Function Templates

Consider the following function:

```
void swap (int& a, int& b) {
   int tmp = a;
   a = b;
   b = tmp;
}
```

Swapping integers. This function let's you swap the contents of two integer variables. But when programming quite a big application, it is probable that you have to swap float, long or char variables, or even `shape` variables . So, an obvious thing to do would be to copy the piece of code (cut-n-paste!) and to replace all `int`s by `shape`s, wouldn't it? A drawback of this solution is the number of similar code pieces, that have to be administered. Additionally, when you need a new swap function, you must not forget to code it, otherwise you get a compile-time error. And now imagine the overhead when you decide to change the return type from `void` to `int` to get information, if the swap was successful - the memory could be too low to create the local `tmp` variable, or the assignment operator (see `shape`) could not be defined. You would have to change all x versions of `swap` - and go insane...

Templates or Parametrized types. The solution to this dark-drawn scenario are templates, template functions are functions that are parametrized by at least one type of their arguments:

```
template <class T>
void swap (T& a, T& b) {
   T tmp = a;
   a = b;
   b = tmp;
}
```

The arguments are references to the objects, so the objects are not copied to the stack when the function is called. When you write code like:

```
int a = 3, b = 5;
shape MyShape, YourShape;
swap (a, b);
swap (MyShape, YourShape);
```

the compiler "instantiates" the needed versions of `swap`, that means, the appropriate code is generated. There are different template instantiation techniques, for example manual instantiation, where the programmer himself tells the compiler, for wich types the template should be instantiated.

Class Templates

Class templates provide a way to parameterize the types within a class. This means that rather than explicitly specifying the data type associated with a class member or method, a placeholder is used. When we instantiate an instance of the class, the actual data type is substituted for the parameter, or placeholder. The process of forming a class with specific data types from a class template is called template instantiation.

In addition to parameterize types, it is possible to use non-type parameters in a class template. These non-type parameters will serve as constants within a particular instance of the class.

Although this may seem complex right now, the basic use of class templates is easy once you see a few examples. In fact, you may be using class templates already in your programs. Many of the container classes are implemented as class templates.

Typically, when building class templates, it is easier to first write and debug a concrete class.

```
class vector {

 public:
   vector (int s) { v = new double [sz = s]; }
  ~vector () { delete[] v; }
   double& operator[] (int i) { return v[i]; }
   int get_size() { return sz; }

 private:
   double* v;
   int sz;
};
```

This class is then turned into a template:

```
template <class T>
class vector {

 public:
   vector (int s) { v = new T [sz = s]; }
  ~vector () { delete[] v; }
   T& operator[] (int i) { return v[i]; }
   int get_size() { return sz; }

 private:
   T* v;
   int sz;
};
```

Notice the syntax used for the class template. The keyword template is used, followed by a list of parameters each preceded by the keyword class (or equivalently by the keyword typename). The parameter list is enclosed using “<” and “>” symbols. Everywhere the

keyword int was used to specify a data type in the vector class, the parameter T is substituted. This is written in bald style.

When the class is instantiated, the data type int is specified within “<” and “>” symbols following the class name. "int" will be substituted for "T" as the class is built by the compiler. Creating a vector of another data type is simple. Here are some examples:

```
vector<int>      int_vector (10);
vector<char>     char_vector (10);
vector<shape>    shape_vector (10);
```

Note that the parameter can be replaced by built-in data types, library classes (such as string) or user defined classes, provided the class itself can support the data type.

The Standard Template Library

The Standard Template Library, or **STL**, is a collection of container classes, generic algorithms and related components that can greatly simplify many programming tasks in C++.

The container classes are classes used to hold objects of any particular type. Among the methods of the container classes are methods to add and remove elements, return elements and return iterators.

Iterators are an abstraction of pointers. They provide a means to traverse containers and access objects within containers.

