C++ Essentials

COSC342: Lab 04

March 26th 2017

1 Introduction

We've used C++ a bit in the labs, but we'll be looking at larger programs soon as we look at the ray tracer and OpenGL pipeline. Before we get into these larger programs, its worth taking a step back and looking at C++'s object model and a few other things. While the basic ideas of objects, instances, and inheritance will probably be familiar to you from Java, you may have already noticed that things work a little differently in C++.

C++ is a very flexible language, which gives you a lot of power, so it is possible to write some pretty nasty code if you feel inclined. However, it is also a language that has a lot of *idiom* associated with it – there are typical uses of the language, and ways of expressing ideas which have become commonly accepted. The compiler does not *enforce* this, but good C++ programmers adhere to them to write consistent code which avoids a lot of potential problems that come from having more ability to manage memory directly.

The reward that comes from the flexibility of the language, is the ability to express yourself in a way that reflects the problem. This is in contrast to languages which adhere tightly to a single paradigm. For example, in Java everything you define is an object, but not every concept has a neat object associated with it.

> Since this lab has a lot of notes in it, the things that have some programming to do are highlighted in blue boxes, like this.

I'm also aware that this is a long document. There is more detail here than you need for COSC342 assignments and labs, but I've made an effort to explain how things work, at least at a basic level.

1.1 Why C++?

There are a lot of programming languages out there, and some are better than others for particular tasks. C++ is a general purpose programming language, so can be used for a wide variety of tasks. Other languages that are often seen to compete with C++ are C, Java, C#. Compared to those languages, I would say that the advantage of C++ is that it gives you more *choice about how you express yourself*. This choice comes at a cost – the language is complex, and most programs only use a subset of the tools it provides. For COSC342, there are a number of advantages that C++ offers. Compared to Java and C#:

- Since it was designed for systems programming, C++ programs are generally faster. This is important for real-time processing (OpenGL) or intensive compute jobs (Ray Tracing).
- Although it is cross-language, OpenGL's syntax is most similar to C, so integrates more neatly with C++.
- Operator overloading allows us to write matrix and vector operations naturally as `u = A*v + 0.5*w;`, rather than `u = A.multiply(v).add(w.multiply(0.5));`.

Compared to C:

- C++ offers better tools for structuring and managing data through objects, standard library containers, etc. A Ray Tracer, for example, is one of the places where objects and inheritance make a lot of sense.
- C++ has safer ways to deal with memory and other resources, a stronger type system, and other features which can help avoid many types of error.

2 Variables, Pointers, and References

You should be familiar from C with normal variables and also with pointers.

```
int var = 3;      // var is an integer variable, initialised to 3
int* ptr = &var;  // ptr is a pointer to an integer, initialised to point to var
```

C++ also has references, which are similar to object references in Java, but can refer to any type:

```
int& ref = var;  // ref is a reference to an integer, referring to var
```

References are like pointers in many ways, but different in important aspects:

- You cannot have an uninitialised or null reference. All references must refer to an actual value type somewhere.
- References are an alias of a variable, rather than a memory address. As a result you don't need to 'dereference' references to get to the underlying value. To set `var` to 5 through the pointer you'd write `*ptr = 5;`, but with the reference you just write `ref = 5`.
- You cannot perform 'pointer arithmetic' on references – in the example above, `ptr = ptr + 2` moves the pointer to a new memory location (possibly unsafe!), but `ref = ref + 2` adds 2 to the variable referred to by `ref`, so is equivalent to `var = var + 2`.

While references are implemented as pointers, they cannot be null or refer to uninitialised memory, so avoid many of the problems with pointers.

