Outline

1. Chapter 12: C++ Templates
   - Template Functions
   - Template Classes
     - Introduction
     - Vector class
     - User-Defined Template Classes

Templates Allow Code For Different Types

Python doesn't associate types with variable names, so the same code might work for different types.
The function Maximum finds the larger of two numbers having the same type (as long as the operator > is defined for that type). For example, the types int, float, and even Rational will work here:

```
def Maximum(a, b):
   if a > b:
      return a
   else:
      return b
```

Dynamic typing is possible in Python because the interpretor waits until it is ready to execute a Python statement before converting it to machine language.

Templates Allow Code For Different Types

Python doesn't associate types with variable names, so the same code might work for different types.
The function Maximum finds the larger of two numbers having the same type (as long as the operator > is defined for that type). For example, the types int, float, and even Rational will work here:

```
def Maximum(a, b):
   if a > b:
      return a
   else:
      return b
```

Dynamic typing is possible in Python because the interpretor waits until it is ready to execute a Python statement before converting it to machine language.

C++: Different Versions For Different Types

In C++ we have learned that C++ variables must be defined with a fixed type, so that the compiler can generate the specific machine instructions needed to manipulate the variables.

```
int maximum_int(int a, int b)
{
  if (a > b){
    return a;
  }
  else {
    return b;
  }
}
```

C++: Different Versions For Different Types

```
double maximum_double(double a, double b)
{
  if (a > b){
    return a;
  }
  else {
    return b;
  }
}
```

There is a *template* mechanism in C++ that allows to write functions and classes with similar to Python's functionalities.

Template Function Example: C++

We used **typedef** statement in the previous chapter, however it doesn't allow the same code to be used for multiple types since the generated machine language code must be specific for the type.

```
template <typename Item> // or template <class Item>

Item maximum(Item a, Item b) {
  if (a > b) {
    return a;
  }
  else {
    return b;
  }
}
```

Comment: you may use any legal identifier instead of `Item`, but commonly `Item` or `Type` are used.

Template Function Example: C++

C++ templates allow us to write one version of the code, and the compiler automatically generates different versions of the code to each data type as needed.

```
 int main()
{
   int a=3, b=4;
   double x=5.5, y=2.0;

   cout << maximum(a, b) << endl;
   cout << maximum(x, y) << endl;
   return 0;
}
```

Template Function Example: C++

- The C++ compiler doesn't generate any code if no template function is called
- Depending on compiler, it may or may not catch syntax errors in template functions that are not called, hence
- It is important to test all the template functions
- The term *instantiate* is used to indicate that the compiler generates the code for a specific type.
  In our previous example, the compiler instantiates an **int** and **double** versions of the **maximum** function.

C++ Template Classes: Container Classes

We can also write classes using templates.

Recall container classes which provide certain access to each item (Stack, Queue, ...) — all behave the same for different data types of the items contained.

Iterators should be provided to allow abstract traversal (without needing to know how the container is implemented).

C++ template classes are able to provide this.

Similarly to the function's template,

- As a container class is used for some datatype, the compiled template class for that type is instantiated.
- No code for a template class instance is compiled until it is needed.

C++ Template Classes: Container Classes

We can also write classes using templates.

Recall container classes which provide certain access to each item (Stack, Queue, ...) — all behave the same for different data types of the items contained.

Iterators should be provided to allow abstract traversal (without needing to know how the container is implemented).

C++ template classes are able to provide this.

Similarly to the function's template,

- As a container class is used for some datatype, the compiled template class for that type is instantiated.
- No code for a template class instance is compiled until it is needed.

C++ Template Classes: Container Classes

We can also write classes using templates.

Recall container classes which provide certain access to each item (Stack, Queue, ...) — all behave the same for different data types of the items contained.

Iterators should be provided to allow abstract traversal (without needing to know how the container is implemented).

C++ template classes are able to provide this.

Similarly to the function's template,

- As a container class is used for some datatype, the compiled template class for that type is instantiated.
- No code for a template class instance is compiled until it is needed.

C++ Template Classes: Container Classes

We can also write classes using templates.

Recall container classes which provide certain access to each item (Stack, Queue, ...) — all behave the same for different data types of the items contained.

Iterators should be provided to allow abstract traversal (without needing to know how the container is implemented).

C++ template classes are able to provide this.

Similarly to the function's template,

- As a container class is used for some datatype, the compiled template class for that type is instantiated.
- No code for a template class instance is compiled until it is needed.