The generic algorithms are a collection of routines that can be used to perform common operations on container classes, such as sorting, merging and finding. They are called generic because they are independent of the container class used and of the data type of the objects held in those container classes. The iterators are used to link a container with an algorithm. A container class has methods to return iterators that provide access to its contents. These iterators are provided, as arguments, to an algorithm. The algorithm can then traverse, access and manipulate the contents of the container.

Containers

The STL provides container classes that can be used to hold objects, similar to the way an array can be used to hold objects. They can be used to store objects of intrinsic types, such as int or double, class objects defined by C++, such as the string, or user defined objects, such as instances of any user defined class.

Few benefits that the container classes offer over arrays:

- Container classes handle all sizing and memory allocation issues. You can't write past the end of a container class like you can with an array
- Container classes are optimized for their intended use. They contain utility methods such as empty(), size() and clear() that can simplify programming.
- They contain other methods that all insertion, deletion and copying.
- They have equality, inequality and assignment operators. You can directly compare two containers or assign on to another.
- They provide iterators. Iterators are an abstraction of pointers. They allow similar syntax and use, but also have some built-in methods and checks that increase their utility.

The STL provides both sequence and associative containers.

The **sequence containers** are meant to hold a collection of objects of one type in a linear sequence. This is similar to an array. The STL provides three sequence containers: vector, list and deque.

Example of use of vector:

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

using std::cout;
using std::endl;

int main( )
{
   // Creates an empty vector
   std::vector <int> v1;

   // size_type is defined in the vector include file as the type return by the
   // size method. Think of it as a really fancy way to declare an int.
   std::vector <int>::size_type i;

   // Add elements to the end of v1
   v1.push_back(5);
   v1.push_back(12);
   v1.push_back(1);
   v1.push_back(392);

   // See what we got
   for (i = 0; i < v1.size(); i++) {
      cout << v1[i] << " ";
   }
   cout << endl;

   // Sort
   sort(v1.begin(),v1.end());

   // See what we got
   for (i = 0; i < v1.size(); i++) {
      cout << v1[i] << " ";
   }
   cout << endl;

   // Reverse
   reverse(v1.begin(),v1.end());
```

```
    // See what we got
     for (i = 0; i < v1.size(); i++) {
        cout << v1[i] << " ";
    }
    cout << endl;
    return 0;
}
```

- Notice how easy it is to create and load a vector. The default constructor creates an empty vector. Push_back() adds elements to the end of the vector.
- The vector was sorted using one simple call to the generic algorithm sort.
- The vector was reversed using one simple call to the generic algorithm reverse.
- Sort() and reverse() take iterators as arguments. The vector class method begin() returns an iterator to the start of the vector; end() returns an iterator one past the end of the vector.

The second type of container in the STL is an **associative container**. These containers support the construction of hashes and sets. A set is a collection of keys. It can be used to determine whether a particular object is present or not. A hash is a collection of key, value pairs. Each value may be inserted by key and located by its key.

- set <key_type> - The set container holds unique keys of any type. It provides for fast retrieval of keys and tests for presence of particular keys.
- multiset <key_type> - The multiset contain also holds keys of any type, but they need not be unique. Multiple copies of each key may be stored. Multiset also provides for fast key retrieval.
- map <key_type, value_type> - A map is a hash. It stores objects of one type, the values, in a sorted order based on objects of another type, the keys. Maps are also optimized for fast retrieval of values by key.
- multimap <key_type, value_type> - Allows storage of values based on keys. The keys may be duplicated.

Example of use of map:

```
#include <map> //The map include file is used for both map and multimap
#include <string>
#include <iostream>
using std::cout;
using std::endl;
using std::string;

int main() {
    // Creates an empty multimap
```

```
std::map<string,string> addressbook;

// Maps and multimaps deal with pairs
std::pair<string,string> p1("John","123 Seseme St.");

// Insert values into the phonebook
addressbook.insert(p1);

std::pair<string,string> p2("Cookie Monster","125 Seseme St.");
addressbook.insert(p2);

addressbook.insert(std::pair<string,string>("Ernie","99 Seseme St."));
addressbook.insert(std::pair<string,string>("Bert","99 Seseme St."));