The other key difference with references and pointers is how you get to members if you have a reference/pointer to an object. Suppose we have an object instance `obj` that has some method called `method` and a member variable called `var`, which we'll assume is an integer. We access the method/member using `->` if we have a pointer to the object, and with `.` if we have the instance directly or via a reference:

```
Object obj;
Object* ptr = &obj;
Object& ref = obj;

// For an actual instance, use .
obj.var = 5;
obj.method();

// For a pointer, use ->
ptr->var = 5;
ptr->method();

// For a reference, use .
ref.var = 5;
ref.method();
```

2.1 Memory Management in C++

In C, dynamic memory is allocated with `malloc` (or `calloc`) then deallocated with the `free` function:

```
double *dptr = (double *) malloc(sizeof(double));    // A pointer to a double
int n = 10;
double *darr = (double *) malloc(n*sizeof(double));  // An array of n doubles
...
free(dptr);
free(darr);
```

In C++ the basic memory operations are replaced by `new` and `delete` (although C-style management is still valid, the two cannot be mixed).

```
double* dptr = new double;      // a pointer to a double
int n = 10;
double[] darr = new double[n]; // an array of n doubles
...
delete dptr;
delete [] darr;
```

there are a number of differences between `new`/`delete`, and `malloc`/`free`, but some of the main ones are:

- `new` returns pointers of the appropriate type, rather than a `void*` which requires casting to the actual type.
- `new` and `delete` make a distinction between arrays and pointers.
- `new` automatically determines the amount of memory required.

This removes some of the errors that can arise from pointers, but one significant issue that remains is that the programmer is responsible for ensuring that dynamically allocated memory is released. C++ offers a safer alternative (or rather a set of alternatives) through the standard library's *smart pointers*. There are several smart pointers provided, but the most common ones are unique and shared pointers.

```
#include <memory> // Header defining smart pointers
...
std::unique_ptr<double> dptr(new double);
std::unique_ptr<double[]> darr(new double[n]);
// no need to delete
```

A unique pointer, as its name suggests cannot be copied (although it can be moved to another unique pointer if returned from a function, for example) – there can only be one pointer to the resource. An alternative is a `shared_ptr`, which allows for multiple pointers to the same resource.

These pointers are 'smart' in that they automatically free the memory they contain. This is not, however, the same as garbage collection in Java. The deallocation of the memory with smart pointers is entirely deterministic, and occurs as soon as the pointers go out of scope (or the last pointer to a resource in the case of a shared pointer).

3 A Simple C++ Object

We'll start with a simple C++ object, which is defined in two files:

- `Object.h` – the *header* file, which gives the definitions of the object, it's member variables, and methods.
- `Object.cpp` – the *implementation* file, which provides the code to implement the methods of the object.

If you look in the header file, you'll see that the `Object` definition is divided into blocks labelled as being `public:`, `protected:`, and `private:`. These terms have very similar meanings to in Java, but affect all the following elements (members or methods), rather than being defined per-element like in Java. There's no equivalent of Java's `package` access modifier within a single class in C++.

While you can mix up the the various access modifiers, it is usual to list the `public` members first, then `protected`, then `private`. This means that the parts of the class definition that people will actually use are first in the file. The separation of header and implementation also means that the header file can act as a quick reference for the available methods etc. without being cluttered with implementation details.

3.1 Construction, Copying, Destruction

The first methods defined for the `Object` class deal with creating, copying, and destroying instances of the class:

```
Object();
Object(int value);
Object(const Object& obj);
~Object();

Object& operator=(const Object& obj);
```

Several of these are quite core to the way C++ objects work:

1. `Object();` – a *default constructor*, which takes no arguments.
2. `Object(const Object& ob);` – the *copy constructor*, which creates a new object as a duplicate an existing object.
3. `~Object();` – the *destructor* which frees or deallocates any resources held by the object.
4. `Object& operator=(const Object& obj);` – the *assignment operator* which copies the details of one object to another.

If no constructors (of any type) are provided, the compiler will generate a default constructor If any of the other three are not defined, then the complier will generate them. For objects which just contain plain data types (`int`, `double`, etc) or instances of classes that deal with their own resources (such as `std::string`s) the compiler generated ones are probably OK. If your class deals with some resource that needs allocating or deallocating, then you may have some work to do.

It is also possible to explicitly say that you want to use the compiler-generated versions, or to stop the compiler from doing so. This can be done to make an class where you cannot copy instances:

```
class NonCopyable {
public:
 NonCopyable() = default;                              // Default c'tor generated by compiler
 NonCopyable(const NonCopyable&) = delete;             // No copy constructor
 ~NonCopyable() = default;                             // Destructor generated by compiler
 NonCopyable& operator=(const NonCopyable&) = delete; // No assignment allowed
};
```

In more recent versions of the C++ standard, there are two additions to this list:

1. `Object(Object&& obj)` – *move constructor*, which creates a new object by taking over the resources of an existing one.
2. `Object& operator=(Object&& obj)` – *move assignment operator* which moves the resources held by one object to another.

