Classes and an Introduction to Multiple Code Files

Data Type: A collection of values and the definition of one or more operations that can be performed on those values.

C++ includes a variety of built-in data types:

```
short, int, long, double, float, char, etc...
```

C++ supports several methods for aggregate and complex data types.
Aggregate data types:
Arrays, Structures, Classes

Complex Data Types:
Structure, Union, Class

C++ also supports other mechanisms that allow programmers to define their own custom data types:
`enum` types and `typedef`

Differences between define and declare:
Defining an object is not the same as Declaring an object and these words cannot be used interchangeably. The difference is that when we *define* something, we are setting up a new data type, such as an *enum*, a *typedef*, or some other type of user-defined data and when we *declare* something, we are setting up a new variable to be of a type we have already *defined* or that has already been *defined* for us as a standard data type in C++.

Types of Programming

During the course you have heard me mention object Oriented programming, but what exactly does that mean?
Well the types of programming paradigms that are defined are many:
Functional, Imperative (or Procedural), Logical, Object-Oriented.
Most programming languages use a few of these paradigms.

Paradigm:
1: EXAMPLE, PATTERN; especially: an outstandingly clear or typical example or archetype
2: an example of a conjugation or declension showing a word in all its inflectional forms
3: a philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them are formulated; broadly: a philosophical or theoretical framework of any kind

I am not going to cover all these paradigms in detail; this is not a theoretical computer science course.

So far we have been programming using an Imperative paradigm. In computer science, imperative programming, as opposed to declarative programming, is a programming paradigm that describes computation in

terms of a program state and statements that change the program state. In much the same way as the imperative mood in natural languages expresses commands to take action; imperative programs are a sequence of commands for the computer to perform.

Through classes we follow a more Object-Oriented paradigm.

These paradigms are essentially just different ways of thinking about programming. The different thoughts though need the languages to support the expression of those thoughts. C++ supports Imperative and Object-Oriented paradigms.

Object Oriented Programming

Object-Oriented programming is a different way of thinking about programming and one that is used in the vast majority of game creation, as well as lots of other commercial software.
In OOP you define different types of objects with relationships to each other that allow the objects to interact.
We have already worked with objects from types defined in libraries – namely the string data type. One of the key characteristics of OOP is the ability to make your own data type from which you can create objects. We are going to be looking at defining our own data types and creating objects from them.

Grouping data: Classes

Organising Heterogeneous Data

Arrays allow programmers to lists of values that are all of the same type (homogeneous).
We are often faced with values of differing types (heterogeneous) that are logically related:

Player	Score	Kills	Accuracy
Jeff	12	6	50.7%
Bob	1	1	2.5%
Tamara	23	20	80%

Classes provide us with a mechanism for organising these logically related values.

What are Classes?

Whether we talk about cars, monsters, spaceships or players games are full of objects. C++ helps use represent these game entities as software objects, complete with member data and member functions. These software objects work just like the `string` objects we have already seen. To use these

objects, DeathMatchPlayers, cars, spaceships, etc, in games we must first define them.

Definition: A fundamental construct in C++. The class groups its data, the class members, with functions that act on them, the methods of the class. Classes are a collection of data values grouped together under a declaration.

The data values are called members or fields
Each member has a type and a unique name
Individual members are accessed by name

Classes are used all the time in games; virtually everything in UT2K3 was a class. Example is player – each player is generally an instance of a class.

If we remember from last week and our talk about objects – the term instance and object can be used interchangeably. This makes the string data type a class and variables of that string class, objects or instances.

It uses the keyword class.
They provide a convenient way of keeping information together.
You may have used other languages where the keyword record is used; these are very similar to classes in terms of basic data organisation – they can both be used to store heterogeneous data.

Using classes we can also form complex hierarchies via inheritance. Classes may be copied to and assigned. They are also useful in passing groups of logically related data into functions.

They also help conceptually with programming especially in the realms of games.

Classes have components, naturally as it were, that are attributed to OOP programming:

Encapsulation: means that data and the operations on that data are combined into a single well-defined programming unit.
Information Hiding: means that operations and functions outside of an encapsulated unit cannot access, alter, or corrupt private data inside that unit.
Encapsulation and Information are sometimes linked together.