C++ Template Classes: Container Classes

We can also write classes using templates.

Recall container classes which provide certain access to each item (Stack, Queue, ...) — all behave the same for different data types of the items contained.

Iterators should be provided to allow abstract traversal (without needing to know how the container is implemented).

C++ template classes are able to provide this.

Similarly to the function's template,

- As a container class is used for some datatype, the compiled template class for that type is instantiated.
- No code for a template class instance is compiled until it is needed.

The Standard Template Library

- The *Standard Template Library* (*STL*) implements most of the common container classes as C++ template classes.
- It is now a standard part of the C++ library.
- It defines a wide variety of containers for classes which implement a few basic operations. (For example, < for binary search trees or priority queues.)
- It provides iterators for these classes.

The vector Template Class: Example 1

One of the simpler *STL* classes is the **Vector** class. It provides functionality similar to the dynamic array classes we developed.

```
#include<vector>

   ...
int main()
{
   vector<int> iv;
   vector<double> dv;
   int i;
   for (i=0; i<10; ++i) {
      iv.push_back(i);
      dv.push_back(i + 0.5); }
   for (i=0; i< 10; ++i) {
      cout << iv[i] << " " << dv[i] << endl; }
   return 0;
}
```

The vector Template Class; example 2

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
   //create a vector with 5 int elems, each set to 3
   vector<int> iv(5, 3);
   //create a vector with 5 double elems, set to 0.0
   vector<double> dv(5);
   int i;

   for (i=0; i<5; ++i) {
      cout << iv[i] << " " << dv[i] << endl; }
}
```

The vector Template Class: example 3

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
   vector<int> iv;
   vector<int>::iterator iter;
   int i;

   for (i=0; i<10; ++i) {
      iv.push_back(i);
   }
   for (iter=iv.begin(); iter != iv.end(); ++iter) {
      cout << *iter << endl;
   }
   return 0;
}
```

Vector class - conclusion

Vector class is implemented as a dynamic array, so its use and efficiency are similar to the C++ dynamic array class we developed and the built-in Python list.

You can visit http://www.cplusplus.com/reference/vector/vector/ for the list of Vector class methods, as well as pages 433–444 in our book.

The *STL* - Conclusion

The *Standard Template Library* provides template class implementations of a queue, list, set, and hash tables along with algorithms and iterators to use with a number of classes.

Check out the algorithms library `http://www.cplusplus.com/reference/algorithm/`. Find sort, min_element and other functions there and see how to use them.

User-Defined Template Classes

- The header file, `<classname>.h`, is the same as for ordinary classes, but class definition has a template data type, a “wild card” typename instead of a normal type like `int` or `double`.
- The class definition is preceded by `template <typename T>` where `T` can be any identifier not in use.
  (for example, `Item` in the `Stack` class.)
- Whenever the template data type is needed in a function declaration, it is used like an ordinary type name:
  ```
  bool pop(Item &item);
  ```
- The last line of the header file includes the implementation file:
  ```
  #include "<classname>.template"
  ```
  (which does **not** include the header file).

User-Defined Template Classes

```
 //Stack.h
...
#include<cstdlib> //for NULL
template <typename Item>
class Stack {
public:
   Stack();
   ~Stack();

   int size() const { return size_; }
   bool top(Item &item) const;

   bool push(const Item &item);
   bool pop(Item &item);
```

User-Defined Template Classes

```
 private:
// prevent these methods from being called
   Stack(const Stack &s);
   void operator=(const Stack &s);

   void resize();

   Item *s_;
   int size_;
   int capacity_;
};
#include "Stack.template"
```

User-Defined Template Classes

```
// Stack.template
template <typename Item>
Stack<Item>::Stack() constructor
{
  s_ = NULL;      size_ = 0;      capacity_ = 0;
}

template <typename Item>
Stack<Item>::~Stack() destructor
{
  delete [] s_;
}
```

The rest see in Stack.template

Comment: we could put all the defs at the end of the header file Stack.h

User-Defined Template Classes

```
 // test_Stack.cpp
#include "Stack.h"
int main()
{
  Stack<int> int_stack;
  Stack<double> double_stack;
  int_stack.push(3);
  double_stack.push(4.5);
  return 0;
}
```

The rest see in test_Stack.cpp

In-Class Work

1. Implement a template minimum function and test it on int and double type values.
2. Implement a Queue using templates along with the code to test it.