We won't need to worry about these, but the `&&` decorator indicates an *rvalue*, which is a concept that can be a bit subtle. Basically a *rvalue* is anything that can appear on the right hand side of an assignment operator, and these are used to avoid the overhead of creating temporary copies when returning objects from functions etc.

The *destructor* is something that is different to Java objects. In Java memory used by objects is managed by the garbage collector, which cleans up objects that are no longer referenced. The garbage collector runs when the JVM thinks it should, so can be hard to predict. In C++ the destructor is responsible for freeing up any memory (or other resources such as files or network connections) held by an object.

The timing of the destructor depends on how the objects are created. In C++ objects are either created directly (on the program stack) like so:

```
Object obj;
```

or via the `new` operator (which adds them to the program heap or free store) like so:

```
Object *ptrToObj = new Object();
```

Objects on the stack are destroyed when they go out of scope, whereas the objects that are created via `new` remain until `delete` is called on the pointer returned by `new`:

```
delete ptrToObj;
```

Note that smart pointers can be used to avoid the need to call `delete` explicitly, and will call it once the object instance is no longer referenced.

3.2 Initialiser Lists

If you look at the implementation of `Object`'s constructors, you'll see that they have a syntax which may be new to you. For example, the default constructor is:

```
Object::Object() : id_(++objectCount_), value_(0) {
  std::cout << "Object " << id_ << " created with Default Constructor" << std::endl;
}
```

The body of the constructor only contains the reporting code which is not actually part of the construction process. The actual initialisation of the member variables is done with an *initialiser list*, introduced with the colon after the constructor's definition. You could write (for example)

```
Object::Object() {
  id_ = ++objectCount_;
  value_ = 0;
}
```

which is valid C++, but the initialiser list is better C++ style as it allows you to explicitly initialise all of the member variables before they can be used in the body of the constructor. With initialiser lists it is quite common to see constructors with empty bodies.

3.3 The `this` Keyword

If you look in the implementation of the assignment operator, you'll see that we use `this` to refer to the current object. This is the same as Java's `this`, except that it is a pointer not a reference so we have to use `->` rather than `.` to access members.

3.4 The Object Life-Cycle

We'll explore the details of the `Object` class by trying out some examples. The class is essentially a wrapper around a single integer value, but has a few other things so that it can report on what it is doing. If you look at the end of the header file, you'll see a declaration inside the class

```
static int objectCount_;
```

The keyword `static` in this case means that there is only one `objectCount_` for *all* `Object` instances – the count is associated with the class as a whole, not with each instance. The purpose of this is to give `Objects` a unique `id` as they are created, allowing us to keep track of them. The trailing `_` is not special in the language, but is a notation I personally use to indicate private or protected members.

Add the following code to the `main` function in the sample code. Before you run the code, write down the method(s) of the `Object` class that you expect to be called and the order in which they will be called. This might include the constructor(s), destructor, and/or assignment operator.

```
Object obj1;
Object obj2(5);
Object obj3(obj1);
Object obj4 = obj2;
Object ref = obj3;
Object* ptr = new Object();

obj3 = obj2;
```

When you run the code, you should see that six objects are created, but only five are destroyed. This is because the pointer is not freed. You could fix this by adding `delete(ptr)` before returning from the program, but it would be easier (and safer) to use a smart pointer. You may also find it surprising that the copy constructor is called when creating `obj4`. If the copy constructor and assignment operator are defined with the usual copying semantics, then this is equivalent but more efficient as it avoids creating an object that is going to have its data immediately overwritten. If you define copy constructors and assignment operators that mean this is not equivalent, you're asking for trouble and the compiler will cheerfully oblige.

Replace `ptr` with a smart pointer (as discussed in Section 2.1). Check that the object's destructor is being called.

