Tute 11: Templates

Review: Templates

What is a template?

Templates provide a way to program C++ classes and functions without regard to the data that they will be working with (similar to macros). The most common use of templates in C++ is for containers, which are data structures intended to hold data, but not necessarily any particular kind of data (ie, a link list, vector, etc.).

If a class is a blueprint for creating an object, think of a template as a blueprint for creating a class or function, and as classes are evaluated to create objects at runtime, templates are evaluated to create classes at compile time. This means that all template code must be able to be executed at compile time and all types must be known at compile time (which also means that templates can be used to write entire programs that execute at compile time - this is called template metaprogramming).

What is the difference between a class template and a function template?

A class template is defined like so:

```
template <typename T>
class Pair
{
public:
	T a;
	T b;
};
```

While a function template is defined like so:

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

Notice that the > operator has been used in the function max(), this means that the data type T must have overloaded that operator for the template function to work. In this way templates can

be used to execute arbitrary functions/methods on any data type, as long as it is valid. This can be thought of as a form of compile time duck-typing, ie:

```
class A
{
public:
	void foo() { … }
};

std::vector<A> v;
for (auto &a : v)
	a.foo();
```

This code will only work if v is populated with a data type that has the method foo(). This can also be used to circumvent the need for inheritance hierarchies in some cases. Depending on what you're doing, this may or may not be a good thing.

When is the typename keyword used?

typename can be used whenever a template data type is declared, eg:

```
template <typename T>
```

although it is also valid to write this as:

```
template <class T>
```

as there is no difference between the two declarations (even primitives such as an int would be a valid argument, even though in the second case class is specified instead of typename).

The typename keyword is also used to clarify circumstances where a type is a subtype of a template type, ie:

```
class A
{
public:
	struct SubA {};
};

template <typename T>
class B
{
public:
```

```
        typename T::SubA subA;
};
```

Here SubA is assumed to be a subtype of T (which in this case it is) and the typename keyword is used to clarify this.

When is the template keyword used?

The template keyword can be used to declare a class or function template, eg:

```
template <...>
class A { … };

template <...>
void f(...) { … }
```

It is also used in the same manner as typename when dealing with function template dependencies, eg:

```
class A
{
public:
        template <typename T>
        static T staticTemplateFunc() { … }

        template <typename T>
        T memberTemplateFunc() { … }
};

template <typename T>
void func()
{
        T::staticTemplateFunc<int>();           // ambiguous
        T::template staticTemplateFunc<int>();  // ok

        T t;
        t.memberTemplateFunc<int>();            // ambiguous
        t.template memberTemplateFunc<int>();   // ok
}

int main(int argc, char **argv)
{
        func<A>();
```

```
    return 0;
}
```

Programming Concepts: More Templates

How can templates be specialised?

Templates that have been defined for generic types:

```
template <typename T>
class A
{
    T t;
    …
};
```

Can be redefined for specific types like so:

```
template <>
class A <bool>
{
    bool b;
    …
};
```

This allows you to provide different implementations for generic classes or functions based on what types have been passed to them.

How do we use both class and function templates in the same class?

Easily:

```
template <typename T>
class A
{
public:
    template <typename S>
    void foo(T t, S s) { … }
};
```

This could then be used as:

```
A<bool> a;
a.foo(true, “Hello”);
```

How is template metaprogramming used?

Template metaprogramming uses structs or classes combined with templates to calculate values at compile time, typically storing results using an enum. For this to work usually two or more structs are required, one for a recursive operation, and one for termination, eg:

```
template <int n>
struct factorial
{
	enum { value = n * factorial<n - 1>::value };
};

template <>
struct factorial<0>
{
	enum { value = 1 };
};

int x = factorial<0>::value; // 1
int y = factorial<4>::value; // 24
```

Note that the second struct defines a particular instantiation that is used whenever the value 0 is used to construct the factorial.

Compilation: Libraries

What is a library?

A C++ library is typically a file with a .a or .so extension. Libraries can either be static (they are linked at compile time) or dynamic (they are linked at runtime). A library is usually a collection of .o files that provide implementations for different functions/classes along with one or more .h files that provide function/class declarations.

How can we create/use a library?

To create a library you first need to compile the necessary object files as you would normally, you can then bundle these object files together into a library with the following command:

```
ar rs library.a *.o // ld could also be used to create a .so library
```

This will archive all .o files into a library called library.a, this can be linked statically by including it in the makefile as a .o file (or if you're using the Makefile from the sample solution, you can add it to the LDFLAGS. Alternatively you can link with the command -lrary (-l takes the place of the lib prefix) and you can store the library in a separate folder, however to do this you need to add its location to the LD_LIBRARY_PATH.

Errors: None this week

Debugging: None this week

Exercises:

Write a program to calculate the nth fibonacci number at compile time.

```
#include <iostream>

template <int n>
struct fib
{
	enum { value = fib<n - 1>::value + fib<n - 2>::value };
};

template <>
struct fib<0>
{
	enum { value = 0 };
};

template <>
struct fib<1>
{
	enum { value = 1 };
```

```
};

int main(int argc, char **argv)
{
        std::cout << fib<10>::value << “\n”;
        return 0;
}
```

Provide templated functions to return a reference to an object of type T, accounting for data that is passed in as a pointer, and data that isn’t.

```
template <typename T>
T &ref(const T &t) { return t; }

template <typename T>
T &ref(const T *t) { return *t; }

…

Object *o1 = new Object();
Object o2;
Object &r1 = ref(o1);
Object &r2 = ref(o2);
```