

## MTRN2500 - Computing For Mechatronic Engineers



### **Summary of Course 2017 Semester 2** **Peter Bassett**

*This is a summary of the MTRN2500 course completed by previous student Peter Bassett. Please leave comments if you have any questions!*

## Abstract

The following document is a personal set of notes for the course MTRN2500. Much of the information has been taken from lectures and examples throughout the course. I am not a professional so please do not rely on these notes as a complete source of information. They should be used in conjunction with the lectures in order to understand more difficult concepts. Some of my examples and solutions to questions are similar to those found in tutorial questions. Please do not plagiarize, it is very obvious to your demonstrators and you are only disadvantaging yourself.

# Table Of Contents

<b>1.The Memory Model of a Computer: Stack, Data, Code and Heap</b>	<b>4</b>
<b>2. All C/C++ Programming Constructs</b>	<b>7</b>
2.1 For, do-while and while loops	7
2.2 If and Else-If Statements	9
2.3 Switch - Case - Break - Default	9
<b>3. Fundamental Data Types</b>	<b>12</b>
3.1 Char, short, int, long, unsigned char, unsigned short, unsigned int, unsigned long	12
3.2 Float, double, long double	12
3.3 Void	13
<b>4. Writing Basic Procedural Functions</b>	<b>13</b>
6.1 Address-of operator (&)	15
6.2 Dereference operator (*)	15
6.3 Declaring Pointers	15
6.4 Pointers and Arrays	16
6.5 Pointer Arithmetic	16
6.6 Pointers and const	18
6.7 Pointers and String Literals	18
6.8 Void Pointers	19
6.9 Invalid and NULL pointers	20
<b>7. Creating User Defined Structures and Classes</b>	<b>21</b>
7.1 Structs, Unions and Classes	21
7.1.1 Structs (Data Structures)	21
7.1.2 Unions	21
7.1.3 Classes	22
7.2 Public, Protected and Private Access Attributes	23
7.3 Member data and Member Functions	23
7.4 Constructor, Copy Constructor, Destructor	24
7.4.1 Constructors	24
7.4.2 Overloaded Constructors	25
7.4.3 Copy Constructors	26
7.4.4 Destructor	27
7.5 Virtual, Friend, Operator and Pure Virtual Functions	27
7.5.1 Friend Functions and Classes	27
7.5.3 Inheritance	30

7.5.4 Virtual Members	31
7.5.5 Pure Virtual Functions	32
7.5.6 Overloading Operators	33
As member functions	33
Use of friend functions (non-member functions)	36
7.6 Static Member Data and Static Member Functions	37
7.6.1 Static Member Data	37
7.6.2 Static Member Functions	39
<b>8. Developing Class Hierarchies</b>	<b>40</b>
8.1 Use of Virtual Functions	40
8.2 Use of Virtual Destructors	40
8.3 Use of Abstract Classes	42
<b>9. Pass Through objects</b>	<b>43</b>
9.1 Pass by reference	43
9.2 Pass by Value	43
9.3 Return by Reference	44
9.4 Return by Value	44
<b>10. Class Derivation and Inheritance</b>	<b>45</b>
<b>11. Abstract Classes. How to Distinguish between abstract classes and non-abstract classes.</b>	<b>45</b>
<b>12. Overloading</b>	<b>46</b>
12.1 Function Overloading	46
12.2 Operator Overloading	46
<b>13. Function Templates and Class Templates</b>	<b>47</b>
13.1 Syntax of Templates	47
13.2 Use of Templates	47
Function Templates	47
Class Templates	48
13.3 Container Class Vector	49
13.4 Iterating Through Vector of Objects	52
13.5 Accessing Object Data Using Iterators	52
<b>14. Type Casting</b>	<b>55</b>
14.1 Ordinary Type Casting	55
Implicit Conversion	55
Explicit Conversion	55
14.2 Dynamic Casting	55
<b>15. Streams</b>	<b>57</b>

15.1 How Data Are Packed as Bytes	57
15.2 How to Alter Data Packing in Memory	57
15.3 Instantiating Stream Objects for Input and Output	58
Output: The Insertion Operator (<<)	59
Input: The Extraction Operator (>>)	59
15.4 Reading and Writing Object Data to an ASCII file	59
15.5 Reading and Writing Object Data To Binary	62
Writing Object Data to Binary File	62
Reading Object Data to Binary File	63
15.6 How to Randomly Access Data in Binary File	64
<b>Question 1 - Example</b>	<b>65</b>
<b>Question 2 - Example</b>	<b>70</b>
<b>Question 3 - Example</b>	<b>73</b>
Solutions	74
<b>Question 4 - Example</b>	<b>83</b>
<b>Question 5 - Example</b>	<b>87</b>
Solutions	87
<b>Question 6 - Example</b>	<b>92</b>
Solutions	93

