

# Introduction to Programming using C++

## Lecture Nine: Templates & the STL

Carl Gwilliam

[gwilliam@hep.ph.liv.ac.uk](mailto:gwilliam@hep.ph.liv.ac.uk)

<http://hep.ph.liv.ac.uk/~gwilliam/cppcourse>

# Templates

- Templates are a very powerful feature of C++.
- They allow you to use one function or class with many different data types, without having to recode specific versions for each data type (avoiding code duplication).
- With templates the user creates generic functions and classes where the type of data the function or class operates upon is specified as a parameter ...

# Function Templates

- A generic function is created via the keyword **template**:

```
template <class T> return-type function-name(args){...}
```

- This is followed by a **template parameter** (within < >), which is a special kind of parameter that can be used to pass a **type (T)** as argument.
- The function templates can use these parameters just like any other regular type (even declaring new objects of that type).
- **T** is a placeholder for the data type used by the function. The compiler automatically replaces this by an actual type when it creates a specific version of the function

# Function Templates

- When we call the function we specify the **actual** type with angle brackets (<...>).

function-name<type>(args)

- The compiler automatically generates a function with T replaced by requested type when function is called.
- Note: the type T must have the relational operators < and > defined for calling this function to make sense

```
[FunctionTemplate.cpp]
template<class T>
T GetMax(T& a, T& b) {
    T res;
    if (a > b) res = a;
    else if (a < b) res = b;
    // return object of type T
    return res;
}

int main() {
    int i=5, j=6;
    double k = 2.3, l = 5.7;
    cout << GetMax<int>(i, j)
          << endl;
    cout << GetMax<double>(k, l)
          << endl;
    return 0;
}
```

6

5.7

# Multiple Template Parameters

- You can define more than one generic type using a comma separated list
- If the generic types are used in the function args, the type can automatically be worked out by the compiler and so we don't need to specify it in angle brackets:

```
cout << GetMin(i, j) << endl;
```

```
[MultiTemplate.cpp]
:
template<class T, class U>
T GetMin(T& a, U& b) {
    T res;
    if (a < b) res = (T)a; //cast
    else if (a > b) res = (T)b;
    // return object of same type
    // as first arg
    return res;
}

int main() {
    int i=5;
    double j = 2.3;
    cout << GetMin(i, j) << endl;
    return 0;
}
```

# Example: Generic Bubble Sort

- Sorting is the sort of operation for which function templates were defined

```
template<class T>
void bubble(T* sample, int size) {
    T swap;
    for (int i = 1; i < size; i++) {
        for (int j=size-1;j>=1; j--) {
            if (sample[j-1] > sample[j]) {
                //exchange elements
                swap = sample[j-1];
                sample[j-1] = sample[j];
                sample[j] = swap;
            }
        }
    }
}
```

**[BubbleSort.cpp]**

**[BubbleSort.cpp]**

```
:
int main() {
    int iarray[] = {7,5,9,3,6};
    double darray[] = {4.3,2.5,10.2, 3.0, 1.1};
    bubble(iarray, 5); bubble(darray, 5);
    for (int i = 0; i < 5; i++)
        cout << iarray[i] << ", ";
    cout << endl;
    for (int i = 0; i < 5; i++)
        cout << darray[i] << ", ";
    cout << endl;
    return 0;
}
```

3, 5, 6, 7, 9

1.1, 2.5, 3, 4.3, 10.2

# Class Templates

- We can also write generic **classes**, which can be used for storage and manipulation of any type:

```
template <class T> class class-name {...};
```

- When we create a specific instance of that class we must again pass the type in angle brackets (<...>):

```
class-name <type> object;
```

- A member function declared outside class declaration must always be preceded with the **template<...>** prefix...