Classes facilitate abstraction. Abstraction is the process of design which allows us to ignore details

Classes also enable the Inheritance to be carried out easily and successfully. Inheritance is in many ways an extension of the use of encapsulation and modularity, although it is concerned with abstraction. Generally, inheritance is a way of ranking or ordering abstractions into hierarchies.

Do not worry at this stage if you are thinking what does all this OOP speak mean, we will cover it in detail as we go along.

Defining New Types: Classes

From a class, after defining it, we create individual objects that have their own copies and of each data member and access to all of the member functions. A class therefore is like a blueprint. Just as a blueprint defines the structure of a building a class defines the structure of an object. Just as builders can make many houses from the same blueprint a game programmer can create many objects from the same class.

Syntax: A class is declared by using the keyword class followed by an optional structure (**model_name**) tag followed by the body of the structure. The variables or members of the structure are declared within the body.

Classes are generally declared in the following way:

```
class model_name
{
  type1 member1;
  type2 member2;
  type3 member3;
  .
  .
} object_name;
```

model_name is the name given to the class, by you as the programmer. **object_name** is the name given by you as the programmer as an identifier for instances of the class.

The types contained within the curly brackets, members, are any valid C++ data type – Arrays, variables etc
They are sub-identifiers in the class.

Ways of declaring:

Ok, we've seen the general syntax of the class above, let's look at some examples.

```
class CPlayer
{
     public:
       int Display_Stats();

     private:
        int m_iHealth;
        int m_iScore;
};
```

The definition of a structured type specifies the name of the class:
The type name – in this case ‘CPlayer’
It also specifies the types and names of the members.

When we define data types, like this, the declaration is almost always global as they are generally required throughout the program.

The definition above does only that, it defines a new structured type. From there we can declare variables of that type.

In classes all data is private by default. This means that generally the only “in scope” functions are functions declared as public that then provide access and the capability to modify the class data.
As a general rule, the attributes or member data of the class is normally defined as private. The methods or member functions of the class are used to access them. This ensures the integrity of the class and is one of the major strengths of Object-Oriented Programming, meaning there are less data integrity problems with classes than with structures.

**Note*: Although all members of a class a private by default it is considered good practice to explicitly define which members are private and which members are public.

**Note: Please note that unlike functions that don’t need a semicolon after declaration structures do.*

**Note: It is important for us to differentiate between the parts of a class - what is a class model, and what is a class object. C++ uses similar terms for all declarations and definitions, so using the terms we used with variables, the model is the type, and the object is the variable. We can instantiate many objects (variables) from a single model (type).*
In most examples the declaration of the structure is the model or type and any declarations of that type e.g. Player. Jeff, Bob; are objects of that type.

Building a class

We will look at building a class now:

Defining a Class

We begin class definition with the keyword class and then the name of our class. The name should, by convention, begin with an uppercase letter or the letter ‘c’ or ‘C’.
We surround the class body with a curly braces and end it with a semicolon.

```
class Player
{
};
```

Declaring Member Data
Let's declare some member data.
Class data members represent qualities of the object. We give the player two qualities, both of which could be represented by an integer so are declared of type `int`.

```
class Player
{
     public:
        int m_iHealth;
        int m_iArmour;
};
```

Every instance or object of our data type, Player, will have its own health and armour represented by the data members named `int m_iHealth, int m_iArmour.`
Note that the member data has been pre-fixed with `m_.` Many game programmers follow this naming convention so that member data is instantly recognisable.

Declaring Member Functions
So far our class doesn't do anything so let's declare some member functions; we will start by declaring the headers (or prototypes) for the functions.
These functions declarations appear in the body of the class and represent the object abilities.

```
class Player
{
     public:
       int m_iHealth;
       int m_iArmour;

       void Add_Health(int H);
       void Display_Stats();

};
```

Defining Member Functions
You can define member functions outside of a class definition. Outside of the Player class definition, I define the Player member functions `Add_Health(int H)` and `Display_Stats()`.

```
class Player
{
     public:
       int m_iHealth;
       int m_iArmour;

       void Add_Health(int H);
       void Display_Stats();

};

void Player::Add_Health(int H)
```

```
{
      m_iHealth += H;
}

void Player::Display_Stats()
{
      std::cout << “Player’s Health is: “ << m_iHealth << “\n“
            << “Player’s Armour is: “ << m_iArmour << “\n“;
}
```

The function definition looks like any other function definition you have seen, except for one thing. The function name is prefixed with `Player::`. When you define a member function outside of its class you need to qualify it with the class name and scope resolution operator so the compiler knows that the function definition belongs to the class.