// The iterator points to a pair, rather than a single value
std::map<string,string>::iterator iter;

// Look up my address
// Note that a direct query of a map causes insertion of  record if the key does not already exist.
cout << "Finding John's address" << endl;
cout << "John lives at " << addressbook["John"] << endl;
cout << endl;



// The key Elmo does not exist - the address will be the default string (empty string)
cout << "Finding Elmo's address via direct query" << endl;
cout << "There are " << addressbook.count("Elmo") << " entries for Elmo" <<  endl;

cout << "Elmo lives at " << addressbook["Elmo"] << endl;
cout << "There are " << addressbook.count("Elmo") << " entries for Elmo" << endl;
cout << endl;

// Here's another way to search for entries that will not create new
// records if a non-existent key is used
cout << "Finding John's address using the find method and an iterator" << endl;
iter = addressbook.find("John");
if (iter != addressbook.end()) {
    cout << iter->first << " lives at " << iter->second << endl;
    // another way
    cout << (*iter).first << " lives at " << (*iter).second << endl;
}
cout << endl;
```

```
    cout << "Finding Oscar's address using the find method and an iterator" << endl;
    iter = addressbook.find("Oscar");
    if (iter != addressbook.end()) {
        cout << iter->first << " lives at " << iter->second << endl;
    }
    else {
        cout << "Oscar not found" << endl;
    }
    cout << "There are " << addressbook.count("Oscar") << " entries for Oscar" << endl;

    return 0;
}
```

Another example with vector

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

int main() {

    // Create a vector, initialized by an array
    int myArray[5] = {4,3,7,8,9};
    std::vector<int> vec1(myArray, myArray + 5);

    // Iterator
    std::vector<int>::iterator iter;

    // This is how to loop through the vector
    // begin() returns an iterator to the start of vec1
    // end() returns one past the end of vec1.
    for (iter = vec1.begin(); iter != vec1.end(); iter++) {
        cout << *iter << " ";
    }
    cout << endl;

    cout << "The vector has " << vec1.size() << " elements" << endl;
    cout << "The second element is " << vec1[1] << endl;
    cout << "The first element is " << vec1.front() << endl;
    cout << "The last element is " << vec1.back() << endl;

    cout << endl;
    vec1.pop_back();    // Removes last element
    cout << "The last element is now " << vec1.back() << endl;
    vec1.pop_back();    // Removes last element

    cout << "The last element is now " << vec1.back() << endl;
    vec1.push_back(2);    // Puts 2 at the end of vec1
    cout << "The last element is now " << vec1.back() << endl;
    cout << endl;

    // Create a second vector
    std::vector<int> vec2;
```

```
    // Insert values from vec1 to vec2
    vec2.insert(vec2.begin(),vec1.begin(),vec1.end());

    for (iter = vec2.begin(); iter != vec2.end(); iter++) {
        cout << *iter << " ";
    }
    cout << endl;

    // Insert more values (Three fives) at the end of vec2
    vec2.insert(vec2.end(),3,5);

    for (iter = vec2.begin(); iter != vec2.end(); iter++) {
        cout << *iter << " ";
    }
    cout << endl;

    return 0;
}
```

One more example:

```
#include <vector>    // Needed to use the vector class
#include <algorithm>    //Needed to use the generic algorithms
#include <iostream>    // You know this one

int main() {

    // Create a vector, initialized by an array
    int myArray[10] = {99,4,3,7,8,67,9,33,12,67};
    std::vector<int> vec1(myArray, myArray + 10);

    // Iterator
    std::vector<int>::iterator iter;

    cout << "Original sequence: ";
    for (iter = vec1.begin(); iter != vec1.end(); iter++) {
        cout << *iter << " ";
    }
    cout << endl;
```

```
    // REVERSE
    reverse(vec1.begin(), vec1.end());

    cout << "Reversed: ";
    for (iter = vec1.begin(); iter != vec1.end(); iter++) {
        cout << *iter << " ";
    }
    cout << endl;



    // SORT
    cout << "Sorted: ";
    sort(vec1.begin(), vec1.end());

    for (iter = vec1.begin(); iter != vec1.end(); iter++) {
        cout << *iter << " ";
    }
    cout << endl;

    // REMOVE ANY DUPLICATE ELEMENTS
    std::vector<int>::iterator newEnd;
    newEnd = unique(vec1.begin(), vec1.end());

    cout << "Duplicates removed: ";
    for (iter = vec1.begin(); iter != newEnd; iter++) {
        cout << *iter << " ";
    }
    cout << endl;

    return 0;
}
```

