Tying up some loose ends

Class Destructors

A destructor is the opposite of a constructor. A constructor is called automatically when a class is instantiated. A destructor is called automatically when a class goes out of scope.
It is often used to clean up any memory, freeing any memory on the heap. This is a good way of avoiding memory leaks.

A destructor has the name of the class, like a constructor, preceded by the ~ (tilde character).
It cannot have any parameters or return type.
Like constructors a default destructor is provided if you don't write your own.

See the destructor.cpp example on line

Copy Constructors

Sometimes an object is copied automatically for you. This occurs when an object is passed by value, returned from a function, initialised to another object through an initialiser, provided as a single argument to the objects constructor.

Copying is done by a special member function called the copy constructor. Like constructors and destructors a default copy constructor is provided if you don't write your own.
The default copy constructor simply copies the value of each data member to data members of the same name in the new object – a member-wise copy.

Why code our copy constructor?

If a copy constructor is not defined in a class, the compiler itself defines one. This will ensure a shallow copy. If the class does not have pointer variables with dynamically allocated memory, then one need not worry about defining a copy constructor. It can be left to the compiler's discretion. If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor.
The default copy constructor works in the following way. If we have a data member that points to a string on the heap the copying would result in a new object that points to the same string. Remember pointers store memory addresses so the memory address from the original member data gets copied into the new instance.
This can cause real problems if one class goes out of scope and the memory gets freed, leaving us with one class that points to some free-memory, essentially creating a dangling pointer.

Coding our copy constructor
To avoid the problems above what we really need is a copy constructor that produces an entirely new object, including its own memory on the heap. Essentially we need a deep copy.

The code is almost like that of an actual constructor, no return type, same name as the class. The only difference is that it accepts a reference to an instance of the same class. The reference is almost always made constant to protect the original object from being changed during the process.

```
class B    //With copy constructor
{

private:
           char *name;

public:

           B()
           {
               name = new char[20];
           }

           ~B()
           {
               delete name[];
           }

           //Copy constructor
           B(const B &b)
           {
               name = new char[20];
               strcpy(name, b.name);
           }
;
```

See the copyconstructor.cpp example on line

Calling base class member functions
You can call a base class member function, whether it be an overloaded operator or a copy constructor, from a derived class. All you need to do is prefix the class name to the member function name with the scope resolution operator.

```
void Regenerate() const
{
```

```
    Enemy::Regenerate();
    m_shield += 50;
}
```

You can extend the way a member function of a base class works in a derived class by overriding the base class method and then explicitly calling the base class member function from this new definition in the derived member class and adding some functionality.

See the classcall.cpp and classcall.h example on line