Overview

30 September 2019 10:29

Course Homepage

https://bb.imperial.ac.uk/webapps/blackboard/content/listContent.jsp?course_id=_17449_1&content_id=_1583498_1

Aims:

The aim of the course is to extend the students' software engineering skills and programming practice in program design and implementation building up on previous courses (Introduction to Computing and Algorithms and Data Structures).

Learning Outcomes:

Students will be able to use a broader range of C++ features, including object based and object oriented ones, in order to design abstract and reusable software components and make good use of those available in the standard libraries. Students will gain a better understanding of software complexity and of how it can be managed and will be able to apply these concepts to other object oriented programming languages.

Syllabus:

From structs to classes; introduction to OOP: abstraction, encapsulation; function and operator overloading; pointers, references, memory leaks; OOP: inheritance, overriding and polymorphism; templates and the Standard Template Library; exceptions; introduction to Java.

From <http://intranet.ee.ic.ac.uk/electricalengineering/eecourses_t4/course_content.asp?c=EE2-12&s=I2#start>

Skipped - C++ Basics

02 December 2019 14:04

Revision - Basic file input

02 December 2019 14:04

File input

Syntax:

```
#include <fstream>
Std::fstream myInput;

myInput.open("filename.txt");


While(myInput >> text){                <- This reads text until the end of file is reached

        //do something//

}

myInput.close();
```

Skipped - Functions basics

02 December 2019 14:10

Revision - Parameters and Overloading

13 October 2019 10:09

Overloading

Overloading - having two or more function definition for the same function name, the function definitions must have different numbers of formal parameters or some formal parameters of different types. The compiler knows which functions to use by comparing the arguments in the function call.

Changing parameters to const or & will not count as overloading

Default Arguments

You can specify a default argument for a call-by-value parameter in a function. If this argument is omitted on function call, then it is replaced by the default argument.

- The default argument is specified in the function declaration but not in the function definition.
- The default argument(s) must be in the rightmost position

Revision Arrays

13 October 2019 10:32

Declaration

Arrays are declared the following way

Type arrayName[arraySize]; e.g. Int myArray[5] = { 0, 5, 2, 3, 12 }

Indexes start at 0

Arrays in functions

Indexed variables can be used as functioned arguments e.g. myFunction(double myNumber, int myArray[i]){}
If an entire array needs to be passed to a function it is called an 'array parameter'

- Int a[] is an array parameter - this is used in function declaration and function definition
- There is no expression inside the brackets
- It behaves like a call by reference parameter, but is not technically one
- You should include another argument (int) saying the size of the array, as the function/computer has no way to know this, and problems may arise without it
- Use const to ensure the array is unchanged (if you want it to stay the same)

If a function is to return an array, it must return a pointer to the array, as it cannot actually return an array.

Multidimensional Arrays

Syntax:

Type myArray[size1][size2]

A 2D array is an array of arrays.

Function declarations using 2D arrays:

Void myFunction(type myArray[][mySize], int myVar);

Structures & Classes

13 October 2019 10:54

Structures

A structure is an object without any member functions

```
Struct myStruct{
        Int a
        Double b
};
```

To create a variable of type struct:

myStruct myVar;

To access the member variables inside the structure of variable myVar:

myVar.a
myVar.b

Two separate structures may have the same member variable name without causing any problems/errors

Hierarchical structures are structures which member variables are of another struct's type

You can initialise a struct with values once it has been defined, it will initialise in order of the member variables in the struct definition e.g.

myStruct myVar = {5, 0.01};
So myVar.a will = 5, and myVar.b = 0.01

You do not need to initialise all variables (you could just do myStruct myVar = {5}, but if you want to initialise myVar.b, you must also initialise myVar.a as it does it in order). Each data member without an initialiser is initialised to a zero value of appropriate type for each member variables.

Classes

- Classes are like structures but with member functions as well as member variables they are a full-fledged type (e.g. int, double etc)
- They have a private and public section (called access specifier)
- A variable of 'Class' type is called an object
- Member variables are accessed with the dot operator (.)
- Member functions are also accessed with the dot operator, but followed by brackets with the parameters required for that particular function
    - myVar.myFunction()

- Member function definition:
    - returnType myClass :: myFunction(){}
- The :: is called scope resolution operator. It is used with a class name to identify which class the function is a member of.

```
        class myClass{
```

```
Public:
        memberVar a
        memberVar b
Private:
        type memberFunction(){}
};
```

Encapsulation

ADT - abstract data type; you do not know how the values and operations are implemented but you can use the operations. All predefined types are ADTs. New classes should also be implemented as ADTs.
Hiding this information from view is called encapsulation (alternatively information hiding or data abstraction), these all mean that the details of the implementation of a class are hidden from the programmer using the class.
To apply this we make all member variables private.

