Templates

RMIT UNIVERSITY

Objectives

- Describe and Implement Template Functions
- Describe and Implement Template Classes

Generic Programming

- Imagine we have a function that needs to perform some operation to many different types.

```
void someFunction(const A&);

void someFunction(const Game&);
```

- The function body is identical – just the types differ. An approach might be to use inheritance to abstract this functionality to a common base class.

Generic Programming

- Java follows this approach – every type is a subtype of class Object – you can pass in any type as an Object.
- C++ does not have this hierarchy, so we would need to mimic it ourselves. But how do we deal with built-in types?

```
void someOtherFunction(int);

void someOtherFunction(double);
```

Generic Programming

- C++'s answer is to provide the notion of a **template**. A template is a way of constructing a function or a class regardless of type.
- This is the **generic programming** paradigm. We write one function definition, and the compiler handles the rest depending on types.

Function templates

```
template <class T>

void someFunction(const T&);
```

- The use of the keyword `template` tells the compiler that what follows is a template.
- The `<class T>` indicates that `T` is a generic type – this is called a **template parameter**.
- Some people prefer `<typename T>` instead of `<class T>`, as `T` could also be a built-in type.

Function templates

- When we use the function, the compiler will generate a new template function for each type that is used as a parameter for the function. Each version of a function template is called a **template function.**

```
someFunction(4);  // generates void
 someFunction(int)

someFunction(3.14159265389); // void
 someFunction(double)
```

- Only one instance of the function needs to be maintained – the compiler handles the rest.

Function templates

- We can use T within the function in the same way we'd use any other type.

```
template <class T>
void swap(T& a, T& b) {
   T stored = b;  // create a temp obj
   b = a;       // do the swap
   a = stored;    //
}
```

Function templates

- We can specify more than one parameter. The compiler will substitute appropriately.

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

swap (1, 4.0);  // call swap(int, double)
```

Function templates

- In this case, there needs to be a way of converting between type S and T if S and T are different types.
- This may require conversion constructors and conversion operators.
- This shows that using templates can involve a lot of hidden function calls that aren’t immediately obvious.

Function templates

- A good idea for implementing a function template is to get it tested and working for a particular type, and then make it generic.
- After making it generic, test it to ensure it still functions correctly for all types.

Function Templates

- C provides us with the macro syntax for defining generic functions:

```
#define MAX(a, b) (((a) < (b)) ? (b) : (a))
```

- These are not type-safe – you can use any text as a substitute for a and b above, providing it compiles. C++ templates enforce type safety. This is a good thing.
- There is also no return type specified.

Function templates

- Function templates enforce type checking by the compiler, and specify return types.

```
template <class T>
T& max(const T& a, const T& b) {
   return ((a < b) ? b : a);
}
```

Function Templates

- C also provides us with void pointers and function pointers as a way to implement generic programming.
- How would you write a generic function in c to process every element of an array?
- How would we implement this as a template function?

Class templates

- It is also possible to specify generic classes.
- These are usually used as **container classes**, or generic classes that are used to hold objects (such as a **stack**, **list**, **vector** or **queue**)

Class templates

```
template <class T>
class Vector {
   protected:
      T* array;
      int size, capac;
      void resize(); // resize array
   public:
      Vector(int=0);
      ~Vector();
      void push_back(const T&); // add to end
      T& pop_back(); // remove from end
      T& back(); // reference to last element
};
```

Class templates

- Note that we can use T as a parameter or member in any part of the class.
- As with a function template, we can also have multiple template parameters if we desire.

Class templates

- Member functions of a template class require a slightly different syntax if defined outside the class (which they should be!)

```
template <class T>

Vector<T>::Vector(int s=0) : array(0), size(s),
 capacity(s) {
   if (size != 0) {
      array = new T[size];
   }
}
```

Class templates (cont.)

```
template <class T>
Vector<T>::~Vector() {
   delete [] array;
}

template <class T>
T& Vector<T>::back() {
   return array[size-1];
}
```

Class templates (cont.)

```
template <class T>
void Vector<T>::push_back(T& obj) {
   if (size == capacity) {
      // increases capacity
      resize();
   }
   array[size] = obj;
}
```

Exercise

- Implement the `resize()` and `pop_back()` functions in a generic manner. `pop_back()` just removes the last element (without returning it).

Container classes

- We’ve looked at a vector of a generic type. What about a vector that holds *any* type. In effect, a container that holds unrelated objects.
- With C++, this is difficult to do – Java’s Object class makes this easy for it.
- How often do you really need it?

Templates and inheritance

- It is possible to have template classes inherit from other classes.
- The base classes can be either templates or non-templates.

```
template <class T> Base { };
class Base2 { };
template <class T> Derived : public Base<T> { };
template <class T> Derived2 : public Base<int> { };
template <class T> Derived3 : public Base2 { };
class Derived4 : private Base<int> { };
```

Templates and inheritance

- In these examples,
  - `Derived` is a template class with a templated base.
  - `Derived2` is a template class inheriting from a *non*-templated base.
  - `Derived3` is a template class also inheriting from a *non*-templated base
  - `Derived4` is a non-template class inheriting from a *non*-templated Base.

Member templates

- It is possible to have members of a class that are also templates. These are **member templates**.

```
class Game {
   public:
      template <class T>
      Game(const T&) { } // template constructor

      template <class T>
      void someFunction() { } // template func.
};
```

Member templates

- A template constructor will never generate a copy constructor – you will need to write your own template version, or accept the compiler default.
- Member templates cannot be virtual, as this breaks the `vtbl` approach to virtual functions.

The `export` keyword

- If a template function (not a member function of a template class or member template) is defined in a separate compilation unit, the `export` keyword must be used.
- C++ handles template functions like inline functions – it expects that they are defined in the same compilation unit.
- To get around this, explicitly declare the template function definition `export`.

The `export` keyword

```
// .h file
template <class T>
void someFunction(const T&);

// .cpp file
export template <class T>
void someFunction(const T&) {
  //..
}
```

Default template parameters

- It is possible to specify default parameters for a class template. This specifies a type if one isn't provided:

```
template <class T = int>

class Stack {
 // templatised implementation

};


Stack<double> doubleStack;

Stack intStack;
```

Explicit Specification

- If a template type cannot be worked out by the compiler, we need to explicitly specify it:

```
template <class T>
T* createSomeObject() {
   return new T();
}

// explicit specification
Game* g = create<Game>();
```

Explicit Specification

- Another, more familiar, example:

```
template <class S, class T>
S static_cast(const T& orig) {
   // let compiler convert via conversion
   // constructors or operators
   return orig;
}

int j = static_cast<int>(4.765);
```

Template specialisation

- Sometimes, we want to provide a different implementation for a template class or function depending on type.
- E.g. a comparison operator:

```
template <class T>
bool less_than(const T& lhs, const T& rhs) {
   return (lhs < rhs);
}
```

Template specialisation

- If used with pointers, this comparison operator will compare the addresses – not the underlying objects. This may not be what we want.
- We can specialise:

```
template <class T>
bool less_than(const T* lhs, const T* rhs) {
   return (*lhs < *rhs);
}
```

Template specialisation

- We can do the same for classes:

```
template <class T>
class vector { };  // general vector

template <class T>
class vector<bool> { };  // specialisation
```

- Here, we can provide a different implementation for a vector of boolean types (the standard library does this).