Part X

Advanced C++ topics

References

- The following are highly regarded books. They are fairly in-depth, and I haven't read them in their entirity.
- However, if you want to write robust code that will not surprise you or others using your code, and which can be extended in the future, you should at least have a look at them.
- Effective C++, Scott Meyers
- Exceptional C++, Herb Sutter
- Modern C++ Design, Andrei Alexandrescu

Base class construction

- Suppose you have a base-class containing some data, with a relatively complex initialization.
- Any derived class should not copy-paste the base's constructor.
- When constructing any object, the base-object is always constructed first, and this can be called from the derived class:

```
class SpecialVector : Vector{
  SpecialVector(int SIZE, int d) : Vector(SIZE), m_data(d){}
  int m_data;
};
```

- Note that the Vector initializer comes first. You could put it later, but it will always be initialized first.
- `gcc` warns (with `-Wall`) if the constructor order is different from what will actually happen.
- See `Examples/baseclass.C`

Resource Acquisition Is Initialization

- A good programming practice to abide by is RAII.
- The idea is that all finite resources such as allocated memory, file handles, etc. are handled through object instances.
- Any class that handles any resource must ensure that once the object ceases to exist, the resource it handles is freed (or is passed onto another object)
- For example, a `Vector` object which allocates memory should have:

```
Vector::Vector(size_t s){
  m_data = new double[s];
}
Vector::~Vector(){
  delete[] m_data;
}
```

- ... as well as similar functionality from other constructors.

RAII ctd

- However, for more complex objects, more care must be taken:

```
DoubleVector::DoubleVector(size_t s){
  m_data = new double[s];
  m_data2 = new double[s * 10];
}
DoubleVector::~DoubleVector(){
  delete[] m_data;
  delete[] m_data2;
}
```

- What happens if the first allocation succeeds and the second fails?
- If you do not catch exceptions, then the `std::bad_alloc` will propagate all the way up the call-stack and cause the program to terminate.
- If you do handle exceptions, then the object will potentially be left in an uninitialized state, and the first block of data will be leaked.
- You should put a `try` / `catch` block around the second `new` statement, and `delete m_data[]` before re-throwing the exception.

Base class destruction

- We saw previously that the base class is constructed before the derived class.
- Conversely, on destruction, the derived class is destroyed first, before the base class.
- If the derived class constructor throws an exception, then any base classes already constructed are destroyed (in order).
- See `Examples/raii.C` for an example.

Polymorphism and RAII

- Suppose you have a simple polymorphic class with dynamically allocated data:

```
class Base{
public:
  Base(){
    m_data = new int[10];
  }

  ~Base(){
    delete[] m_data;
  }
private:
  int* m_data;
};
```

```
class Derived : public Base{
public:
  Derived(){
    m_myData = new int[10];
  }

  ~Derived(){
    delete[] m_myData;
  }
private:
  int* m_myData;
};
```

Polymorphism and RAII

- You would naturally use:

```
Base* b = new Derived;
delete b;
```

- However, this leaks the memory allocated by the `Derived` object; its destructor is never called.
- Only the `Base` destructor is called.
- The solution is to make the `Base` destructor `virtual`:

```
virtual ~Base(){
  delete[] m_data;
}
```

- Now `delete b` causes first the `Derived` destructor then the `Base` destructor to be called.
- See `Examples/destructors.C` for the full example.

Polymorphism and RAII

- Note that any memory allocated by member data of the `Derived` object is also leaked.
- For example, even if `Derived` only contains a `std::vector<int>`, and `Base` contains no internal data, the `Base` class must still have a virtual (and empty) destructor.
- Destructors of member data are only called when the class's destructor (auto-generated or explicitly written) is called.
- Destructors are not virtual by default because making them so results in extra code. The principle is "don't pay for what you don't need". However, a non-virtual destructor on a polymorphic base-class is almost always wrong.

SFINAE

- You may see references to "Substitution Failure Is Not An Error"
- This refers to a C++ feature that allows us to selectively allow compilation of templated functions.
- It prevents instantiation of templated functions that result in ill-defined types.
- An example is the use of `std::enable_if<bool b, typename T>`
- If the parameter `b` is true, then: `std::enable_if<true, T>::type` is the same as `T`.
- If `b` is false, then `std::enable_if<false, T>::type` does not exist.