In both functions I use the classes member data, either altering it in the case of `Add_Health(int H)` or simply printing it to the screen in the case of `Display_Stats()`. You can access member data and member functions of an object in any member function simply by using the members name.

Instantiating Objects
When you create an object, you instantiate it from a class. That’s why specific objects are called instances of the class.
We create instances of a class just like creating variables of any other basic data type:

```
Player    Jeff;
Player    Hanna;
```

Please note that these should occur in `main()`, just like any other variable declaration.

```
int main()
{
     Player    Jeff;
     Player    Hanna;

     return 0;
}
```

In main we now have two Player objects – Jeff and Hanna.

Accessing Member Data
The individual members of a class can be accessed using a dot (.), the member access operator. We saw this operator when looking at strings last week.

```
Player    Jeff;
```

This accessing allows, via the (.) operator, us to store or retrieve data.

To access a member of the instance `Jeff`, e.g. `m_iHealth`, `m_iArmour`, we use the instance name followed by the dot (.) operator and then the member data name, in the following way:

```
Jeff.m_iHealth;
Jeff.m_iArmour;
```

The members of a class - now we know how to access them can have values assigned to them in the usual way.

```
Jeff.m_iHealth = 100;
Jeff.m_iArmour = 50;
```

We can even retrieve the information there:

```
// print out the players Health
cout << Jeff.m_iHealth;

int nHealth = Jeff.m_iHealth;
```

This example shows how we can use the elements of the structure as normal variables or types.

It also possible to perform assignments using cin for example to read user data in and then store it

```
// get playerName
cin >> Jeff.m_iHealth;
```

Calling Member Functions
To call functions we again use the member access operator, dot (.).

To access a member function of the instance `Jeff`, e.g. `Add_Health(int H)`, `Display_stats()`, we use the instance name followed by the dot (.) operator and then the member data name, in the following way:

```
Jeff.Add_Health(int H);
Jeff.Display_stats();
```

Constructors

When you instantiate objects you often want to do some initialisation – usually assigning some values to data members.

This is possible through the classes special member function known as a constructor. This member function is called every time a new object is instantiated. We can therefore use the constructor to perform the initialisation of the new objects member data.

A constructor is special member function that is called when an object of the class is instantiated. This means that we do not call it, it is called automatically when we instantiate an object.

Declaring a Constructor
A constructor is a class method that has the same name as the class, but has no return value. We are therefore limited in how it looks.

```
class Player
{
     public:
       int m_iHealth;
       int m_iArmour;

       void Player();
       void Add_Health(int H);
       void Display_Stats();
};
```

The constructor prototype looks like this:

```
void Player();    // constructor prototype
```

As you can see the constructor has no return type and its name matches that of the class.

This is just the constructor declaration. We still need to define the constructor, as we did with the other member functions.
So below our class we can write the following:

```
Player::Player()
{
      m_iHealth = 100;
      m_iArmour = 0;

      std::cout << "Health = " << m_iHealth
            << "\nArmour = " << m_iArmour << std::endl;
}
```

If you declare no constructors the compiler will create a default one automatically. This default constructor is empty and doesn't have any effect on member data. It is considered good practice to define your own, and if you want to initialize member data you won't have a choice.

Member Access Levels

At the beginning of the lecture it was indicated that classes enable encapsulation and that this is part of the OOP paradigm. This section is going to look at using the keywords `private` and `public` to make the class an encapsulated.

What this means in real terms is that we should avoid directly altering or accessing an objects data members. This includes defining the class so that others who use it can not directly alter member data. To alter an objects data we should call an objects member function(s). This allows the object to maintain its own data members and ensure their integrity.
Fortunately for us we can enforce data member restrictions when we define a class. We set the member access levels.

```
class Player
{
     public:
       void Player();
       void Add_Health(int H);
       void Display_Stats();

     private:
       int m_iHealth;
       int m_iArmour;
};
```

In our code above we have set the member access level. The member data is private and the member functions public. What this means is that we can no longer access the data members. The only way we can alter their value is by using the provided functions.

```
Player Noob;
Noob.m_iHealth = 250;     // error!!
```

The member data is now private, this means that we cannot access externally from the class. Members of the class, such as the member functions can still access it.

