Tute 6: Memory Management

Review: Constructors

What is a copy constructor?

Copy constructors are used to initialize a new object with the data contained in an existing object. For example, if we call:

```
Polygon p1 = Polygon(5);
Polygon p2 = p1;
```

A new Polygon will be created on line 1 which will call the Polygon constructor which takes an int argument while the code on line 2 will create a new Polygon which will call the Polygon constructor which takes an argument of const Polygon&. Note that if no such constructor is declared the compiler will provide one by default.

What is the difference between a deep copy constructor and a shallow copy constructor?

Referring to the above example, without defining a copy constructor, if any pointers exist inside the Polygon class, only the pointers will be copied, rather than the data they point to. This means that if we alter this data in p1, the corresponding data in p2 will also be affected. This is called a shallow copy. We can declare a copy constructor like so:

```
Polygon(const Polygon &p)
{
        for (auto &value : p.values)
                this->values.push_back(value);
}
```

A deep copy is a copy where a new object is created entirely separate from the old object. Above we have defined a deep copy constructor for the Polygon class which copies the values for the data stored in the Polygon object, rather than simply copying a pointer. This means when we call the copy constructor, p1 and p2 will both be separate Polygon objects pointing to separate locations in memory, as well as any pointers and references contained in them. Now when we change data in one Polygon object, it will not affect data in the other.

What is the difference between a copy constructor and a move constructor?

While a copy constructor makes a new instance of the class using the instance passed in, copying all the data, a move constructor creates a new instance by moving the data. A move constructor essentially rips the contents from one instance and changes the ownership to another a new instance.

Have a look at move.cpp in the week 6 examples folder - this defines a move only type with no copy constructor. Discuss with the students what the positives and negatives are here, especially when it comes to speed.

What is a conversion constructor?

A conversion constructor is a constructor that is used to convert a variable from one type to another. Using a Decimal class as an example:

```
Decimal(int i);
```

Would mean that anytime a Decimal object is created using an int, that constructor is called. Which allows us to do things like:

```
Decimal d = 1;
```

What does the explicit keyword mean?

The explicit keyword is used on conversion constructors to prevent cases such as:

```
Decimal d = 1;
```

from happening, and forces the programmer to explicitly call the constructor, like so:

```
explicit Decimal(int i);
Decimal d = Decimal(1);
```

This does not affect the behaviour of the conversion constructor, only how it is called.

Programming Concepts: Memory Management

What is an assignment operator?

Unlike the copy constructor, which is used to initialize a new object with the data contained in an existing object, the assignment operator is used to copy data from one object to another object which has already been created.

In the above example, the copy constructor will be called since the Polygon p2 has not yet been created (so it needs a constructor to be called, the copy constructor in that case). However, if the object has already been constructed, there is no need to call the constructor again, and so the assignment operator will be used instead.

```
Polygon p1 = Polygon(5);
Polygon p2 = p1;
p1 = p2;
```

In the above case, the call to p1 = p2; will call the assignment operator, rather than the copy constructor. An assignment operator can be defined as:

```
Polygon &operator = (const Polygon &p) { … }
```

What are the different types of pointers available?

Note: this is a C++11 feature.

The three main types of smart pointers available in C++ (in addition to traditional C pointers) are:

- std::unique_ptr<T>
- std::shared_ptr<T>
- std::weak_ptr<T>

A unique_ptr maintains sole ownership of the memory it points to. When the unique_ptr goes out of scope, the destructor of the object it is associated with is called, and the memory is freed. A unique_ptr can be created like so:

```
class Object { … };

std::unique_ptr<Object> o = std::unique_ptr<Object>(new Object);
```

Other pointers to this data can be obtained like so:

```
Object *oPtr = o.get();
```

However if the unique_ptr goes out of scope, oPtr will now point to junk memory.

A shared_ptr operates in a similar way to a unique_ptr, except that an object can be owned by multiple shared pointers, and so it will only be deleted when all shared_ptrs go out of scope (instead of just the one unique_ptr). Multiple shared_ptrs can be created like so:

```
std::shared_ptr<Object> o1 = std::shared_ptr<Object>(new Object);
std::shared_ptr<Object> o2 = o1;
```

A weak_ptr holds no ownership over the memory it points to. If it points to an object owned by a unique or shared pointer, then it will be valid, but once the unique/shared pointer deletes the object, the weak_ptr will point to null.