## 1.The Memory Model of a Computer: Stack, Data, Code and Heap

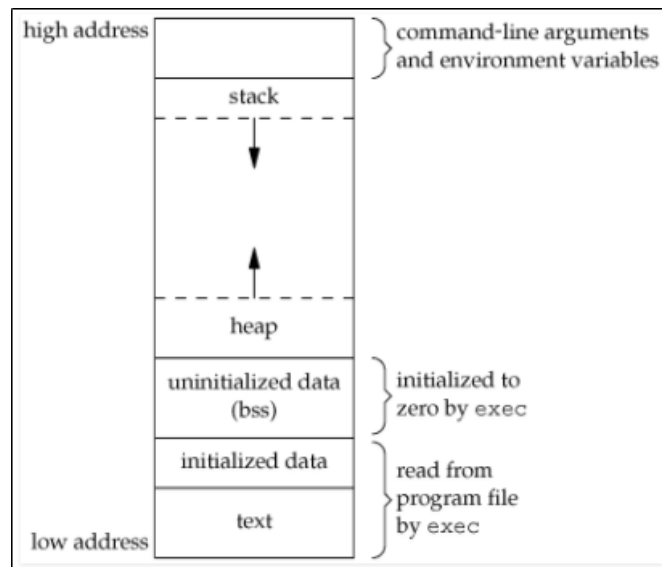


Figure 1.1: Memory Model of Computer

### Code Segment (Text)

- This is the section of a program which contains executable instructions.
- It is placed below the heap or stack to prevent heaps/stack overflows from overwriting it.
- Often the code segment is in a read-only format to prevent a program from accidentally modifying it.

### Data Segment

- **Initialised Data Segment**
  - Contains the global variables and static variables that have been initialised by a programmer (no non-zero variables).
  - It is not read-only as the variables can be altered during runtime.
  - *Examples: `static int i = 10;` `global int I = 10;`*
- **Uninitialized Data Segment**
  - Often called the “bss” segment
  - This is where data that is initialised by the kernel (core of the operating system) to 0 before the program executes is stored.
  - *Examples: `static int j;` `global int p;`*

### Stack

- This is where the memory for automatic variables is contained within functions.
- Follows a *Last in First Out (LIFO)* device where the new storage is allocated and deallocated at only one end called the top of the stack.

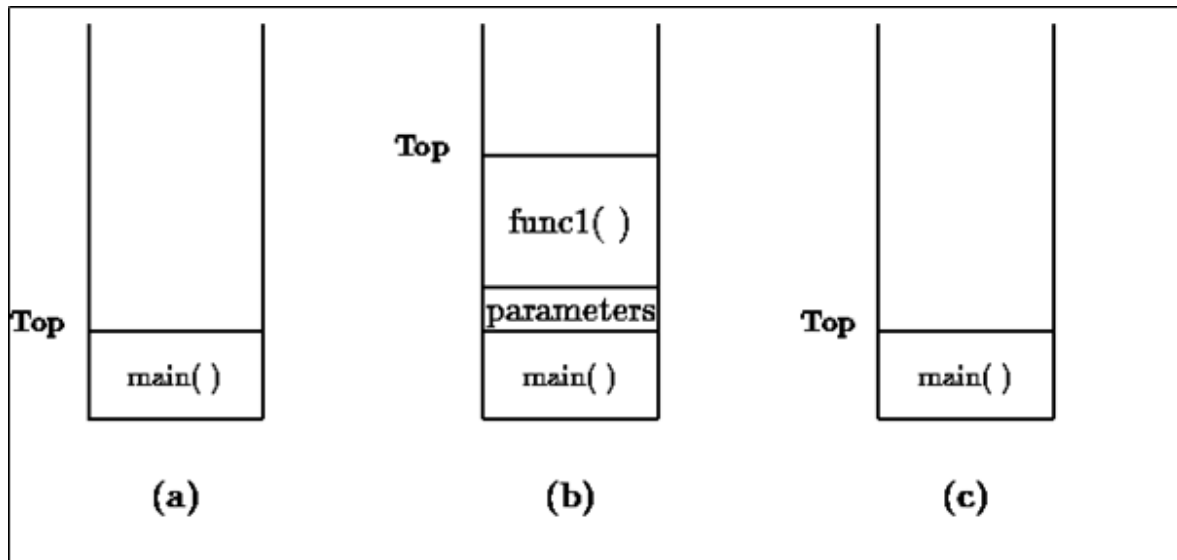


Figure 1.2: Organisation of Stack

- The above figure demonstrates the process that occurs within the Stack.
- As seen, when a main() function is executed, space is stored at the top of the stack for all the variables. If the main() function called a func1() then more storage is allocated for the function at the top of the stack. Those parameters that were passed from main() to func1() are also stored. When func1() returns, the storage for its local variables is deallocated and main() returns to the top of the stack.