Public and Private Members

The only things that can access private members of a class, are the member functions of class (these are in public). Usually you will need the following functions then:
getX <--- Accessor function
setX <----Mutator function
To access the member variable(s) X inside of the class. The accessor and mutator functions allow you to control and filter changes to the data (e.g. only allowing the date to be input as a number between 1-31).

Once a member variable is made private there is no way to change or retrieve its value without using a member function. There are no restrictions on public members.
If you do not specify public or private, the members of the first group will automatically be private.

Constructors

13 October 2019 11:55

Constructors are used to initialise member variables of a class, it is automatically called when an object of that class is declared.

Constructor Definitions

- Constructors **MUST** have the same name as the class
- No type can be given at the start of the declaration, and a constructor definition cannot return a value
- Constructors should usually be public

Example:

```
class myClass{
Private:
Int a
Int b

Public:
myClass(int a, int b);   <------- Constructor which will set a and b  ------->
myFunction()….
…..
…..
};
```

The constructor definition can be done in 2 ways

```
myClass :: myClass(int aValue, int bValue){
        a = aValue;
        b = bValue;
}
```

```
myClass :: myClass(int aValue, int bValue)
        : a(aValue), b(bValue)
{
}
```

myClass can now be declared like this:

myClass x(1, 9), myClass y(0, 0);

So there is no need to use get/set functions and also multiple objects can be declared at once

Constructors may also be overloaded, this is useful when making a default constructor e.g.

```
class myClass{
Private:
Int a
Int b

Public:
myClass(int a, int b);          <--- Constructor to set a and b as user specifies
myClass();                      <---- Default constructor when no information is specified -------->

myFunction()….
…..
…..
};
```

The default constructor may be something like this

```
myClass :: myClass(){
        a = 0;
        b = 0;
}
```

If you are using the default constructor (i.e. you are not initialising the variables when making the object) then **DO NOT** use any parentheses.

```
myClass x;
```

This will set the member variables of x to the default values

The reason you cannot include parentheses is because technically:

```
myClass x();
```

Is declaring a function that returns a myClass object and has no parameters

You can also create a class like this

```
y = myClass(1, 2)
```

If you explicitly call a constructor with no arguments you **DO** include parentheses.

```
y = myClass();
```

This will set the member variables of y back to the default.

In these 2 cases the constructor is acting like a function that returns an object of its class type

Using Const&

A constant parameter is where the parameter is given as 'const &myVar'

Inline Functions

For very short functions, the function definition can be placed inside the class definition. It is more efficient so preferred but can cause issues compiling on certain compilers. To define an inline function, use keyword 'inline'.

Static Members

A static variable is a variable shared by all objects of a class, and can be used by objects to communicate with each other/coordinate actions. They are indicated by the keyword 'static' at the start of their declaration. Static variables **MUST** be initialised outside of the class definition, and cannot be initialised more than once. They are initialised in the following way:

```
int myClass :: myVar = 0;
```

A static function is a function that does not access data of any object but is a member of the class. They can be called normally or through the following syntax:

```
myClass :: myFunction()
```

Nested and Local Class Definitions

Classes can be defined inside classes. They can be public or private, and whichever it is, it can be used in member function definitions of the outer class.

Classes may also be defined in a function definition, which would be called a 'local class'. It cannot contain static members.

Operator Overloading, Friends and References

13 October 2019 13:40

Basic Operator Overloading

You can overload any operator (+-=/* etc) so that it will accept arguments of a class type. The syntax of this is:

```
Const myClass operator + (const myClass& a, const myClass& b);
```

The const and & are not necessary but usually used. The 'operator' keyword is necessary. This specific example will return type myClass, but it could be any type.

Returning by const - if returning a class type, use const. For basic types const is unnecessary.

Unary Operators (-, ++, --)

A unary operator is an operator that takes only one operand.
If you overload ++ and -- the normal way it will only work in the prefix position (--x, ++x)

Overloading as member functions

If you overload + in a class definition then the calling object serves as the first parameter -

```
Total = a + b
```

So here the only argument the function takes is b, as a is the calling object. e.g. :

```
Const myClass operator +(const myClass& input2) const;
```

The const is added at the end if the calling object (first operand) is not to be changed.

Overloading the function call operator ()

The function call operator must be overloaded as a member function. It allows you to use an object of the class as if it were a function.

Operators that cannot be overloaded

- Dot operator .
- Scope resolution operator ::
- Sizeof
- ?:
- .*

Overloaded operators cannot have default arguments, and you cannot overload a binary into a unary operator or vice versa (e.g. + must have 2 arguments, ++ must only have 1).

Friend functions and Automatic Type Conversion

Friend functions are non-member functions that have all the privileges of member functions.

