FIT2096 - Games Programming 1

Week 1 Supplementary – C++ Refresher

This brief supplementary is designed to jump-start your C++ brains back into action. Think of this document as a curated repository of useful guides and articles from all over the net, with a particular emphasis on the concepts you'll need for this unit.

As a bit of an into to the unit, there'll also be some brief discussion on how these concepts apply in game development, with a specific example from this unit. Specifically, we'll cover:

- Pointers
- Enums
- Function overloading
- Function overriding (and running the base version first)
- Virtual functions
- Abstract classes and pure virtual functions
- Calling parent constructor from the child class

Remember you don't need to recall exactly how all of these work today – the last thing we want to do is scare you away from the unit. If something is still a bit hazy, you can revisit it when the time comes to actually use it. We'll also review some of these again later in the semester.

Pointers

Our old friend the pointer re-emerges. A lot of people are scared of pointers, but they're really not bad once you've used them for a while. A pointer is simply a memory address – that's it! We call it a pointer as it's a way of keeping track of where something is in memory – it points to it. The power of pointers is that we can change what they're pointing to. This is a very powerful concept which we'll hopefully start to appreciate by the end of the unit.

If there were any required reading for this unit, it would be the below chapter on Pointers and References from the Foundations of C++ content repository. It's highly recommended you flick through this chapter if you're hazy on pointers and why we need them:

https://www.alexandriarepository.org/syllabus/fit2071-foundations-of-cpp/41898

Most of the objects we create in this unit will be created on the heap using the new keyword, and we will use pointers as our way of talking to these objects.

Often an object will need to be accessible from many parts of our engine. It will be very common for us to instantiate an object once, then pass its memory address into other objects which will need to talk to it.

```
int score = 0;
int lives = 3;

// Store a memory address containing an int
int* pointerToAnInt = &score;

// Sets the value contained in the memory address
*pointerToAnInt = 10;

std::cout << "Highscore: " << *pointerToAnInt << std::endl;
// Outputs "Highscore: 10"

std::cout << "Highscore: " << score << std::endl;
// Outputs "Highscore: 10"

// Both the pointer and the original are the same thing!

// Make the pointer point to a different memory address
pointerToAnInt = &lives;

std::cout << "Lives: " << *pointerToAnInt << std::endl;
// Outputs "Lives: 3"

*pointerToAnInt = 1;

std::cout << "Lives: " << lives << std::endl;
// Outputs "Lives: 1"

lives = 5;

std::cout << "Lives: " << *pointerToAnInt << std::endl;
// Outputs "Lives: 5"
```

Enumerated Types

Enumerated Types (or enums for short) are really cool. If you've ever used ints or chars to represent something like direction or colour, you'll know that after a while it gets hard to keep track of what's what. Enums let us define a new type that can only be assigned to a set of values we specify.

"The idea behind enumerated types is to create new data types that can take on only a restricted range of values. Moreover, these values are all expressed as constants rather than magic numbers - in fact, there should be no need to know the underlying values. The names of the constants should be sufficient for the purposes of comparing values."

http://www.cprogramming.com/tutorial/enum.html

"An enumeration provides context to describe a range of values which are represented as named constants and are also called enumerators."

https://msdn.microsoft.com/en-us/library/2dzy4k6e.aspx

Note we now have C and C++ style enums. See below:

http://stackoverflow.com/questions/18335861/why-is-enum-class-preferred-over-plain-enum

```
// Define some directions to make describing movement easier
enum class MovementDirection
{
   UP,
   DOWN,
   LEFT,
   RIGHT
};

enum class RotationDirection
{
   CLOCKWISE,
   COUNTER_CLOCKWISE,
};

void Ship::Move(MovementDirection direction)
{
   if (direction == MovementDirection::RIGHT)
      m_targetPosition.x += 1.0f;
   // etc.
}

void Ship::Rotate(RotationDirection direction)
{
   if(direction == RotationDirection::CLOCKWISE)
   {
      m_targetRotation += 90;

      // Wrap rotations so they stay within 0-360 range
      if (m_targetRotation == 360)
         m_targetRotation = 0;
   }
```

Function Overloading

We can have multiple functions with the same name and return type, as long as they accept different parameters.

> "Overloading functions is where you have two or more functions with the same name, that have different implementations. Why name them the same? Since a function is typically designed to a specific job, we might have a case where the job is almost the same, but it just needs some different data to do it."
>
> https://www.alexandriarepository.org/syllabus/fit2071-foundations-of-cpp/50927

> "You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You can not overload function declarations that differ only by return type."
>
> http://www.tutorialspoint.com/cplusplus/cpp_overloading.htm

Don't get this confused with function overriding – that's the one related to inheritance. Speaking of which…

```
// Two versions of the same function.

void Direct3D::BeginScene(float red, float green, float blue, float
alpha)
{
    float colour[4];
    colour[0] = red;
    colour[1] = blue;
    colour[2] = green;
    colour[3] = alpha;

    // No point duplicating the heart of the function, so flow through
    // to the original version now we've done some data formatting.
    BeginScene(colour);
}

void Direct3D::BeginScene(float colour[])
{
    // Clear the scene to the passed in background colour
    // Render the scene
}
```

Function Overriding

A child class can re-implement a function from the parent class and provide more specific logic. Generally, the further down the inheritance line we get, the more information we have, hence needing to re-implement a function from our parent is very common.

https://www.alexandriarepository.org/syllabus/fit2071-foundations-of-cpp/41948

One cool thing you can do with this is run both versions of the function by calling the overridden parent version from the child. This allows the parent to do some initial setup based on the data it has, and then our child can come along and perform more specific logic. The example on the right does this.