SFINAE example

From **Examples/sfinae.C**:

```
template<typename T>
void print(T t, typename
    std::enable_if<std::is_pointer<T>::value, char>::type = 0){
  std::cout << "Pointer to " << *t << std::endl;
}

template<typename T>
void print(T t, typename
    std::enable_if<!std::is_pointer<T>::value, char>::type = 0){
  std::cout << "Value = " << t << std::endl;
}

int a = 9;
print(a);
print(&a);
```

SFINAE example

- The first parameter is each `print` is the plain `T` so that the compiler can deduce the type.
- The second parameter is either non-existent or an `char`, whose value defaults to zero.
- This looks like partial-template specialization, but isn't.
- What it really is are two overloaded functions called `print`, only one of which will ever be considered for any particular type `T`. because the second parameter does not give a valid type in one case.
- The non-working version does not give an error because Substitution Failure Is Not An Error.

SFINAE example

Similarly, you can make the functions differ only on their return type:

```
template<typename T>
typename std::enable_if<std::is_pointer<T>::value>::type
    print2(T t){
  std::cout << "Pointer to " << *t << std::endl;
}

template<typename T>
typename std::enable_if<!std::is_pointer<T>::value>::type
    print2(T t){
  std::cout << "Value = " << t << std::endl;
}
```

Specifying no type for `enable_if` defaults to `void`.
The use of `enable_if` often requires the use of a dummy function parameter or template parameter.

Curriously Recurring Template Pattern

- You may sometimes need to implement a base class as an interface for a templated class, but whose functionality depends on the templated class.
- A classic example is cloning. You want the functionality:

```
Vehicle* myCar = new Car;
Vehicle* anotherCar = myCar->clone();
```

- This cannot be implemented in `Vehicle` normally as `Vehicle` does not have access to all elements of `Car`, and so would slice off any data not contained in `Vehicle`.
- We could implement a `clone` function for every new type derived from `Vehicle`.
- That seems error-prone and wasteful.

Curiously Recurring Template Pattern

- Instead, insert a templated class (see `Examples/crtp.C`):

```
class Vehicle{
public:
  virtual Vehicle* clone()const = 0;
};
template<typename Derived>
class VehicleInterface : public Vehicle{
  virtual Vehicle* clone()const{
    return new Derived(*static_cast<const Derived*>(this));
  }
};
class Car : public VehicleInterface<Car>{};
```

- Now, the `clone()` function works because the fully-derived type is known and used at instantiation.
- This is "Curiously Recurring" because it seems to occur a lot, not because it is recursive.
- This approach can be used to reduce copy-pasting for other class member functions as well.

Has-a or is-a?

- In the first lecture series, you implemented a class hierarchy structure.
- One important approach to designing a class structure is to consider whether a kind of object is a particular kind of more general object, or whether the more general object should contain this new object type.
- This is abbreviated to a *is-a* or *has-a* relationship.
- For example, a **Car** is a kind of **Vehicle**:

```
struct Car : public Vehicle{ };
```

- but it contains an **Engine** (has-a) and other components:

```
struct Car : public Vehicle{
  Engine m_engine;
};
```

Other considerations

- Another relationship between classes is the *implemented-in-terms-of* approach.
- For example, a (horribly inefficient) Vector class could be implemented in terms of a List class.
- The `Vector` should not be convertible to a `List`, and it will probably not use any of the internal implementation details of `List`.
- Thus we should have:

```
class Vector{
public:
  Vector(size_t s);
  double operator[](size_t i)const;
private:
  List m_list;
};
```

More object-orientation

- If you implement a polymorphic set of classes, ensure that whatever constraints the generic interface (base-class) suggests or enforces are also enforced by all derived classes.
- Recall the example from the exercises in the first C++ course: a `Circle` is *not* a kind-of `Ellipse`.
- More details of object orientation can be found in Sutter's Exceptional C++: Item 24.

Factory construct - why

- In modular programs, or at least ones where multiple options are open to the user, you often need a function like:

```
const Shape* getShape(const std::string& name){
  if(name == "Sphere") return new Sphere;
  if(name == "Triangle") return new Triangle;
  return nullptr;
}
```

- Maintaining this kind of function is error-prone, and needs updating every time you add a new object.
- A better (more complicated to set up, but easier to maintain) approach is the Factory construct.

Factory construct

- Consider a mapping:

```
std::map<std::string, const Shape* (*)()> factory;
```

- We can now use:

```
const Shape* getShape(const std::string& name){
  if(factory.find(name) != factory.end()){
    return factory[name]();
  }
  return nullptr;
}
```

- Note that the stored object is a pointer to function that, when passed no parameters, returns a pointer to a constant Shape.