Automatic Type Conversion

If you call a function but your input parameters are not of the same type as any function(e.g. class type), the system will next look for if that function has been overloaded for the combination of inputs you put in, if it cannot find any then it will see if there is a constructor that takes a single argument that is of the type you put in. It uses that constructor to convert the type into the class type required, this will not work unless there is a suitable constructor.
Conversion only works for arguments not calling objects, so it must be the 2nd object (e.g. b in a + b).
If you have overloaded as a non-member then it doesn't matter which order as the conversion will work either way.

Friend Functions

A friend function of a class is not a member function of the class but has access to the private members of that class like a member function does. Friend functions **MUST** be declared in the class definition. You can make friend operator or function by copying the operator or function declaration in the definition of the class and using the keyword 'friend' in front of it.
Friend functions are defined normally (without the ::) and 'friend' is not needed in the definition, they have automatic type conversion of all arguments just like non-member operators.

```
Friend int myFriendFunction();    <---- declaration of a friend function
```

Friend Classes

If class a is a friend of class b, then every member function of class a is a friend of class b. Typically the class reference each other in their class definitions. To do this you must include a forward declaration in the class defined first to the class defined second.

```
Class a;          <---------- forward declaration

Class b{
        Public:
        ….
        Friend class a;
        …
        Private:
        ….
};

Class a{
….
….
};
```

References and specialised overloading operators

Overloading << and >>

Syntax:

```
Class myClass{
        Public:
                Friend ostream& operator <<(ostream& os, const myClass& myVar);
                Friend istream& operator >>(istream& is, const myClass& myVar);
};


ostream& operator <<(ostream& os, const myClass& myVar){
        //some code e.g. os << myVar.myFunction()  '\n'; //
        Return os;
}
istream& operator >>(istream& is, const myClass& myVar){
        Char x;
        In >> x;
        //some code//
        Return in;
}
```

These should always be declared as friend in the function declaration if inside the class definition, they cannot be members of the class being input/output. The ostream and istream

must be called by reference.

Overloading assignment operator =

This should be done as a member of the class.

Revision - Strings

14 October 2019 13:12

There are two types of representing strings of characters

1. An array of characters with the null character ('\0') at the end to show where the string stops (an 'end marker'). If you use double quotes (e.g. "Hello") in C++ they are implemented in this way
2. Using the class 'string'. A modern addition to C, and generally advisable to use this one as it offers more functionality.

Strings as Arrays

You can initialise a C-string variable when you declare it:

Char myMessage[] = "Hi there.";

You can safely omit array size as C++ will automatically make the size needed. Note that the following two things are not equivalent

Char myMessage[] = "abc" =/= Char myMessage[] = {'a', 'b', 'c'}

One of these has a null character at the end, the second does not. Initialising with the double quotations automatically puts a null character at the end.

When manipulating the string be careful **NOT** to replace/remove the null character at the end, or it will no longer behave as a C-string.

Using C-string requires an include

#include <cstring>

Using = and == with C-strings

Other than in declaration, you cannot use the = with C-strings. If you wish to assign a value to a C-string variable you must use something else.

- Using 'strcpy' -------> strcpy(myString, "hello");
  - This method does not check for size

To test for equality you cannot use ==, you must use 'strcmp', this cycles through the characters until it reaches the end or there is a difference in characters.

- If the strings are the same, 0 is returned
- If the character from the first string > second string, a positive number is returned
- If the character from the first string < second string, a negative number is returned
- This means that if you want to use this as a bool to test for equality, any non-zero value will be converted to true, and 0 to false, so you must **reverse** the logic.

Character Manipulation

Character I/O

Sometimes, you may not want to use >> to input values, as it does everything automatically. The function .get() takes an argument of type char, which receives the input character that is read from the input stream. This will also read blank spaces, new lines, etc that >> would skip.

Example:

Code:

```
Char c1, c2, c3, c4, c5;
Cin.get(c1);
Cin.get(c2);
Cin.get(c3);
Cin.get(c4);
Cin.get(c5);
```

This would store the following:
C1 = 'A'
C2 = 'B'
C3 = '\n'
C4 = 'C'
C5 = 'D'

Terminal:

```
AB
CD
```

If you had used >>, the \n would not have been stored.

You can use this get function to also input characters until a specific one has been input using a do while loop.

```
Char myInput
Do{
```

```
        Cin.get(myInput)
        ….
        ….
    }while(myInput != '\n')
```

So this one will stop taking inputs when a newline (enter) is input.

Put is the output form of get, but does not achieve anything that << could not do.

Putback, peek and ignore

Putback reads the next character and then removes it.
Peek looks at the next character but does not store it.
Ignore skips over an input until a certain point e.g. (cin.ignore(1000, '\n') ignores the input until newline, or 1000 characters have been skipped).