# Class Templates

```
// templated class
:
template <class T>
class mypair {
    T a, b;
public:
    mypair (T first, T second)
        {a=first; b=second;}
    T getmax ();
};
// member function: must be
// preceded by template<...>
template <class T>
T mypair<T>::getmax () {
    T res;
    if (a > b) res = a;
    else if (a < b) res = b;
    return res;
}
[MyPair.h]
```

```

:
int main () {
    //specify actual type in angle brackets
    mypair <int> myints (100, 75);
    mypair <float> myfloats(6.3,32.4);
    cout << myints.getmax() << "\t"
         << myfloats.getmax() << endl;
    return 0;
}
[ClassTemplate.cpp]
```

|     |      |
|-----|------|
| 100 | 32.4 |
|-----|------|

- Confused by so many T's. There are 3 T's in this declaration
  - first is template parameter
  - second is return type of function
  - third one (in brackets) specifies that the function's template parameter is also the class template parameter



# Template Specialisation

- We can define a different implementation for a template when a specific type is passed as template parameter => This is called a **specialisation** of that template.
- Here we precede the declaration with an **empty** template parameter (template<>) and place the type for which we are specialising in angle brackets after the class name:

```
template <> class class-name<specific-type> {...};  
//contrast to template <class T> class class-name {...};
```

- When we declare a specialisation of a class we must define all it's members, even if same as generic template class (since there is no "inheritance" between them).
- Template **functions** can similarly be specialised

# Template Specialisation

- A container class that has an increment func, except for char types where it has uppercase func instead:

```
int main () {  
    mycontainer<int> myint (7);  
    mycontainer<char> mychar ('j');  
    cout << myint.increase()  
        << endl;  
    cout << mychar.uppercase()  
        << endl;  
    return 0;  
}
```

**[MyContainer.h]**

```
// class template:  
template <class T> class mycontainer {  
    T element;  
public:  
    mycontainer (T arg) {element=arg;}  
    T increase () {return ++element;}  
};  
  
// class template specialization:  
template <> class mycontainer <char> {  
    char element;  
public:  
    mycontainer (char arg) {element=arg;}  
    char uppercase () {  
        if ((element>='a')&&(element<='z'))  
            element+='A'-'a';  
        return element;  
    }  
};
```

**[Specialisation.cpp]**

# Non-type and Default Parameters

- Besides template args which represent types, templates can also have regular typed parameters, similar to those found in functions. E.g.

```
template <class T, int size> class myclass {...};
```

- You can also give default values to template parameters (including non-typed ones) in exactly the same way as to arguments. E.g.

```
template <class T=int> class myclass {...};
```

- This specifies what type will be used in the case that no type is explicitly given in the class initialisation.

# Example: Generic Array

- Generic type array class with bounds checking:

```
template<class T, int size> class MyArray {
    T a[size];
public:
    MyArray() {
        for (int i = 0; i < size; i++) a[i] = 0;
    }
    T& operator[] (int i);
};
// range checking
template<class T, int size>
T& MyArray<T, size>::operator[](int i) {
    if (i < 0 || i > size-1) {
        cout << "Index " << i
            << " out of bounds" << endl;
        exit(1);
    } else return a[i];
}
```

**[MyArray.h]**

```
:
int main() {
    MyArray<double, 5> dbobj;
    for (int i = 0; i < 5; i++) {
        dbobj[i] = (double)i/2;
    }
    for (int i = 0; i < 5; i++) {
        cout << dbobj[i] << ", ";
    }
    cout << endl;
    return 0;
}
```

**[GenericArray.cpp]**

0, 0.5, 1, 1.5, 2

# Templates and Multiple Files

- From the point of view of the compiler, templates are not normal functions or classes.
- They are compiled on demand => template function is not compiled until an instantiation with specific args is required. Compiler then generates a function specifically for those args from the template.
- Because templates are compiled when required, the implementation (definition) of a template class or function must be in the **same** file as its declaration => can't separate interface in separate header file.

# Standard Template Library (STL)

- The Standard Template Library (STL) provides general-purpose templated classes/functions that implement many commonly used data structures and algorithms.
- As the STL is constructed from templates, the data structures and algorithms can be applied to nearly any type of data.
- Here we only have time for a brief overview of two of the more commonly used & useful components of the STL.
- For more detailed information see:

<http://cppreference.com/cppstl.html>

# Components of the STL

- Sequence Containers, e.g.
  - **Vector**  
Array of elements allowing fast random access and appending of elements.
  - **Lists**  
Faster insertion and deleting but slower random access
  - **Double-Ended Queue**  
Fast appending and prepending
- Associative Containers, e.g.
  - **Maps**  
Values are mapped to unique keys
- Algorithms to manipulate (all) containers, e.g.
  - Find, Sort, Copy ...