Most C++ programmers follow an idiom called *Resource Acquisition is Initialisation* or RAII. This essentially means that all resources (memory, system resources, etc.) are allocated in a constructor, and freed in the corresponding destructor. This means that well-written C++ can avoid most (if not all) of the errors that can occur when these resources are not freed (or are freed more than once).

3.5 Inheritance and Polymorphism

As with most object-oriented languages, C++ provides direct support for inheritance and polymorphism. The sample code provides an example of this in the `Derived` class, which is a subclass of `Object`. Looking in `Derived.h` we see that this relationship is declared as:

```
class Derived: public Object {
  ...
;}
```

The access specifier (`public` in this case) specifies the level of access that `Derived` instances provide to `Objects` members. For the typical 'is-a' inheritance pattern, this should be `public`. Other use-cases, such as 'is-implemented-using' relationships might use `private` or very rarely `protected` inheritance. Often, however, non-`public` inheritance is a sign of poor design, as most instances where it would be used are better implemented using object composition instead.

As in Java, inheritance means that instance of a derived class can (usually) be treated as if they were instances of the base class.

Lets start by changing our `main` function to create and use some `Derived` instances. Again, you should think about what method(s) of `Object` and `Derived` will be called in what order before running the code.

```
Derived d1;
Derived d2(d1);

d2.setValue(5);
d1 = d2;

std::cout << "Derived d1 has value: " << d1.getValue() << std::endl;
std::cout << "Derived d2 has value: " << d2.getValue() << std::endl;
```

You should see that whenever we create a `Derived` instance, the constructor of `Object` gets called *before* the `Derived` constructor. You should notice a few unusual things about the methods of `Object` that are (or are not) being called:

- When the *copy constructor* of `Derived` is called, the *default constructor* of `Object` is called.
- When the assignment operator of `Derived` is called the corresponding assignment operator of `Object` is *not* called.

- As a result the `value_` of `d1` is not updated in the assignment.

To fix this we need to explicitly call the appropriate methods of `Object`. For constructors, we add the base class constructor we wish to call to the initialisation list. The default constructor of `Derived` would then become

```
Derived::Derived() : Object() {
  ...
}
```

Fixing the assignment operator is a little trickier, but we basically need to call the `Object`'s assignment operator. To specify that, we need to use the form

```
Object::operator=(d);
```

which we can do inside of `Derived::operator=(...)`. Note that although the overloaded operator is typically used with the form `a = b`, we can always use `a.operator=(b)` if we prefer. We need to do so in this case, to specify that it is `Object`'s assignment operator that we want to call.

Update the `Derived` class to ensure that the appropriate `Object` methods are being called. You should check that the copy constructor and assignment operator in particular are successfully copying the `value_`.

Now lets look at polymorphism a little more closely. If you look at `Object` and `Derived`, you can see that they both have methods called `whatAmI`, which report their specific type. We want this to be different for instances of the different classes.

Change the main function to the code below, and as usual think about what you think the output should be before running it to check.

```
Object obj;
Derived der;
Object& ref = der;
Object* ptr = new Derived();

obj.whatAmI();
der.whatAmI();
ref.whatAmI();
ptr->whatAmI();

delete ptr;
```

You should spot a couple of obvious, and one slightly more subtle issues. The more obvious issue is that the reference and pointer report that they are `Object`s rather than `Derived`s. The more subtle one is that when the pointer to a `Derived` is `delete`d, only the `Object` destructor is called, not the `Derived` destructor.

To fix this, we need to specify in the `Object` definition which methods should be referred to the derived class if we have a pointer or a reference to the base class. We do this by specifying them as `virtual` methods, and *it is important to include the destructor in this.*

Make `Object`s destructor and `whoAmI` method virtual as shown below. Check that this fixes the issues we've identified. You should only need to modify `Object.h`.

```
virtual ~Object();
...
virtual void whatAmI();
```

4 The Standard Template Library

C++ provides a standard library of utility classes which make extensive use of *templates*. Templates allow for *generic programming* where algorithms and data structures can be specified independently of the specific types they use. This is similar to Java's generics (and predates them), but more general and powerful. As with many C++ features this gives you more choices at the cost of greater complexity. This is called the Standard Template Library or STL. The STL provides a number of algorithms, data structures, and other functions, and uses the template mechanism to provide generic implementations that can work (efficiently) with different types of base values.