Using the class 'string'

When using class string you do not need to worry about size, as it will be automatically adjusted/allocated for you during operations. To use the class string you must include it.

```
#include <string>
```

When you enter something in quotation marks, it is actually a C-string, if you assign it to a type string, there is an automatic conversion that will take place to allow this. e.g.

```
myString = "hello"   (this is also equivalent to myString("hello)   )
```

I/O with the class 'string'

If you try to input a string, the >> operator will stop at whitespace or a newline, so if you wish to input a full sentence or something with multiple lines/with whitespace you must use 'getline'.

```
Std::string myString;
Getline(cin, myString);
```

The get function can also be used for string input, although it is technically reading characters.

You can use [] to access a specific character in a string. e.g. myString[i]. There is no check for an illegal index here, so make sure to account for this (usually if you are looping through or otherwise you can use myString.size() - 1 to make sure you have not gone too far). The function 'at' can check for illegal index values.

Converting between class string and C-strings

Although there is automatic conversion for things like myString = "abc", you cannot set myCstring = myString. There is no automatic conversion from the string objects to C-strings. Explicit conversion can be done using 'c_str()'

Revision - Pointers and Dynamic Arrays

14 October 2019 14:09

Pointer

A pointer is the memory address of a variable. A pointer can be stored in a variable, and must be declared to have a pointer type.
Example:
Double *p <---- this p can hold pointers to variables of type double
p = &a <----- & is the dereferencing operator, which gives the address of the variable it is dereferencing

If you then write
a = 2
*p = 4
a will be set to 4, as *p and a refer to the same variable

If you have 2 pointers: p1, p2 and set p1 = p2, you are setting the addresses to be equal not the values, if you wanted to change p1 to the value of p2 then *p1 = *p2

A function can return a pointer, or have a pointer in its arguments:
Int* myFunction(int* p);
This function will return a pointer to a variable of type int, and takes in a pointer to a variable of type int.

Using typedef means you don't have to use the * every time you create a pointer.
e.g. typedef int* IntPtr
Everytime you now want to create a pointer to an int, you can just write 'IntPtr myPointer;'. When you call a pointer by reference you will have to only type myFunction(IntPtr& myPointer), whereas without this typedef, you would have to write myFunction(&*myPointer) which might get confusing.

Using the 'new' keyword

If you want to create a variable that has no identifier to serve as a name you can use the keyword new
E.g. p1 = new int;

This nameless variable can be accessed then using *p1, without having to make another variable for it.
'new' produces a new, nameless variable and returns a pointer that points to this new variable, the type is specified after the 'new' operator. These variables are called dynamically allocated variables.

This can be used for classes as well
E.g. myPtr = new myClass;

If there is insufficient space to create a variable with 'new' then the program will terminate.

After you have finished using your dynamic variable you can delete it with the 'delete' operator, so that the memory can be reused. It destroys the variable that the pointer is pointing to, and the pointer that was being used now is undefined, setting this = NULL when you use 'delete' will prevent any errors.

Dynamic Arrays

A dynamic array is an array whose size is determined whilst the program runs, rather than when you write the program.
We create them using 'new'.
IntPtr a;
a = new int[arraySize]
Once we are finished with the array we write 'delete [] a';

Title

The -> operator

The arrow operator combines . and *. Example:
Struct myStruct {
Int var
….

```
};

myStruct *p;
p = new myStruct;
p->var = 2000;
```

The 'this' pointer

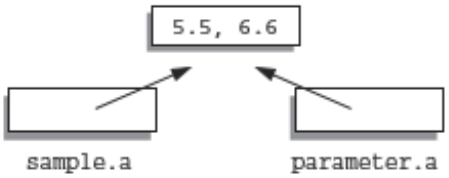
The 'this' pointer points to the calling object, so you cannot use this in the definition of any static member functions.

Copy Constructors

Syntax:

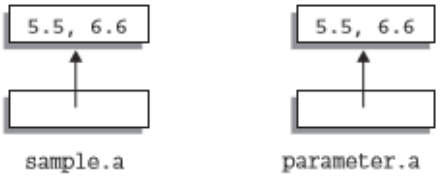
```
myClass(const myClass& myVar);
```

Default copy constructor works fine except where you use pointers or dynamic data in a class' member variables. Copy constructor is invoked if the LHS is not already of type myClass. If you do not create a copy constructor and you have pointers (or dynamic data) then when the values are copied you will end up with:



Screen clipping taken: 02/12/2019 18:45

If default copy constructor is used



Screen clipping taken: 02/12/2019 18:45

If manual copy constructor is used

Destructors

Syntax:

```
~myClass(){
    Delete[] myDynamicVariable;
}
```

Assignment Operator

Syntax:

```
myClass& operator =(const myClass& myVar);
```

The assignment overloaded operator is invoked if the LHS is already of type myClass.