#### Heap

- This is where the dynamic memory allocation takes place
- For example memory allocated in a funcA() will still be valid after funcA() returns.
- A drawback of using heap is that if you forget to release heap-allocated memory then you may exhaust it. This is sometimes called a memory leak.

## 2. All C/C++ Programming Constructs

### 2.1 For, do-while and while loops

**For Loops** - Constructs a loop that executes a specified number of times

```
for (initial expression, end expression, incrementation) {  
  
}
```

*Figure 2.1.1 For Loop Construction*

```
int main() {  
    //Declaring the counter in the loop  
    for (int i = 0; i < 2; i++) {  
        //do something  
    }  
  
    //Declaring the counter outside the loop  
    int i;  
    for (i = 0; i < 2; i++) {  
        //do something  
    }  
  
    int i, j;  
    for (i = 0, j = 0; i + j < 10; i++, j = j + 2) {  
        //do something  
    }  
}
```

*Figure 2.1.2 Examples of For Loops*

**While Loops** – A loop that tests an expression before it executes. It therefore will execute zero or more times. A While loop can also be terminated using a *break*, *goto*, or *return*. The statement *continue* can be used to terminate the current iteration and continue onto the next iteration.

```
int main() {
```



```

//Simple example
int myNumber = 0;
while (myNumber != 1000) {
    //do something
    myNumber++;
}

//Break and continue
int myNumber = 0;
while (myNumber != 1000) {
    if (myNumber % 2 == 0) {
        continue;
    }

    if (myNumber % (myNumber - 50) == 0) {
        break;
    }
}

//Simple use of nested loops

int myNumber = 0;
while (myNumber != 1000) {
    while (myNumber != 500) {
        if (myNumber % 10 == 0) {
            goto loop_end
        }
    }
} loop_end;
}

```

*Figure 2.1.3 Example of While Loops*

**Do While Loops** – A while loop where the test of termination condition is executed after each iteration. Therefore it will execute one or more time. Similarly, it can be terminated using *break*, *goto*, or *return*.

```

int main() {

```

```

    ///Simple example
    int i = 0;
    do
    {
        //something
        i++;
    } while (i < 3);
}

```

*Figure 2.1.4 Example of Do While Loop*

## 2.2 If and Else-If Statements

If statements evaluate a condition and execute a command given that the condition is achieved. An *else* statement can follow, which will execute a different command given failure of the condition.

```

int main() {

    ///Simple example
    int i = 0;
    while (i < 10) {
        if (i % 2 == 0) {
            //do something
        }
        else if (i % 3 == 0) {
            //do something
        }
        else {
            //do something
        }
    }
}

```

*Figure 2.2.1 If/If Else Statement Example*

## 2.3 Switch - Case - Break - Default

**Switch Case Statements** can be used as a substitute for long if statements that involve many integrals. The switch statement is given a value, and the case says that if the parameter has its value then do whatever is after the

colon. A **break** is used to break out of the case statements. A **default** allows a code to execute if the variable does not equal any of the cases (kind of like an **else statement**).

```
int main() {  
  
    ///Simple example  
    int input;  
  
    cin >> input;  
    switch (input) {  
    case 1:  
        playgame();  
        break;  
    case 2:  
        loadgame();  
        break;  
    case 3:  
        exitgame();  
        break;  
    default:  
        cout << "Please enter an input" << endl;  
        break;  
    }  
    cin.get();  
}
```

*Figure 2.3.1 Switch Case Statement example*

Note: MUST be an integral value. The following code is not legal:

```
int main() {  
  
    ///Simple example  
    int a = 10;  
    int b = 10;  
    int c = 50;  
  
    switch (a) {  
    case b:  
        playgame();  
    }
```

```
        break;
    case c:
        loadgame();
        break;

    default:
        cout << "Please enter an input" << endl;
        break;
}
}
```