Factory construct - initialize

- Now, each Shape-derived type needs to take the form:

```
class Sphere : public Shape{
public:
  static const Shape* create();
private:
  static const bool isInFactory;
};

const Shape* Sphere::create(){
  return new Sphere;
}

const bool Sphere::isInFactory =
factory.insert(
  std::make_pair("Sphere", Sphere::create)
).second;

Sphere dummy_sphere;
```

- ... recalling that the **insert** function returns an iterator and a boolean indicating whether insertion succeeded.

Factory construct - how?

- The reason that this approach works is that the `static const bool` members of the `Sphere` and `Triangle` have to be initialized before your code starts.
- They are therefore initialized, before `main` is called, using the `insert` function call.
- There is no guarantee about the order in which the various function calls occur.
- If the `dummy_sphere` variable were not specified, the compiler would not necessarily generate code to initialize the member data of an unused class.
- See `Examples/factory.C` for the full code.

Outline

Templated functions

- Functions can be templated:

```
template<typename T>
T sum(const std::vector<T>& v);
```

- They cannot be partially specialized:

```
template<typename T, typename S>
std::vector<T> product(const std::vector<T>&, const
std::vector<S>&){...}

template<typename T>
std::vector<T> product(const std::vector<T>&, const
std::vector<T>&){...} // Invalid
```

- This is because otherwise it becomes difficult to separate overloaded function calls from partial template specialisations.
- See `http://ww.gotw.ca/publications/mill17.htm` for a discussion of the problems.

Ambiguous templated functions

- You may have tried to compile:

```
double dT = calcTimeStep();
double dTActual = std::min(dT, 1);
```

  and been surprised when it failed.

- Even though '1' is an integer, surely the compiler can figure out that you want `std::min<double>`?
- No (not without a lot of type traits to indicate which types can be promoted).
- There are two solutions to the above:

```
double dTActual = std::min(dT, 1.0);
double dTActual = std::min<double>(dT, 1);
```

- The first is probably nicer.

Ambiguous templated functions

- The reason this fails is that `std::min<T>(const T&, const T&)` only has one template parameter, and a consistent `T` cannot be deduced.
- The C++ standard specifies this form, rather than a more general one.

Scope of templated classes

- If you have a templated class with a templated base, you may encounter confusion:

```
template<typename T>
struct A{
  int mySize(){
    return sizeof(T);
  }
};

template<typename T>
struct B : A<T>{
  void print(){
    std::cout << mySize() << std::endl;
  }
};
```

- will fail to compile:

```
error: there are no arguments to 'mySize' that depend on a
    template parameter, so a declaration of 'mySize' must be
    available
```

- ... which is not very revealing.

Scope of templated classes

- The reason is that C++ has a two-phase name look-up.
- The first time that the compiler parses a class or function it must be able to work out what the types of any non-template-parameter-dependent functions or types are.
- In the previous slide, `mySize()` when called does not depend on a template parameter, and therefore the compiler looks through the set of functions, variables, and types that are available, without knowing `T`.
- Since `mySize()` will actually be found in the base-class only when `T` is known, this phase fails.
- The solution is to make the call clearly depend on `T`:

```
std::cout << this->mySize() << std::endl;
std::cout << A<T>::mySize() << std::endl;
```

- Either of these causes name-look-up for `mySize()` to be delayed until the second phase, when `T` is known.

Scope of templated classes

- Further problems arise if you want to call a templated member function of a templated base.
- If the template parameter can be deduced, then it's simple:

```
template<typename T>
template<typename S>
int A::itsSize(S a){
  return sizeof(S);
}

template<typename T>
void B<T>::print(){
  std::cout << this->itsSize(2) << std::endl;
}
```

(See `Examples/templatedBase.C`)

Calling a templated member

- If you want a different (or non-deducible) template parameter, you must use:

```
std::cout << this->template itsSize<double>(2) << std::endl;
```

- Otherwise, `this->itsSize<double>` is interpreted as an unresolved overloaded function, followed by a less-than sign, followed by `double`
- This will not succeed (possibly unless you've overloaded a very weird `operator<`, and even then I think it might be impossible).

typename

- Due to the two-phase look-up, the compiler sometimes needs to be told in advance whether a typename or something else will result.

```
template<typename X>
struct Y : X{
  using typename X::F;
  void f(){
    int x = F();
  }
};
```

- The `F()` construct on its own could either be a function call, or a construction of an object of type `F` using no parameters.
- Further, `F` is only brought into scope by the `using` directive.
- The `typename` tells the compiler to expect `F` to be a type.
- If `typename` was absent then `X::F` would be assumed to be a function.
- Whichever is required will be checked against the first phase's deductions when `X` is known, on the second phase.
- See `Examples/typename.C` for a complete demonstration.