Remember to check for self-assignment:

```
If(this == &myVar){
    Return *this;
}
```

Separate Compilation and Namespaces (Encapsulation)

14 October 2019 21:31

Header files

.h or .hpp indicates header files, and program that wants to use the class 'myClass' must have #include myClass.hpp at the start of the program.

Using ifndef

C++ doesn’t allow you to define a class more than once, so you can add a section in the includes, which tells it not to include again if it's been included before:

So in the header file you write at the top:
#ifndef myProgram
#define myProgram

Then at the end you write:

#endif

So now whenever you try to include myProgram, the class will only be defined one time, and every time the compiler encounters the file again, it will skip from ifndef to endif, so the class is not defined again.

Namespaces

I/O streams

02 December 2019 14:41

Basics

Stream is a flow of data.

Instream >> myNumber
Puts whatever is in input file connected to instream into number.
Outstream << myNumber
Puts whatever is in number into the output file connected to stream outstream.

```
#include <fstream>

Std::ifstream instream;
Std::ofstream outstream;
```

Stream variables must be connected to files using the .open("filename"). Files should be closed with .close() after finishing. You can do the opening in one line:

Syntax:

```
Std::ifstream instream("filename.txt");
```

You cannot assign a value to a stream variable, you can have a parameter of stream type, but must be called by reference.

Appending files rather than overwriting

Syntax:

```
Outstream.open("filename.txt, std::ios::app);
```

Ios is defined in <iostream>.

Testing a file was opened successfully

```
Instream.open("filename.txt");
If(instream.fail()){
    Std::cout << "Failed to open file" << std::endl;
    Exit(1);
}
```

Include <cstdlib> for access to exit functions.

End of file

```
Do{
Instream.get(next);
//some code //
}
While(!Instream.eof());
```

You can also use the instream >> myVar in the while instead and the code looks cleaner.

```
While(instream >> myVar){
    //some code//
}
```

Getting a filename from user

```
Std::string filename;
Ifstream instream;
Getline(std::cin, filename);
Instream.open(filename.c_str() )
```

Open takes in a cstring as parameter, so if you use string variable type, a type conversion must be made.

Skipped - recursive functions

22 October 2019 16:19

Inheritance

22 October 2019 16:04

Inheritance is where a new class, (**derived/child**) is created from another class (**base/parent**). The derived class automatically has all the member variables and functions that the base class has, but can have additional member functions and variables.

Syntax:

```
Class myNewClass : public myClass
```

Inherited functions can be changed in the definition of a derived class, this is **redefining**. This is done the same way as you would just write a new function in the class. (e.g. it must be defined and then implemented, but if you don't want to redefine it, you don't define it and you will be able to use it as its original functionality).

Redefining and overloading are **not** the same. Overloading requires a change in parameter list and new function whilst still keeping the original function, whereas redefining could change or leave the parameter list the same, and changes the function entirely.

If you have redefined a function in the derived class, but want access to the original function in the base class you can use the scope resolution operator with the name of the base class.

Syntax:

```
myVar = new myNewClass
myVar.myClass::myFunction();
```

This will invoke the myFunction as it is written in myClass, rather than as it is redefined in myNewClass.

Constructors are not inherited in the derived class, but can (and should) be invoked when the defining the constructor for the derived class.

Syntax:

```
Class Base{
        Public:
                Base();
};

Class derived{
        Public:
                Derived() : Base();
};
```

Private functions, destructors, copy constructors and the assignment operator (=) are not inherited in the derived class. If you do not define a copy constructor in your derived class, C++ will create one for you.

When you call destructors in derived classes, you must do it in reverse order to the constructors. e.g. if A->B->C, so C is derived from B, B is derived from A, then when you call destructor for C, you must call C destructor, then B destructor, then A destructor.

Objects of a derived class has more than one type

This means for 'myVar = new myNewClass', I can use functions both in myClass, and myNewClass when using myVar.

Using 'protected' rather than private or public

When you use protected qualifier, the derived class can have access to it, but for other things, it is for all intents and purposes the same as 'private'.

When you define a derived class, you may use:

Class myNewClass : public myClass
Class myNewClass : private myClass
Class myNewClass : protected myClass

If you use 'protected' then members that are public in the base class, will become protected in the derived class when they are inherited. If you use 'private' then all members in the base class will become private (inaccessible). If you use protected or private, then the derived object is **not** a member of the base class.

Multiple Inheritance

A derived class may have more than one base class. Don't do this unless you are 5head.

Polymorphism and virtual functions

22 October 2019 16:56

Virtual Functions

Virtual functions are functions that can be used before they are defined. When a function is labelled as virtual, it means C++ will get the correct implementation corresponding to the calling object.

Overriding - If a virtual function definition is changed in a derived class (redefining is where the function is not virtual)

The virtual property is inherited, so if the base class has: virtual void myFunction(), the derived class may simply write void myFunction() - it is fine to write virtual void too.

