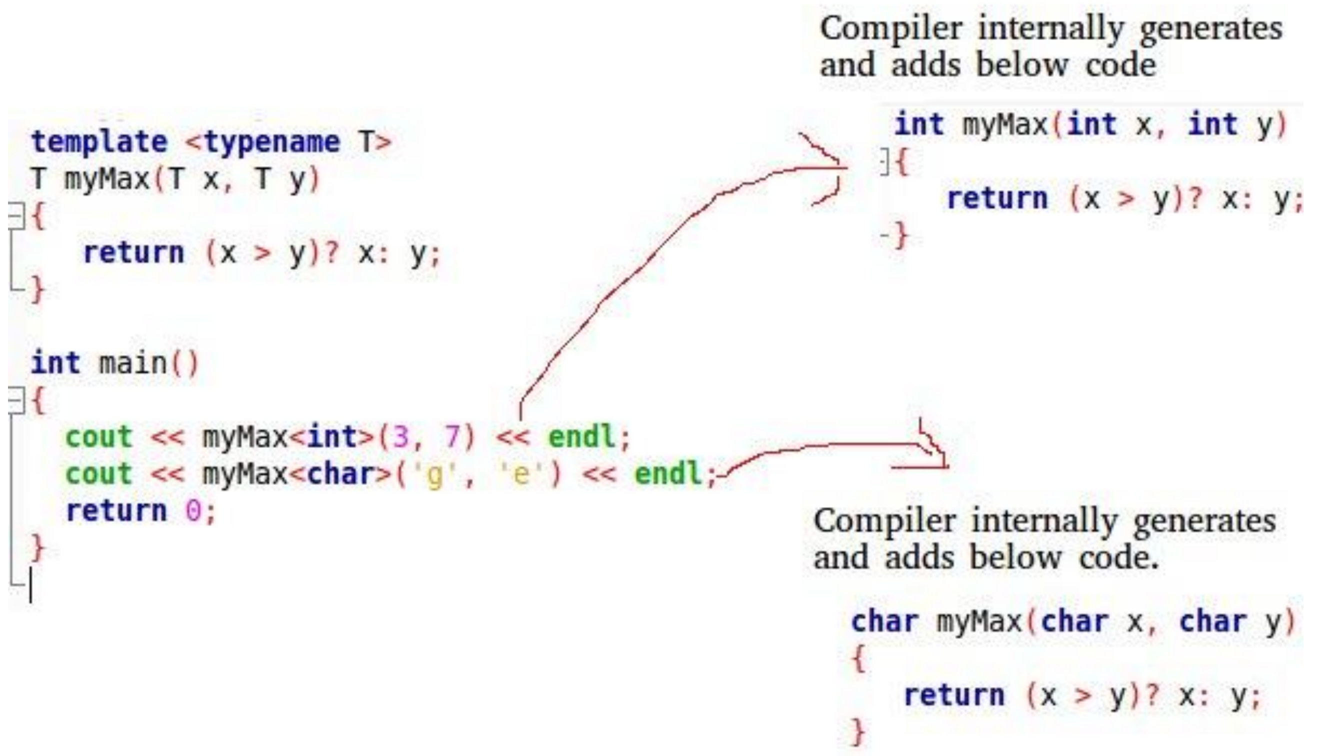
Function Templates defining a function template

```
template <typename T>
T myMax(T a, T b) {
   return a > b ? a : b;
}
```

```
// max<int> , max<char> and max<double> are
called by implicit argument type deduction
std::cout << myMax(3, 7) << std::endl;
ctd::cout <<myMax('A','a') << std::endl;
std::cout << myMax(3.0, 7.0) << std::endl;

// Which version is called depends on the compiler
as two different datatypes are passed as
arguments. Some compilers handle this by defining
a template function like double max <double> (
double a, double b);, while in some compilers you
need to explicitly cast it, like std::cout <<
max<double>(3,7.0);

std::cout << myMax(3, 7.0) << std::endl;
std::cout << myMax<double>(3, 7.0) << std::endl;
```



```
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}

int main()
{
   cout << myMax<int>(3, 7) << endl;
   cout << myMax<char>('g', 'e') << endl;
   return 0;
}
```

Compiler internally generates and adds below code

```
int myMax(int x, int y)
{
    return (x > y)? x: y;
}
```

Compiler internally generates and adds below code.

```
char myMax(char x, char y)
{
    return (x > y)? x: y;
}
```

Class templates

- A class template provides a specification for generating classes based on parameters
- Class templates are generally used to implement containers
- A class template is instantiated by passing a given set of types to it as template arguments
- Types can be both basic / fundamental /primitive / built-in datatypes and user defined datatypes (classes / structures)