Any weird code you see mentioning rendering and cameras is not important for today. It's presented merely as a preview of what's coming up this semester.

```
/* This example is from a Direct3D implementation of BattleShip.
   In this case, a GameBoard represents the board each player has
   and manages the ships and hit markers placed on that board.
   GameBoard inherits from GameObject, which is the base class
   for all objects we want to render in our game (more on this
   later in the semester).
*/

void GameBoard::Render(Direct3D* renderer, bool drawShips)
{
   // Run parent's render first to draw board mesh
   GameObject::Render(renderer, m_camera);

   // A board also needs more specific render logic
   // A board manages the ships and markers placed on it
   // so a board should be responsible for drawing them
   if (drawShips)
   {
      for(unsigned int i = 0; i < m_ships.size(); i++)
      {
         m_ships[i]->Render(renderer, m_camera);
      }
   }

   for (unsigned int i = 0; i < m_markers.size(); i++)
   {
      m_markers[i]->Render(renderer, m_camera);
   }
}
```

Virtual functions

Marking a function as virtual ensures the most specific version of that function runs when we call it using a pointer to the parent class.

The first answer to this StackOverflow question gives a very nice example of why virtual functions are useful.

http://stackoverflow.com/questions/2391679/why-do-we-need-virtual-functions-in-c

"When using polymorphism, the Parent class functions will always have priority over the child classes, unless we use Virtual Functions."

https://www.alexandriarepository.org/syllabus/fit2071-foundations-of-cpp/41966

"Non-virtual members can also be redefined in derived classes, but non-virtual members of derived classes cannot be accessed through a reference of the base class"

http://www.cplusplus.com/doc/tutorial/polymorphism/

We'll often mark a function we know we're going to override at some point as virtual. Destructors will also be marked virtual to ensure child classes can clean up after themselves when delete is called on a pointer to the parent.

```
// You know... that StackOverflow example in the first box is
// so good I can't beat it. Seriously, go check it out as this example
 // builds on it.

// One addition though. Instead of using the intermediate function,
// this example is more akin to what we'll do in this unit.

// We have a collection of pointers to the base class, meaning
// this can store animals and cats. That's polymorphism in action.
std::vector<Animal*> m_animals;

Animal* animal = new Animal();
Cat* cat = new Cat();

m_animals.push_back(animal);
m_animals.push_back(cat);

for(int i = 0; i < m_animals.size(); i++)
{
   // Now we're calling eat on what the compiler thinks is
   // an animal, so the base class version of eat() takes
   // priority. But, if eat is marked virtual in the base class,
   // then something called late binding occurs meaning the
   // version of eat() to run is determined at run-time based
   // on what this animal really is
   std::cout << m_animals[i]->eat() << std::end;
}

// if eat() is marked virtual
// Outputs "I'm eating generic food."
// Outputs "I'm eating a rat."

// if eat() is not virtual
// Outputs "I'm eating generic food."
// Outputs "I'm eating generic food."
```

Abstract Classes and Pure Virtual Functions

Sometimes a class simply doesn't know enough to exist – it may exist only to make inheritance more robust. A class which can never be instantiated is called an abstract class.

To mark a class as abstract, the class must contain one or more pure virtual functions. A function can be marked as pure virtual using the virtual keyword and suffixing the declaration with = 0. What we're saying when we do this is "oh by the way, I want my children to provide an implementation for this function, because I don't know enough to provide one". If a child doesn't provide a body for a pure virtual function, the child also becomes an abstract class.

> "We implement these when it makes sense to create a class so that we can numerous create inheriting classes from it, but we want to make sure that we can never have an instance of that class."
>
> https://www.alexandriarepository.org/syllabus/fit2071-foundations-of-cpp/41972

We'll be using abstract classes occasionally in this unit not because we need to, but because we can (we're learning after all, so why not).

```
/* Another example from Battleship. In this case, we have a
   human player and a computer player, who both launch attacks
   differently (humans listen to keyboard input, computers use
   basic AI). This abstract Player class contains things both
   types of players will need, however it doesn't know enough
   to exist on its own hence it is abstract.
*/

#ifndef PLAYER_H
#define PLAYER_H
#include "GameBoard.h"

class Player
{
protected:
   // Which gameboard does this player use
   GameBoard* m_myGameBoard;

   // Players remember previous shots so we don't shoot the
   // same cell twice
   bool m_hasFiredHere[10][10];

public:
   Player(GameBoard* myGameBoard);

   // Pure virtual. We don't know enough to launch an attack
   virtual Vector2 Attack() = 0;
};

#endif
```

Reusing Parent Constructor from Child Class

The constructors of child classes will often need to accept the data required by their parent (allowing us to pass a variables initial value into the child constructor). Instead of passing this up to our parent by accessing its protected variables, we can call the parent's constructor from the child.

https://www.alexandriarepository.org/syllabus/fit2071-foundations-of-cpp/41948

```
/* Building on the last example, here is the constructor for a
   HumanPlayer which inherits from Player. It accepts something
   specific to it, in this case the InputController so it can
   talk to the keyboard. It also accepts the game board this
   player controls which is something our parent class
   wants to know about (this is common to all types of player)
*/

HumanPlayer::HumanPlayer(InputController* input, GameBoard*
myGameBoard) : Player(myGameBoard)
{
   // Did you see where we're calling the parent constructor?

   m_input = input;

   // Don't have to assign gameboard here as parent handles it
}
```