Members declared after the private keyword are only visible to the other members of the class. Members following the public keyword are visible to any creator of the class. If the public or private keyword does not appear, the default visibility determines it. The default visibility for classes is private.

So we set member access levels using `private` and `public` keywords. Members that proceed the public keyword are public, this means they can be accessed by any part of the program.

```
Noob.m_iHealth = 250;     // error!!
Noob.Display_Stats();  // ok, Display_Stats() is public
```

Members that proceed the private keyword are private. This means that only code it its associated class, Player in the example, can directly access it.

In the example I have made all member data private and all member functions public. Although member data should generally be private, you can also make member functions private as well – they just need to proceed the private declaration.

You can also repeat access modifiers. If you wanted you could have a private section and public section and then another private section.

Accessor Member Functions

An accessor member function is one that gives indirect access to a data member.

```
class Player
{
     public:
       void Player();
       void Add_Health(int H);
       void Set_Health(int H);
       int  Get_Health();

     private:
       int m_iHealth;
};

void Player::Set_Health(int H)
{
      if(H < 0)
      {
            cout << “Health must be a + number\n“;
      }
      else
      {
            m_iHealth = H;
      }
}

int Player::Get_Health()
{
      return m_iHealth;
}
```

The functions `Add_Health(int H), Set_Health(int H), Get_Health()` are examples of accessor member functions. Through these functions we are indirectly accessing member data.

If you are wondering why we don’t just make the member data public. The answer is that we want to define how the member data can be accessed, therefore protecting data integrity and limiting access to ways that we think are appropriate to the class. Access to member data through functions allows us to limit access and enables us to carry out a variety of checks to make sure what they are providing is valid.

As a note, a lot of game programmers begin there accessor member functions name with `Get` or `Set`. Yet another example of programming practice and naming conventions that has arisen.

Constant Member Functions

A constant member function is one that can't modify a data member or call a non-constant member function of its class.
This restricting of what a member function can do is part of the principle of asking for what you need. If you don't need to change any data members in a member function then it's a good idea to declare that member function to be constant. It protects you from accidentally altering a data member in the function and it makes your intentions clear to others who are reading your code.

You declare a constant member function by putting the keyword `const` at the end of the function header in the class declaration.

```
class Player
{
     public:
       void Player();
       void Add_Health(int H);
       void Set_Health(int H);
       int  Get_Health() const;

     private:
       int m_iHealth;
};

int Player::GetHealth() const
{
      return m_iHealth;
}
```

In the above class the Get_Health() function is declared to be a constant member function.
This means that Get_Health() can't alter the value of a data member nor can it call any non-constant member function. Get_Health() was made constant because it only returns a value and doesn't need to modify any data member. Generally `Get` member functions return data values and don't need to alter member data therefore can be defined as constant.

Multiple Code Files

"While many simple programs fit into a single C or CPP source file, any serious project is going to need splitting up into several source files in order to be manageable. However, many beginning programmers may not realize what the point of this is - especially since some may have tried it themselves and run into so many problems that they decided it wasn't worth the effort." (Ben Sizer)

The splitting of code in to different files and the use of header files are introduced in this lecture. Modularising the code in this way is principally a code design issue, one that has very big repercussions particularly in

applications such as games where the code base is huge. It becomes necessary to master such division for your code to be excellent.

A Warning

There are many different compilers out there and they all handle the methods of compiling and running multiple code files differently. To see how your compiler handles this becomes a matter of checking the documentation.

Why Divide Code into Multiple files?

Some people, especially new programmers, ask this question especially when you directories full of separate code files. It can often seem daunting and confusing looking at large directories with many code file.
The division of any reasonably sized project into code files gives some big advantages; the advantages listed are often considered an aspect of modularity. Modularity being another OOP paradigm that deals with breaking up the code into logically related “chunks”, often able to work independently of the program they are apart of.