Container classes

- There are seven standard “first-class” container classes, three container adaptor classes but only seven header files that provide access to these containers or container adaptors.
- Sequence Containers: implement data structures which can be accessed in a sequential manner.
  - **vector**
  - **list**
  - **deque**
  - arrays
  - forward_list(Introduced in C++11)
- Container Adaptors : provide a different interface for sequential containers.
  - queue
  - priority_queue
  - stack
- Associative Containers : implement sorted data structures that can be quickly searched (O(log n) complexity e.g. binary search).
  - **set**
  - **multiset**
  - **map**
  - **multimap**
- Unordered Associative Containers : implement unordered data structures that can be quickly searched
  - unordered_set (Introduced in C++11)
  - unordered_multiset (Introduced in C++11)
  - unordered_map (Introduced in C++11)
  - unordered_multimap (Introduced in C++11)

Other (non-STL) classes with STL-like characteristics

- There are some other classes that are regarded as near-containers, in that (for example) they can be used with iterators, which makes them accessible to manipulation via the STL algorithmsSequence Containers: implement data structures which can be accessed in a sequential manner.

  - **String**
    - Not to be confused with the legacy C-string data type, inherited from the C-language, and consisting of a simple character array with the characters of interest terminated with a null character. The C++ string class is indeed a bona fide class, with much more flexibility (and safety) than the older C-string, and should probably be used anywhere the C-string is not explicitly required.
  - **bitset**
  - **Valarray**

- Note that for many purposes even "ordinary" C-style arrays (which are not classes) and C-strings (which are also not classes) can be accessed from the STL with C-style pointers in the same way that STL containers are accessed with STL iterators

Class Templates - Vector

- Vector is a generic class template which can operate with generic types
- There is a single definition of the vector container, but it can be used to define many different kinds (datatypes) of vectors
- Vectors are similar to dynamic arrays with the ability to resize automatically when an element is inserted or deleted
- Their storage is handled automatically by the container
- Vectors also have safety features that make them easier to use than arrays, automated bounds checking and memory management
- Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators.
- Vectors consume more memory and have higher processing overheads than arrays in exchange for the ability to handle storage and grow dynamically

Vectors

- Vector is a generic class template which can operate with generic types

template < class T, class Alloc = allocator<T> > class vector;

- There is a single definition of the container **vector**, but it can be used to define many different kinds of vectors

vector <int> variablename
vector <string> variablename
vector <mytype> variablename

```
#include <iostream>
#include <string.h>
#include <vector>

sing namespace std;

class My_Student {
  private:
    string name;
  public:
     print();
};
int main(){
  vector <int> vi;
  vector <string> vs;
  vector <My_Student> vms;
return 0;
}
```

iterators

1. begin() – Returns an iterator pointing to the first element in the vector
2. end() – Returns an iterator pointing to the theoretical element that follows the last element in the vector
3. rbegin() – Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
4. rend() – Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)
5. cbegin() – Returns a constant iterator pointing to the first element in the vector.
6. cend() – Returns a constant iterator pointing to the theoretical element that follows the last element in the vector.
7. crbegin() – Returns a constant reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
8. crend() – Returns a constant reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

```
// C++ program to illustrate the  iterators in vector
#include <iostream>
#include <vector>
using namespace std;

int main()
{
	vector<int> g1;

	for (int i = 1; i <= 5; i++)
		g1.push_back(i);

	cout << "Output of begin and end: ";
	for (auto i = g1.begin(); i != g1.end(); ++i)
		cout << *i << " ";
```

```
	cout << "\nOutput of cbegin and cend: ";
	for (auto i = g1.cbegin(); i != g1.cend(); ++i)
		cout << *i << " ";

	cout << "\nOutput of rbegin and rend: ";
	for (auto ir = g1.rbegin(); ir != g1.rend();
	++ir)
		cout << *ir << " ";

	cout << "\nOutput of crbegin and crend : ";
	for (auto ir = g1.crbegin(); ir != g1.crend();
	++ir)
		cout << *ir << " ";

return 0;
}
```

```
Output of begin and end: 1 2 3 4 5
Output of cbegin and cend: 1 2 3 4 5
Output of rbegin and rend: 5 4 3 2 1
Output of crbegin and crend : 5 4 3 2 1
```

