C++ Programming

Templates

Templates

- Templates are a generic way of writing functions and classes that are capable of operating on more than one data type without having to duplicate the code
  - In a way, the data type becomes a parameter in the definition of the templated function or class
- In the end, templates can match hand-written, less general code in terms of run-time and space efficiency, but are more general and only require a single implementation to be defined

Function Templates

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

template <typename Type>
Type maximum(Type a, Type b) {
   if (a > b)
      return a;
   else
       return b;
}

int main(){
   std::cout << maximum(2, 4) << endl;
   std::cout << maximum(2.0, 4.0) << endl;

}
```

A function template defines a generic function (`maximum`) that can handle parameters of different types, while the code stays the same. (Here, finding the maximum of two numbers).

At call, different types of parameters are passed (e.g., integer and real numbers), giving us two different functions in the end.

Class Templates

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

template <class Type> class calc
{
   public:
     Type multiply(Type x, Type y);
     Type add(Type x, Type y);
};

template <class Type> Type calc<Type>::multiply(Type x, Type y) {
   return x*y;
}

template <class Type> Type calc<Type>::add(Type x, Type y) {
   return x+y;
}

int main() {
   calc<int> c;
   cout << c.add(10,20) << endl;
}
```

We are declaring a class template …

We tell the member functions to use the type provided by the class

When we declare an instance of the templated class, we specify the type we want it to operate on; in this case we are making an integer variant of the class

Final Notes on Defining Templates

- You can create templates that take more than one type using
  `template<typename T1, typename T2>`
- `template<typename …>` and `template<class …>` syntaxes are interchangeable when defining basic templates
  - That said, the syntax does matter when creating things like template templates (yes, you can do that!)
- Templates can get rather complicated; for more advanced use cases, check out your favourite C++ reference …

Standard Template Library (STL)

- The C++ Standard Template Library (or STL) provides a collection of general-purpose templated classes and functions that implement many commonly used algorithms and data structures
- The STL is standardized across C++ implementations; for portability and maintainability of code, the STL should be used wherever feasible instead of re-implementing things unnecessarily
- At the core of the STL are:
  - Containers, algorithms, and iterators

Standard Template Library (STL)

Component	Description
Containers	Used to manage collections of objects of a certain kind; for example: vector, list, map, queue, stack, and so on
Algorithms	Act on containers; for example: perform initialization, sorting, searching, and transforming of the contents of containers
Iterators	Step through the elements of collections of objects

Standard Template Library (STL)

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

int main() {
   vector<int> numbers;

   for (int count=0; count < 10; count++) {
      numbers.push_back(count);
      cout << numbers[count] << " is in the vector!" << endl;
   }

   cout << "Added " << numbers.size() << " numbers to the vector!" << endl;
}
```

We have created a vector of integers … it starts empty, but we can add to it quite readily …

Standard Template Library (STL)

- There are too many goodies in the STL to go through them all here

- Please consult your favourite C++ reference for a complete listing, and API documentation for this
  - www.cplusplus.com and www.cppreference.com are both fairly decent in this regard