4.1 Templates

A simple example of templates would be a function that finds the minimum of two numbers. The numbers could be integers, floats, chars, or whatever. Without templates you could write a version for each type, and let the type system figure out which one to call:

```
int minimum(int a, int b) {
  return a < b ? a : b;
}

float minimum(float a, float b) {
  return a < b ? a : b;
}

...
```

However, the same code is repeated many times making maintenance harder and it doesn't extend to new types without new code.

In C++ you can use templates to write

```
template <typename T>
const T& minimum(const T& a; const T& b) {
  return a < b ? a : b;
}
```

where `T` is any type for which the code will compile – basically anything with a less-than operator. If you write `int a = minimum(3, 5);` the compiler will use the template above to generate a function of the form

```
const int& minimum(const int& a, const int& b) {
  return a < b ? a : b;
}
```

then call it with 3 and 5 as the values for `a` and `b`. The use of `const` references means that even large objects can be compared (as long as they have `<` defined on them via operator overloading) without copying a lot of information, and the function guarantees that the objects won't be altered in the process.

4.2 The `std::vector` Class

The STL provides a lot of general-purpose containers including `stack`, `queue`, and `map` (an associative array or dictionary), but perhaps the most commonly used is `vector`, which provides a variable length array, much like Java's `ArrayList`. Here's an example using a `vector` of `int`s:

```
#include <iostream>
#include <vector>

int main() {
  std::vector<int> squares(10); // array of 10 integers

  for (size_t i = 0; i < 10; ++i) {
    squares[i] = i*i;
  }

  squares.push_back(10*10);
  squares.push_back(11*11);

  for (const int& square: squares) {
    std::cout << square << " ";
  }
  std::cout << std::endl;

  return 0;
}
```

Things to note:

- The `vector` is in the `std` namespace, which avoids naming conflicts.

- There is no memory management code – the `vector` does it all.
- The size of the vector is not fixed – you can extend it with `push_back`.
- C++ supports a range `for` loop over containers, where you just get (a reference to) each element in turn rather than looping over indices.
- Unlike Java's `ArrayList`, you can put basic types like `int` into a `vector`. In general, C++ doesn't distinguish between built-in and user-defined types very much.

Another point to note, which isn't immediately obvious above is that `vector` gives you a *choice* between speed and safety. When accessing the elements of a `vector` in the first loop we used the overloaded `[]` operator, giving us C-like syntax. This type of access does not check the bounds of the array and so is very efficient. In fact, one of the design goals of the STL was that it should never make the users of the library suffer additional cost. The raged for structure in the second loop however, is safer since you cannot go out of bounds. A `vector` offers other ways to access its elements – the `.at(i)` method, for example, is equivalent to `[i]` but does check the bounds. This is a good example of C++ offering greater choice to the programmer at the cost of increased complexity.

4.3 The `auto` Keyword

C++ has a fairly strong type system, and many compiler errors (for me at least) arise if you get the types wrong. This can be difficult to get right with templated libraries, as the types can become complicated. For example, C++ also supports iterators over containers, which provides a pointer to elements in the container. We could re-write the second loop in the example above as:

```
for (std::vector<int>::const_iterator iter = squares.begin(); iter != squares.end(); ++iter) {
std::cout << *iter << " ";
}
```

While the pattern is simple (start at the beginning, go to the end, incrementing by 1 each time), the type of the iterator – `std::vector<int>::const_iterator` – is non-trivial to get right. And this is a simple example – things can get quite complex quite quickly if you have nested containers. However, the compiler knows what type is expected (otherwise it wouldn't be able to complain), so C++ lets you just say "use the right type" with the keyword `auto`.

```
for (auto const iter = squares.begin(); iter != squares.end(); ++iter) {
std::cout << *iter << " ";
}
```

Here we have to specify `const` to promise that we won't change the contents of the `vector`, but otherwise `auto` just works.