File Input and Output

Learning how to read and write files is an important step in learning any programming language. Any real world application is likely to process large amounts of information. A simple technique to transfer and record data is via files. More advanced techniques for data

storage and manipulation can involve the use of relational databases or special data formats such as XML. For now, we will study the use of text files.
One important issue with writing results to output files is data format. Whether the ultimate consumer of a programs output be man or machine, the format of this output can be as significant as its content. If the output is being consumed by another program, then the output format is likely to be predetermined and highly specific in order for the consuming program to be able to properly read and access the data. If the output file is to be read by people, data format can significantly influence understanding and utility. Iostream manipulators provide a way to control output format.

File Input and Output

The techniques for file input and output, i/o, in C++ are virtually identical to those introduced in earlier lessons for writing and reading to the standard output devices, the screen and keyboard. To perform file input and output the include file **fstream** must be used: #include <fstream>
Fstream contains class definitions for classes used in file i/o. Within a program needing file i/o, for each **output file** required, **an object of class ofstream** is instantiated. For each **input file** required, **an object of class ifstream** is instantiated. The ofstream object is used exactly as the cout object for standard output is used. The ifstream object is used exactly as the cin object for standard input is used. This is best understood by studying an example:

```
#include <iostream>
using std::cout;
using std::endl;

#include <fstream>
using std::ofstream;

int main()
{
   ofstream myFile("out.txt"); // Creates an ofstream object named myFile

   if (! myFile) // Always test file open
   {
      cout << "Error opening output file" << endl;
      return -1;
   }

   myFile << "Hello World" << endl;
   myFile.close();

   return 0;
}
```

Let's examine this program. The first step created an ofstream object named myFile.

The constructor for the ofstream class takes two arguments. The first specifies a file name as a C-style string, the second a file mode. There are two common file open modes, truncate and append. By default, if no mode is specified and the file exists, it is truncated. The mode is specified by using an enumerator defined in the ios class. The ios class contains members which describe modes and states for the input and output classes.

To open a file, truncating any existing contents, use any of the three equivalent statements:

```
ofstream myFile("SomeFileName");
ofstream myFile("SomeFileName",ios::out);
ofstream myFile("SomeFileName",ios::out | ios::trunc);
```

To open a file, and append to any previous contents:

```
ofstream myFile("SomeFileName",ios::app);
```

To open a file for binary output:

```
ofstream myFile("SomeFileName",ios::binary);
```

To open a file for input, an object of ifstream class is used:

```
 ifstream inFile("SomeFileName");
```

A program to copy the content of a file into another one:

```
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include <fstream>
using std::ofstream;
using std::ifstream;

#include <string>
using std::string;

int main()
{
   char ch;
   string iFileName;
   string oFileName;

   cout << "Enter the source file name: ";
   cin >> iFileName;

   cout << "Enter the destination file name: ";
   cin >> oFileName;

   ofstream oFile(oFileName.c_str());
   if (! oFile) // Always test file open
   {
      cout << "Error opening output file" << endl;
```

```
        return -1;
    }

    ifstream iFile(iFileName.c_str());
    if (! iFile)
    {
        cout << "Error opening input file" << endl;
        return -1;
    }

    while (iFile.get(ch))
    {
        oFile.put(ch);
    }

  return 0;
}
```