Speed up compilation

Most compilers work on a file at a time and if you remember building any program occurs in two phases, the compilation phase and the linking phase. Linking object code files into an executable file is fairly quick. But compiling source code into object code can be fairly involved. Compiling a large source code project from scratch can take several hours (many games companies perform nightly builds on their projects) while linking the same project can take just a couple of minutes.
So if you have a project that contains 10000 lines of code and it is in one file, if you change one line, then you have to recompile 10000 lines of code. On the other hand, if your 10000 lines of code are spread evenly across 10 files, then changing one line will only require 1000 lines of code to be recompiled. The 9000 lines in the other 9 files will not need recompiling. (Linking time is unaffected.)

Increase organization

Splitting your code along logical lines will make it easier for you (and any other programmers on the project) to find functions, variables, structure or class declarations, and so on.
Even with the ability to jump directly to a given identifier that is provided in many editors and development environments (such as Microsoft Visual C++), there will always be times when you need to scan the code manually to look for something.
Just as splitting the code up reduces the amount of code you need to recompile, it also reduces the amount of code you need to read in order to find something. When a source code file becomes large, a programmer can waste a lot of time just moving around in the file. There are often many functions and variables that simply do not apply to the specific problem that a programmer is tackling at the moment. A programmer will deal conceptually with a program at many different levels. Having to scroll or search through a lot of code is anywhere from annoying to confusing.

Imagine that you need to find a fix you made to the code a few weeks ago. If all your code is in one large file that is potentially a lot of searching. If you have split the several small files you have a better chance of knowing where to look, thereby reducing your browsing time.
As an example imaging a Game project that has 4 code files: Sound, Graphics, Input, MainGame. Now imagine you want to change some of the graphics – you have potentially ¼ of the code to search.

Facilitate code reuse
If your code is carefully split up into sections that operate largely independently of each other, this lets you use that code in another project, saving you a lot of rewriting later. There is a lot more to writing reusable code than just using a logical file organization, but without such an organization it is very difficult to know which parts of the code work together and which do not. Therefore putting subsystems and classes in a single file or carefully delineated set of files will help you later if you try to use that code in another project.
Such organisation is often found in games, where the games engine is independent of much (if not all) of the actual game.

Share code between projects
The principle here is the same as with the reuse issue. By carefully separating code into certain files, you make it possible for multiple projects to use some of the same code files without duplicating them. The benefit of sharing a code file between projects rather than just using copy-and-paste is that any bug fixes you make to that file or files from one project will affect the other project, so both projects can be sure of using the most up-to-date version.

Split coding responsibilities among programmers
For really large projects, this is perhaps the main reason for separating code into multiple files. It isn't practical for more than one person to be making changes to a single file at any given time. Therefore you would need to use multiple files so that each programmer can be working on a separate part of the code without affecting the file that the other programmers are editing. Of course, there still have to be checks that 2 programmers don't try altering the same file; configuration management systems and version control systems such as CVS or MS SourceSafe help you here. A quick note – There are many such pieces of software that track file use and alteration.

Header Files
When the compiler compiles a source code file, it does so in complete isolation from other source code files. Each source code file is independent and compiles alone. But each source code file expects to use resources found elsewhere, call functions or access variables. It is able to know what's available through the use of header files.
Essentially, the purpose of header files is to communicate what's available in a source code file.

A simple header file contains a list of function prototypes or a class declaration. These are the functions that are available to the world of the program. When a source code file includes a header file, the compiler knows the names and types of classes, variables and functions, and can check for proper matches for function arguments. It can generate the proper code to call the function, even though it doesn't know what the function does.

You generally want one header file for every source file. That is, a SPRITES.CPP probably needs a SPRITES.H file and a SOUND.CPP needs a SOUND.H, and so on. Keep the naming consistent so that you can instantly tell which header goes with which normal file.

Use #include "header_file" rather than #include <header_file> to include your own header files.
In most cases, it doesn't matter which style you use, but there are some subtle differences in how the compiler finds the files. More importantly, it's considered good style to use quote includes for your own header files and angle includes for system headers.

For instance the code would look like this:

```
#include <d3d9.h>
#include <d3dx9.h>
#include <mmsystem.h>
// user defined headers
#include "cMeshes.h"
#include "dxutil.h"
#include "Lights.h"
```

Where the header files in comments (“ “) are user defined.

Classes and Multiple Files

Classes go hand in hand with multiple code files. Classes and multiple code files are a part of the C++ implementation of modularity.
We often use one or two files in which we implement our class, generally a header file and a source file.

In the tutorial we go through adding multiple files and putting our class in them.