- Using virtual classes increases the overhead, using more storage and making the program run slower. If you don't need to make it virtual, then it's best not to.

If you need a virtual function in your subclasses, but have no use for it in the base class, you may make a **pure virtual** function, where you needn't give any definition to it.

Syntax:

Virtual void myFunction() = 0;

Adding = 0, signals that the virtual function is pure.

Abstract class - A class that uses pure virtual functions, it is only used as a base class to derive other classes.

Pointers and Virtual Functions

Objects of derived classes may be assigned to a variable of type base class (but not the other way)

Example:

Base *temp = new Derived;

When doing this always use pointers, otherwise it will **slice** the derived class, and just be of type Base. When using pointers you may access the functions/variables from the derived class, if you do not use pointers you may only access the ones of the Base class.

- Pointers to Ancestors may be set equal to pointers of descendants
  - pAbove = pBelow

Destructors should be virtual for these reasons, to ensure that the derived destructor is called even when set to a type pointer to base class. Simply marking the base class destructor as virtual will ensure all derived classes are also.

When we make a call to a member function using a pointer, it does not use the type of the pointer to decide what function to use.

Polymorphism = late binding = virtual functions

All essentially mean the same thing.

- Late binding: The decision of which version of a member function to use it decided at run time
  - Virtual: Member function that uses late binding
    - Polymorphism - another word for late binding.

Templates

01 December 2019 15:25

Syntax:

Template<class T>

This is the template prefix - it tells the compiler that the function follows a template and that T is a type parameter.

- You cannot have unused parameters - each parameter must be used in the template definition

Separate compilation can sometimes cause errors - place the template in the file where it is invoked, and have the definition appear before the first invocation of the template.

Placing the template before a class definition makes a class template.

Syntax:

```
Template<class T>
Class myClass{
    …
    Void myFunction()
};
```

Function definitions syntax: (don't forget the <T> after the class name)

```
Template<class T>
Void myClass<T>::myFunction(){
…
}
```

Type Definitions

Skipped - Linked data structures

01 December 2019 16:05

Exception Handling

01 December 2019 16:07

Try - Catch blocks

Syntax:

```
Try{
..//Some Code//..
}
Catch(some value){
        Give error message here
}
```

Try blocks will execute until an exception is thrown.

Throw Statements

Throw //some value//
You can throw a value of any type.
When something is thrown the code in the try block stops executing and the code in the catch block begins execution.

Catch Block parameters

Catch block parameters are of type name that specifies what kind of thrown value the catch block may catch.

Defining Exception Classes

```
Class myException{
Public: myException(int number)
        Int getNumber()
Private:
        Int number
};

Int main(){
        Int someNumber)
        Try{
                …
                If( myVar <= 0){
                        Throw myException(someNumber)
                }
        }
        Catch(myException e){
                Std::cout << "my error message" << e.getNumber()<< std::endl;
        }
Return 0;
}
```

When you throw an exception (e.g. myException(someNumber), you are invoking the constructor for class exception, and that object is then thrown, so when you catch it you may use any of the member variables/functions of that class.

Multiple Throws and Catches

Catches can only catch exceptions of one type, so placing more than 1 catch block after the try block allows you to catch exceptions of different types.

The order of catch blocks may be important. When an exception is thrown the catch blocks that follow are tried in order and the first one that matches the type of exception thrown is the one that is executed. If you include a default catch block place it at the very end.

Default catch block syntax:

```
Catch(…){
        //some code//
}
```

The ellipsis means it will catch any type.

Exceptions may not need member variables/functions, you may just have:

```
Class myException{};
```

Which is just used to get you into the catch block, so you can omit the catch block parameter as there is nothing you can do with the exception.

Exceptions in functions

```
Try{
        myFunction(a, b)
}
Catch (myException){
        //some code//
}


myFunction(a, b){
        //some code//
        If(b ==0){
                Throw myException()
        }
        Return a;
}
```

Exception specification

If there are exceptions that might be thrown but not caught in the function, they should be listed in an exception specification (throw list).

Syntax:

```
myFunction(a, b) throw (myException, myOtherException);
```

This must be in the function definition and declaration.

If there is no throw list then the code behaves as if all possible exception types were listed in the throw list. If an exception is thrown that is not listed in the throw list, then the behaviour is unexpected and terminate() is called.

If the function should not throw any exceptions not caught inside the function you can use an empty throw list.

Syntax:

```
myFunction(a, b) throw ();
```

If you redefine or override a function definition in a derived class, it must have the same or a subset of the throw list it had in the base class (i.e. you cannot add any more exceptions, but you may remove some).