# Vector

```
#include<vector>
```

- A vector is perhaps the most general-purpose container
- It is a **dynamic** array => it can allocate memory as needed and so grow to accommodate any reasonable number of elements. This is an advantage over an array.
- Among others, we can construct an empty vector:

```
vector<type> v;
```

or one with num copies of a particular value:

```
vector<type> v(num, value);
```

- The type is specified in exactly same way as the class templates we have already seen, using angle brackets.



# Vector

- We can add/remove elements from the end of the array using **push\_back()** and **pop\_back()** and the memory will increase/decrease automatically.  

```
v.push_back(20);  
v.pop_back();
```
- Elements of a vector can be set and accessed using the **[ ]** operator as with an array.  

```
v[3] = 30;  
cout << v[5] << endl;
```
- But there's **no** checking that index is within array bounds
- If we access the elements of the vector via the **at()** function it will give an error if access an index outside the bounds of the array.  

```
cout << v.at(i) << "\n"
```
- A vector knows about its current size (unlike array), which can be queried via **size()**  

```
cout<<v.size()<< "\n"
```

# Vector Example

```
#include<vector>
:
int main(){
    vector<int> v1(10, 0);           // vector with 10 elements, all 0
    vector<float> v2;                // vector with no initial elements

    for (int i = 0; i < v1.size(); i++) v1[i] = i; // assign elements non-0 values
    for (int i = 0; i < v1.size(); i++)
        {cout << v1[i] << “, ”;} // access with no bounds checking
    cout << endl;

    for (int i = 0; i < 10; i++)
        {v2.push_back((float)i/2);} // add elements and increase size
    for (int i = 0; i < v2.size(); i++)
        {cout << v2.at(i) << “, ”;} // access with bounds checking
    cout << endl;

    return 0;
}
```

[VectorEg.cpp]

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

0., 0.5, 1., 1.5, 2., 2.5, 3., 3.5, 4., 4.5

# Iterators

- Iterators are objects that are very much like pointers.
- They allow us to cycle through a container in much the same way we can use a ptr to cycle through an array.
- Iterators are handled just like pointers: we can increment and decrement them and dereference (\*itr) them to give the object at that position in the container.
- Iterators are defined within the container classes so to declare one we need to qualify it with the container name.

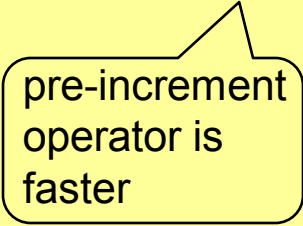
```
container<type>::iterator object
```

- There are actually several different types of iterator (input, output, forward, reverse ...) but we won't go into that here.

# Accessing Vectors via Iterators

- `i` is initialised to point to start of vector using **`begin()`**, which returns an iterator to start of vector.
- We can then increment `i` as needed & access the elements of the vector by dereferencing it (like a pointer).
- To determine when we have reached the end of the vector we use **`end()`**, which returns an iterator to **one past** end of vector.

```
#include<vector>                                [VectorEg.cpp]
:
int main(){
    vector<int> v(10,0);
    vector<int>::iterator i; //vec iterator
    // assign elements to vec via iterator
    int j = 0;
    for (i = v.begin(); i != v.end(); ++i) {
        (*i) = j;
        j++;
    }
    // access elements of vec via iterator
    for (i = v.begin(); i != v.end(); ++i) {
        cout << (*i) << ", ";
    }
    cout << endl;
    return 0;
}
```



pre-increment operator is faster

# String

```
#include<string>
```

- So far when we have needed a string we have used a null-terminated char array. This is the so-called C-string.
- The C-string is fast but can be clumsy to use.
- Another way to deal with strings is to use the container class string. This has several advantages:
  - **consistency:** a string is now a data type
  - **convenience:** all the standard C++ operators can be used
  - **safety:** array boundaries cannot be overrun
- We can construct (among others):
  - empty strings
  - strings from char arrays
  - strings from other strings.
- Note: If needed, a null-terminated char array of the string can be obtained via **C\_str()**.

```
string s1;    //empty
char* c = "alpha";
string s2(c); //from char
string s3(s2); //from string
```

```
const char c* = s1.c_str();
```