Capacity

1. size() – Returns the number of elements in the vector.
2. max_size() – Returns the maximum number of elements that the vector can hold.
3. capacity() – Returns the size of the storage space currently allocated to the vector expressed as number of elements.
4. resize() – Resizes the container so that it contains ‘g’ elements.
5. empty() – Returns whether the container is empty.
6. shrink_to_fit() – Reduces the capacity of the container to fit its size and destroys all elements beyond the capacity.
7. reserve() – Requests that the vector capacity be at least enough to contain n elements

```
// C++ program to illustrate the capacity function in vector
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector<int> g1;

   for (int i = 1; i <= 5; i++)
      g1.push_back(i);

   cout << "Size : " << g1.size();
   cout << "\nCapacity : " << g1.capacity();
   cout << "\nMax_Size : " << g1.max_size();

   // resizes the vector size to 4
   g1.resize(4);

   // prints the vector size after resize()
   cout << "\nSize : " << g1.size();
```

```
// checks if the vector is empty or not
   if (g1.empty() == false)
      cout << "\nVector is not empty";
   else
      cout << "\nVector is empty";

   // Shrinks the vector
   g1.shrink_to_fit();
   cout << "\nVector elements are: ";
   for (auto it = g1.begin(); it != g1.end();
it++)
      cout << *it << " ";

   return 0;
}
```

Output:

```
Size : 5
Capacity : 8
Max_Size : 4611686018427387903
Size : 4
Vector is not empty
Vector elements are: 1 2 3 4
```

Element Access

1. reference operator [g] – Returns a reference to the element at position ‘g’ in the vector
2. at(g) – Returns a reference to the element at position ‘g’ in the vector
3. front() – Returns a reference to the first element in the vector
4. back() – Returns a reference to the last element in the vector
5. data() – Returns a direct pointer to the memory array used internally by the vector to store its owned elements.

```
// C++ program to illustrate the  element accesser in vector
#include <bits/stdc++.h>
using namespace std;
int main()
{
   vector<int> g1;

   for (int i = 1; i <= 10; i++)
      g1.push_back(i * 10);

   cout << "\nReference operator [g] : g1[2] = " << g1[2];

   cout << "\nat : g1.at(4) = " << g1.at(4);

   cout << "\nfront() : g1.front() = " << g1.front();

   cout << "\nback() : g1.back() = " << g1.back();

   // pointer to the first element
   int* pos = g1.data();

   cout << "\nThe first element is " << *pos;
   return 0;
}
```

```
Output:
Reference operator [g] : g1[2] = 30
at : g1.at(4) = 50
front() : g1.front() = 10
back() : g1.back() = 100
The first element is 10
```

Modifiers

1. assign() – It assigns new value to the vector elements by replacing old ones
2. push_back() – It push the elements into a vector from the back
3. pop_back() – It is used to pop or remove elements from a vector from the back
4. insert() – It inserts new elements before the element at the specified position
5. erase() – It is used to remove elements from a container from the specified position or range.
6. swap() – It is used to swap the contents of one vector with another vector of same type. Sizes may differ.
7. clear() – It is used to remove all the elements of the vector container
8. emplace() – It extends the container by inserting new element at position
9. emplace_back() – It is used to insert a new element into the vector container, the new element is added to the end of the vector

```
// C++ program to illustrate the Modifiers in vector
#include <bits/stdc++.h>
#include <vector>
using namespace std;
int main()
{
      vector<int> v;   // Assign vector

       v.assign(5, 10);   // fill the array with 10 five times

       cout << "The vector elements are: ";
       for (int i = 0; i < v.size(); i++)
         cout << v[i] << " ";

       v.push_back(15);  // inserts 15 to the last position
       int n = v.size();
       cout << "\nThe last element is: " << v[n - 1];

       v.pop_back();   // removes last element
       cout << "\nThe first element is: " << v[0];

       v.emplace(v.begin(), 5);  // inserts at the beginning
       cout << "\nThe first element is: " << v[0];
```

```
// Inserts 20 at the end
  v.emplace_back(20);
  n = v.size();
  cout << "\nThe last element is: " << v[n - 1];

  v.clear();    // erases the vector
  cout << "\nVector size after erase(): " << v.size();
```

```
The vector elements are: 10 10 10 10 10
The last element is: 15
The first element is: 10
The first element is: 5
The last element is: 20
Vector size after erase(): 0
```

```
// C++ program to illustrate the Modifiers in vector - continued
// two vector to perform swap
   vector<int> v1, v2;
   v1.push_back(1);
   v1.push_back(2);
   v2.push_back(3);
   v2.push_back(4);

   cout << "\n\nVector 1: ";
   for (int i = 0; i < v1.size(); i++)
      cout << v1[i] << " ";

   cout << "\nVector 2: ";
   for (int i = 0; i < v2.size(); i++)
      cout << v2[i] << " ";

   // Swaps v1 and v2
   v1.swap(v2);

   cout << "\nAfter Swap \nVector 1: ";
   for (int i = 0; i < v1.size(); i++)
      cout << v1[i] << " ";
```

```
   cout << "\nVector 2: ";
   for (int i = 0; i < v2.size(); i++)
      cout << v2[i] << " ";
   return 0;
}
```

```
Output from Vector swap
Vector 1: 1 2
Vector 2: 3 4
After Swap
Vector 1: 3 4
Vector 2: 1 2
```