When to throw exceptions

```
void functionA( ) throw (MyException)
{
        .
        .
        .
    throw MyException(<Maybe an argument>);
        .
        .
        .
}
```

Screen clipping taken: 02/12/2019 11:46

```
void functionB( )
{
        .
        .
        .
    try
    {
        .
        .
        .
        functionA( );
        .
        .
        .
    }
    catch(MyException e)
    {
                <Handle exception>
    }
        .
        .
        .
}
```

Screen clipping taken: 02/12/2019 11:46

Throw statements should be used within functions and listen in a throw list for the function. They should be used when the way the exception condition is handled depends on how and where the function is invoked, otherwise avoid throwing an exception

If an exception is thrown and not caught, terminate() is called.

Standard Template Library

02 December 2019 12:41

Iterators

An iterator is a generalisation of a pointer, and acts similar to one. The following operators can be used with iterators

- ++
- --
- == (and !=)
- Dereferencing operator (*) - in some cases this is read only, and in others you may use it to change the value of the variable.

Syntax:

```
STL_container<datatype>::iterator p;
for (p = container.begin( ); p != container.end( ); p++)
   Process_Element_At_Location p;
```

Screen clipping taken: 02/12/2019 13:13

Where STL_container is the name of the container class (e.g. vector, array), datatype is the type (e.g. int) of items stored in that container.

Random Access

Random access is where you go directly to a particular element in one step, e.g. using [i], and can be used with iterators.

Syntax:

Container[i]
p[i]
*(p + i)

All allow random access at I.

Constant and Mutable Iterators

If the iterator is const, then *p will be read only (you cannot change the element in the container that p points to).

Syntax:

Vector<char>::const_iterator p;

Reverse Iterators

If you wish to reverse traverse through a container you cannot simply do p-- instead and keep code the same, you must declare the iterator as reverse:

Syntax:

```
STL_container<datatype>::reverse_iterator rp;
for (rp = c.rbegin( ); rp != c.rend( ); rp++)
    Process_At_Location p;
```

Screen clipping taken: 02/12/2019 13:26

Containers

Container classes include: lists, queues and stacks, vectors and arrays.

Sequential containers

E.g. linked lists. It arranges its data items into a list such that there is only one link from one location to another.

You cannot use random access on some sequential containers (list) but you can on some (vector).

Generic Algorithms

REVISION - WINTER EXAM

02 December 2019 15:39

Concept	Understanding 😀😐😶😭	What to work on	How will I do this?
C++ Basics	😀		
File I/O	😐	Remembering syntax/functions	Practise examples
Functions Basics	😀		
Parameters & Overloading	😀		
Arrays	😀		
Structures & Classes	😀		
Constructors	😀		
Operator overloading …	😐	Remembering syntax/function	Practise examples
Strings	😀		
Pointers and Dynamic arrays	😐	Understanding content, remembering syntax/functions	Read through some notes, watch videos, practise examples
Separate Compilation	😀		
I/O streams	😐	Remembering syntax/functions	Practise examples
Recursive functions	😐	Forgotten from last year	Quick overview on content/do labs from last year
Inheritance	😀		
Polymorphism and virtual functions	😀		
Templates	😶	Syntax/when to use	Examples
Data structures	😶	Forgotten from last year	Quick overview/do labs from last year
Exception Handling	😐	Syntax	Examples
STL	😶	When to use/purpose	Revision/examples

Topics/Questions I'm not sure on:

- When/How to use destructors
- Why have a function type reference (e.g. lecture 5)
- Why return *this (e.g. lecture 5)
- When to delete if I use 'new' keyword
- Main where it takes in arguments
- When to make something protected?
- Lecture 7 pg. 22?
- Makefile flags/etc
- Iterators? Const_iterator?

What's in the exam

- Reading / writing text files
- 2 derived classes of a base class
- Overload insertion operator in base class
- Exceptions (thrown by member functions)
- Storing in a vector of pointers / list
- Sorting algorithms

Lectures/Lab notes

Week 1

Lecture:

- Use dot operator to access member variables/functions myVar.myFunction()
- Different structs/classes can have same name functions/variables as members without conflict
- Declare everything in the header file (.hpp), and define functions in the cpp file
- Functions must have a class specified in the cpp file
  - Void myClass::myFunction();
- Use getters and setters to change private member variables
- Structures typically all public and no member functions, Classes all variables private and member functions public

Labs:

- Makefile:

```
all: Point.o main.o
	g++ Point.o main.o -o prog

point.o: Point.cpp Point.h
	g++ -c Point.cpp
```

All program files by compiling cpp files linked here

Compile the cpp files

```
        g++ Point.o main.o -o prog

point.o: Point.cpp Point.h
        g++ -c Point.cpp

main.o: main.cpp
        g++ -c main.cpp

run:
        ./prog

clean: rm *.o prog
```

