COMP322 - Introduction to C++

Lecture 12 - More about Templates

Robert D. Vincent

School of Computer Science

31 March 2010

Template specialization

- Ideally, we want to express any non-trivial concept exactly once.
- C++ templates allow us to do exactly that.
  - Class templates
  - Function templates
- Either one allows us to specify the implementation of an abstract data type or algorithm, *independently of the subtypes that may be manipulated*.
- The result can get large and complicated if we have to instantiate the template for many different types.
- Template specialization exists to minimize this.

Specialization example

- Consider a generic `Vector` class, which can any number of objects of any type:

```
template <class T> class Vector {
  T *v;
  int length;
public:
  Vector();
  explicit Vector(int);
  T & operator[](int i);
  // ...
};
```

- We can specialize this template for a specific type:

```
template <> class Vector<bool> {
  // code for boolean bit vectors
};
```

- We remove any parameters with fixed values from the parameter list.

Specialization example

- Many vectors will contain pointer types. We can use specialization to handle this case more efficiently:

```
template<> class Vector<void *> {
  void **v;
  int length;
  // ...
  void *& operator[](int i) { return v[i]; }
};
```

- This template specialization will be specific to `void *`, but since this is a generic pointer type, we could extend it to others.

Specialization example

- We can derive a partially specialized template from the `void *` implementation, to handle other pointer types:

```
template<class T> class Vector<T *> : private Vector<void *> {
public:
 typedef Vector<void *> Base;
 // ...
 T * & operator[](int i) {
   return reinterpret_cast<T *&>(Base::operator[](i));
 }
};
```

- This template will match *any* pointer type.

```
Vector<int *> vpi;  // Matches above template, T=int
Vector<void *> vpv; // Matches specialized void * vector
Vector<Shape> vs;   // Matches generic template
Vector<bool> vb;    // Matches specialized bool vector
```

Specialization details

- The generic template must be in scope and appear before any specializations.
- Any parameter which is set is removed from the parameter list:

```
template <class T, int length> class fixedList {
  // ...
};

template <int length> class fixedList<bool, length> {
  // ...
};

template <> class fixedList<int, 512> {
  // ...
};
```

Typename keyword

- Historically, C++ re-uses the `class` keyword to declare template parameters which may be any type.
- Arguably, this is confusing.
- More recent implementations have added the `typename` keyword to address this confusion:

```
template <typename T> class A {
  T *data;
  int sz;
public:
  /* ... */
}
```

Templates and inheritance

- Inheritance is not preserved across templates:

```
template <typename T> class A { /* ... */ };
class B { /* ... */ };
class C : public B { /* ... */ };

int main() {
  B b;
  C c;
  A<B> ab;
  A<C> ac;
  b = c;   // legal, as B is derived from C
  ab = ac; // error!
}
```

Deriving templates from templates

- We can derive a class template from another template.
- Normally the template parameter will be used as the parameter of the base class:

```
template<class T> class A { /* ... */ };
template<class T> class B: public A<T> { /* ... */ };
```

- A range of situations are possible:

```
template <class T> class C { /* ... */ };
template <class T> class D : public C< D<T> > { /* ... */ };
```

- Or we could inherit from two different template classes:

```
template <class T> class E { /* ... */ };
template <class T> class F { /* ... */ };
template <class T, class X> class G : public E<T>, public F<X> {
};
```

Member templates

- A class or class template can contain templates as members:

```
template <typename T> class A {
  // ...
public:
  template <typename X> A(X &arg);
  // ...
};
```

- This syntax would allow us to construct an `A` from an object of an arbitrary type - although presumably a type with some well-known set of operations.