What is constexpr?

Note: this is a C++11 feature.

A method or function marked as constexpr is evaluated at compile time rather than runtime, eg:

```
constexpr void APlusB(int a, int b) { return a + b; }
int x = APlusB(5, 10);
```

x would be given the value 15 at compile time, which means during runtime, the function will never actually be called. A function/method can only be used in this way if the arguments to the function are assigned at compile time (ie, variables assigned at runtime that are passed to the function will result in a compile error).

Compilation: Object Files

What is an object file?

An object file is the compiled code that is produced by a compiler, the file extension for C++ object files produced by g++ is ".o". Object files cannot themselves be executed, however they contain executable code intended for their target hardware and operating system. Object files can be compiled using the compiler flag -c. Typically the g++ compiler will produce one object file for each .cpp file being compiled.

An object file consists of several parts, the three main ones are:
- Machine code: fairly obvious.
- Symbol table: names, addresses and linkage types of all identifiers.

- Relocation table: list of addresses of all placeholders in the machine code, along with their named identifiers.

What is a linker?

A linker is a program separate from the compiler (although it is invoked by the compiler once compilation is complete) that links object files and libraries together to produce an executable file. The linker can be invoked on its own without the use of g++ with the command "ld" (ld is the linker executable used by gcc).

A linker will take the given input object files and read their symbol tables, substituting all placeholder identifiers with their addresses (ie, a function may be implemented in one object file, and used in another). The linker will then merge the machine code of all object files into a single executable and translate all addresses into one common address space.

Errors: Linker

What is a linker error?

These are errors produced by the linker when it tries to link together object files to create an executable. When a linker error occurs, typically an error message saying "undefined reference to function x first referenced in file y".

What causes a linker error and how do you fix it?

A linker error is usually caused when the linker cannot find the location of a function in any of the symbol tables that is required by the relocation table of an object file (ie, a function has been declared somewhere, but not implemented, or you have forgotten to compile a .cpp file or provide a .o file that is required by another .o file.

Debugging: None this week.

Exercises:

Look at memory.cpp from the week 6 examples folder as a reference.

What output will the following code produce and why?

```
class A
{
public:
        A(int i) : i(i) { printf("Constructing<%d>...\n", i); }
        ~A()            { printf("Destructing<%d>...\n", i); }
        int i;
};

A a = 1;
int main(int argc, char **argv)
{
        printf("======\n");
        a = 3;
        printf("======\n");
        a = 5;
        printf("======\n");

        return EXIT_SUCCESS;
}
```

The output will be:

```
Constructing<1>...
======
Constructing<3>...
Destructing<3>...
======
Constructing<5>...
Destructing<5>...
======
Destructing<5>...
```

In the first call (A a = 1) the constructor A(int i) is called, which has been defined, and assigns the A object to a. This object exists in global memory.

In the next call (a = 3) another A object is created by calling the A constructor, and is assigned to a, but since a has already been constructed, rather than using a copy constructor to do this, the assignment operator will be used instead. This means that the original temporary A object that was created when calling A(3) needs to be deleted, which is why there is a call to the destructor.

The next call to (a = 5) will operate much the same way as the previous.

There is an extra destructor call at the end of program execution, because even though the temporary objects that were created have been deleted, the initial A object has not. This means the destructor will be called, but it will use the value 5 instead because the assignment operator has been used to copy an A object with the value 5 over to a.

What would happen if a copy constructor was defined for class A?

In the above program, since we are never creating any new A objects, a copy constructor would make no difference here, as it is really the assignment operator that is being called, which in this case, is the default assignment operator.

What would happen if an assignment operator was defined for class A?

Providing an assignment operator will make no difference to the program, however if an assignment operator were provided with no implementation the program would fail to compile due to it needing an assignment operator to function.

What would happen if a was declared and used as a pointer instead?

Using pointers would mean that we would need to use the keyword new with each constructor call, meaning that only the constructor would be called and not the destructor, which would result in memory leaks. In order to avoid this a would need to be deleted before each A constructor call.