5 `const` Correctness

You've probably noticed the keyword `const` scattered through the examples. It's worth taking a brief look at this, as it is important in writing good C++ code. Essentially, `const` indicates a guarantee that something won't change. Here are some examples:

- You can use `const` to define fixed values (constants):

  ```
  const pi = 3.14159265359;
  ```

- You can make `const` references which provide read-only access to objects:

  ```
  Object obj;
  const Object& ref = obj; // Can't change obj through ref
  ```

- You can guarantee that a function won't change parameters passed by reference:

  ```
  template <typename T>
  const T& minimum(const T& a, const T& b) {
    // Can't change a or b here
  }
  ```

- You can guarantee that a method won't change an object when it is called:

```
class Object {
  ...
  int getValue() const; // calling this won't change the object
  ...
}
```

Using `const` wisely is called `const`-correctness, and a good approximation to the practice is *use* `const` *whenever possible*. This protects you from unexpected side-effects when calling object's methods or passing object references or pointers to functions. It also allows the compiler to make additional optimisations because it knows what values are guaranteed to remain the same over a block of code.

6 An Example – Shapes

Finally, lets look at a classic OO example which is also a graphics example – Shapes. We want to be able to define and draw different types of Shape like Circles, Squares, and Triangles. To do this we'll define an abstract class `Shape` and subclasses like `Circle` and `Square`. Each `Shape` will have a colour, and each particular type of `Shape` will implement a `Draw` method. We can re-use the `Image` and `Pixel` classes from last tutorial to help with drawing and representing colour.

If you look at the code for `Shape` and `Rectangle` and in `shapeMain.cpp` you'll see many of the techniques we've discussed being used. You'll also see a couple of new things.

Firstly, the `draw` method of `Shape` is declared as

```
virtual void draw(Image& image) = 0;
```

This declares a *pure virtual function* and tells the compiler that no implementation will be provided for the base `Shape` class. It also means that `Shape` becomes an *abstract base class* or ABC, and it is not possible to create a `Shape`, only subclasses of it.

Secondly, we've included `<algorithm>` in the `Rectangle` class. This is part of the STL that provides (wait for it...) generic algorithm implementations. Here we're using `min` and `max`, but it also provides searching and sorting of containers, and a lot of other useful utility functions.

Add a Circle class to the program. A `Circle` should be created by providing x and y co-ordinates for the centre and a radius, r.

You'll need to update the `CMakeLists.txt` to include your new `Circle.h` and `Circle.cpp` files and regenerate your build files. Pseudocode for the drawing method of a circle (ignoring bounds checking) is

```
// centre is at (cx, cy), radius is r, colour is c
for dy = -r to r:
  xrange = sqrt(r*r - dy*dy)
  for dx = -xrange to xrange:
    image(cx+dx,cy+dy) = c
```

6.1 Extension

There are a number of ways that you could extend this example if you have extra time:

- You could add more shapes, such as a `Triangle` defined by three corners.
- More generally, you could make a `Polygon` defined by a list of corners. See the Lecture 2 notes for details on painting polygons.
- **Harder:** Drawing is represented here as a method of a `Shape`. However, it makes more sense in some ways to make that a method of an `Image`. One way to do this for the shapes to provide a method that returns a list (a `vector`, say) of scanlines to paint. Each scanline would be a y value and a starting and finishing end x value. You could then draw shapes with something like this:

```
Image::draw(const Shape& shape) {
  for (auto scanline: Shape.scanlines()) {
    for (int x = scanline.startX; x < scanline.endX; ++x) {
      (*this)(x,scanline.y) = Shape.getColour();
    }
  }
}
```

7 For More Information

As you've probably noticed, C++ is a big language. The key thing to remember, though, is that most of the time you only need a small part of the language, depending on the choices you've made. There are, as usual, a lot of useful resources online, and here are a few:

- `http://en.cppreference.com/w` and `http://www.cplusplus.com/` provide reference information about C++ and the STL.
- Wikipedia has a lot of useful articles on aspects of C++. These generally contain a description, the rationale behind the methods, and code examples.
- `https://isocpp.org/` has a lot of resources about C++ and the standards. This includes a useful FAQ and some notes about getting started.
- As with most things in programming, asking the Internet and looking for Stack Overflow results is often helpful. Make sure you read the comments and second answer on Stack to make sure you're getting the right solution for your problem.
- The canonical book on C++ is Stroustrup's *The C++ Programming Language*, but that is more useful as a reference than a tutorial.