All program files by compiling cpp files linked here

Compile the cpp files

Use this command to run the program (e.g. make run will run the program or you can manually type make, then ./prog

Screen clipping taken: 05/12/2019 11:54

- Header files:
  - At start: #ifndef H_CLASSNAME, #define H_CLASSNAME, at end: #endif
  - This stops the file being included multiple times which may cause errors

Week 2

Lecture:

- To function overload simply have the same name but different parameters input
- Function overloading cannot only differ by return type
- If call by reference, then changes to myVar inside the function apply to the parameter myVar also.
- Call by const if parameter is not to be changed.
- Call by const reference for efficiency (a copy is not made, but parameter is also not to be changed).
- Const member function (e.g. void myFunction() const;) will not modify the current state
- Constructor (myClass() in hpp, and myClass::myClass() in cpp) does not have a return type, and has same name as class
- Constructors can only be called on object creation, not after.
- Create a default (myClass()) and parametrised constructor (myClass(int a, int b)).
- Overload operators as friend functions in the class.

Labs:

Week 3

Lecture:

- Copy constructor - initialises an object of myClass, from one already defined (myClass(const myClass &myObject);)
- Dynamically allocate objects (myClass *p1 = new myClass();), using the pointer means this is dynamic.
- Dynamically allocated objects need destructors (~myClass();)
- Once an object's scope end's the destructor is automatically called, you can call it manually with 'delete'.

Lab:

- To pass in arguments, write: int main(int argc, char *argv[])

Week 4

Lecture:

- -> is equivalent to *myPointer.
- Deleting dynamic arrays: delete[] myArray;
- Only use delete with dynamically allocated (i.e. things made with 'new')
- This points to current object
- Pointer to const means we cannot change that location's value using that specific pointer, but can be changed by pointers not declared as constant
- Assignment operator - myClass& operator=(const myClass& myVar){ … return *this; }

Labs:

- User defined destructor in cpp file (e.g. for polynomial delete [] coefficients)
- Declare function as reference, and return *this to allow assignments

Week 5

Lecture:

- Inheritance: how member variables/functions are available to derived classes

Base	Private	Protected	Public
Derived	Not accessible	Private (only members can access)	Public
Derived Again	Not accessible	Not accessible	Public

Labs:

Week 6

Lecture:

- Class Derived : Public Base
  - Means the public and protected parts of base class are added to interface of derived.
- Member functions can be redefined in the derived class.
- Classes don't inherit constructor, destructor or assignment operator
- If derive as protected, all public and protected base members become protected

- If derive as private, all public and protected base members become private
- Derived objects may be converted to Base objects, but slicing may occur (member variables in derived will be removed from derived)
- Pointers to base objects can be redirected to point at a derived object.
- If functions are declared virtual, then the choice of which to perform will be decided at runtime
- Destructors may be virtual (for instance if you are using a base pointer for an object of derived y

```
template <class Type>
// or template <typename Type>
void my_swap(Type& a, Type& b) {
    Type c;
    c = a;
    a = b;
    b = c;
}
```

When to use typename rather than class?

Labs:

Week 7

Lecture:

- Templates generalise functions (providing operators are overloaded correctly)
  - Syntax:
- Compiler creates actual versions of the function when called with specified parameters
- Template declaration is not full declaration, so fully specify template in header file
- You can overload templates, and have multiple parameters in the parameter list
  - Template<class T1, class T2>
  - T2 myFunction(…) {…}
- When overloading the most specific version will be selected
- To force a version use myFunction<type, type>(n1, n1) e.g. here it uses the 2 parameter version
- Class templates are the same and must have declaration and definition in header file
- STL: Containers (vector), Iterators, Algorithms etc.
- Iterator: like a pointer for containers
  - Vector<int> :: iterator it; Blue part is the type, 'it' is the variable.

```
for(it = v.begin(); it != v.end(); it++) {
   cout << *it << endl;
}
```

  - It is like a pointer so needs to be dereferenced

Labs:

Week 8

Lecture:

- Can use [] or .at() function for vector
  - At function will do a range check
  - You can do v.at(x) = y;
- After opening a file, check if it is open in if/else statement
- Use try-catch blocks for exceptions, when function throws an exception execution stops and control goes to the catch block, or up one level until a try block is encountered
- Defining own exceptions:
  - Class myException : public standardException{
    Public:
    myException(const string& what) : standardException (what) {}
    };
  - Order of this matters, put most specific first after try block, and last most generic.

Labs:

OTHER HELPFUL THINGS

- If the input is not always same number of items (3 3 3, 1, 2 2) use getline, then stringstream:
  Syntax:
  While(std::getline(infile, line){
  std::stringstream s (line)
  Std:string temp
  While(s << temp){
  // … //
  }
  }
- Static variables are shared by ALL objects of that class. If it is changed, it changes for all.
- Static functions can only access static fields