# String

- Strings can be assigned other strings, char arrays or even single chars.

```
string s1 = "beta";  
string s2 = s1;
```

- Strings can be concatenated with other strings or char arrays using the **+** operator.

```
string s3 = s1 + " and " + s2;
```

- Strings can be compared using comparison operators: **==**, **!=**, **<**, **>**. Strings are **equal** if have same size & same elements at same positions.

```
if(s1==s2) cout << "Same\n"
```

- Strings can be searched using the **find()** function. If a match is found the start index of the match is returned. If not it returns **string::npos** (think of this as -1).

```
if(s1.find("ta") != string::npos) cout << "Match Found\n"
```

# String Example

```
#include<string>
:
int main() {
    string s1("alpha");           // constructing strings
    string s2;
    string s3;

    s2 = "beta";                 // assigning strings
    cout << s1 << ", " << s2 << endl;
    s3 = s1 + " and " + s2;      // concatenating strings
    cout << s3 << endl;

    if (s1 == s2) cout << "s1 same as s2" << endl; // comparing strings
    else cout << "s1 not same as s2" << endl;

    unsigned int i = s1.find("ha"); // searching strings
    if (i != string::npos) cout << "Match found at " << i << endl;
    else cout << "No match found" << endl;
    return 0;
}
```

alpha, beta  
alpha and beta  
s1 not same as s2  
Match found at 3

# Map

```
#include<map>
```

- Maps are associative containers in which **unique** keys are mapped to values. Duplicate are not allowed.
- A **Key** is just a “name” by which we can refer to the value. The key can be any type/object (often string).
- The key can then be used to look up and retrieve the value from the map.
- Since the map is ordered by key, any key must have the comparison operators defined.
- So a map is a list of key/value **pairs** ...



# Pairs in C++

```
#include<utility>
```

- In C++, a **pair** is a template class which can be used to store two values together.
- We can construct a pair either using it's constructor, e.g.

```
pair<string, int> ("Carl", 27);
```

or using the **make\_pair()** utility, e.g.

```
make_pair("Carl", 27);
```

- The advantage of `make_pair()` is that the types are determined automatically rather than explicitly specified.
- The key and the associated value are retrieved from a pair via the members **first** and **second** respectively:

```
cout << "Name: " << p.first << ", Age: " << p.second << endl;
```

# Map

- Inserting entries into a map is done either using **insert()** with a pair or the **[ ]** operator.
- Accessing a value from the key can also be done by **[ ]**.

```
m.insert(make_pair("Carl", 27));  
m["Andy"] = 37;  
cout << m["Carl"] << endl;
```

- Like a vector, you can also loop through the map using an **iterator** and the **begin()** and **end()** functions.
- Dereferencing the iterator returns a **pair**, from which the key and value can be accessed via **first** and **second**.

```
map<string, int>::iterator i = m.begin();  
cout << (*i).first << " = " << (*i).second << endl;
```

- You can search the map for a particular key using **find()**.

```
map<string, int>::iterator i = m.find("Carl");  
if (i != dir.end()) cout << (*i).second << endl;
```

# Map Example

[Directory.cpp]

```
:
int main(){
    map<string, int> dir;
    dir.insert(make_pair("Carl", 43403));           // add to directory
    dir["Andy"] = 43409;
    dir.insert(pair<string,int>("Joost", 43386));

    cout << dir["Andy"] << endl;                  // access an entry

    map<string, int>::iterator i;
    for (i = dir.begin(); i != dir.end(); ++i) {    // print ordered directory
        cout << "Name: " << (*i).first << " Number:" << (*i).second << endl;
    }

    i = dir.find("Carl");                           // find someone's number
    if (i != dir.end()) cout << (*i).second << endl;
    else cout << "Name not in directory" << endl;

    return 0;
}
```

```
43401
Name: Andy Number:43409
Name: Carl Number:43403
Name: Joost Number:43386
43403
```