*Figure 2.3.2 Illegal Switch Case Example*

### 3. Fundamental Data Types

#### 3.1 Char, short, int, long, unsigned char, unsigned short, unsigned int, unsigned long

Data Type	Definition
Char	A single character stored in one byte (8 bits) <b>Char c;</b> <b>C = 'A';</b> C++ uses ASCII for characters
Short	An integral type that is larger than or equal to the size of char, and shorter than or equal to the size of type int. (i.e. between 8-16bits) It is often used when memory needs to be conserved.
Int	Integer: a whole number. Its size is between (16-32 bits).
Long	Guarantees that the integral type is larger than 32 bits (size of an int).
Unsigned Int	Unsigned allows for a greater maximum when using int, short long etc. For example, a regular (signed) int uses 31 bits to represent a number and 1 bit to represent the sign (- or +). This has a range of to . An unsigned int uses all 32 bits for numbers which gives you a range of 0 to .
Unsigned Short	
Unsigned Long	
Unsigned Char	

*Table 3.1.1 Fundamental Data Types*

#### 3.2 Float, double, long double

<b>Data Type</b>	<b>Definition</b>
Float	A real number (i.e. a number with a fractional part (2.33)). It is the smallest floating-point type (4 bytes).
Double	A type double is a floating-point number of size 8 bytes.
Long Double	A long double is a floating-point number of size 12 bytes.

*Table 3.2.1 Fundamental Data Types*

### 3.3 Void

<b>Data Type</b>	<b>Definition</b>
Void	Void is defined as a data type but can be considered as a function return type. It is a keyword that specifies that the function does not return a value.

*Table 3.3.1 Fundamental Data Types*

## 4. Writing Basic Procedural Functions

This topic has to do with skill. So Practice!!

## 5. Dynamic Memory Allocation

Dynamic memory allocation refers to when memory is needed to be determined during runtime (unlike initialising memory in program execution). The memory is allocated using the operator **new**. A pointer allows the program to access the allocated memory. An example is:

```
int *pointer;  
pointer = new int[5];
```

*Figure 5.1 Example on Pointers*

In this example, the `pointer[0]` access the first element in the memory created. `Pointer[1]` access the second element and so on. The dynamic memory requested by our program is allocated in memory heap as stated above. However, since memory is limited it can be exhausted. If memory is exhausted then an exception of type **bad\_alloc** is thrown, causing the program to most likely terminate. A method to avoid this termination is to use a **nothrow**.

```
int *pointer;  
pointer = new (nothrow) int[5];
```

*Figure 5.2 How to avoid pointer errors*

If memory allocation fails, the point returned will be a **null pointer**, allowing the program to continue executing.

Once the memory allocation is no longer needed within a program, it can be freed using the operator **Delete**.

```
delete pointer; // for single elements  
delete pointer[]; // for arrays of elements
```

*Figure 5.3: Deleting pointers*

A **NULL pointer** is a special reserved value that indicates when a pointer is not pointing anywhere.

## 6 Pointers

As explained earlier, variables are located inside specific addresses within computer memory. It is often useful to obtain the address of variable during runtime.

### 6.1 Address-of operator (&)

The address of a variable can be obtained by placing an ampersand (&) before the name of a variable.

```
address = &variable;
```

*Figure 6.1.1 Address of Operator Example*

### 6.2 Dereference operator (\*)

A variable that stores another variable's address is called a **pointer**. They can be used to access the variable they point to directly using a *dereference operator*.

```
int variable = 25;  
address = &variable; //a is the address of the variable (say 1776)  
b = *address; //b is the value of the address = 25
```

*Figure 6.2.1 Dereference Operator Example*

### 6.3 Declaring Pointers

A pointer has different properties when it points to different data types. Once dereference, the type needs to be known. This data type is the type of the value stored in memory that the pointer points to.

```
int main() {  
  
    int v_1, v_2, v_3;  
  
    int *p_1, *p_2;  
  
    p_1 = &v_1;  
    *p_1 = 10;  
  
    p_1 = &v_2;  
    *p_1 = 20;
```



```
p_2 = &v_3;
*p_2 = *p_1;

//v_1 = 10
//v_2 = 20
//v_3 = 20
}
```

*Figure 6.3.1: Declaring Pointer Example*

## 6.4 Pointers and Arrays

The assignment of pointers to arrays allows them to have very similar properties. The name of an array can be used as a pointer to its first element.

```
int main() {

    int array[5];
    int *p;

    p = array; //assignment of pointer to first element of array
    *p = 10; //array[0] = 10

    *p++; *p = 20; //array[1] = 20

    p = &array[2]; *p = 30; //array[2] = 30

    p = array + 3; *p = 40; // array[3] = 40

    p = array; *(p + 4) = 50; //array[4] = 50

    //array[] = [10, 20, 30, 40, 50]
}
```

*Figure 6.4.1 Pointer-Array Example*

## 6.5 Pointer Arithmetic

Only addition and subtraction operations are allowed for pointers and both have slightly different behaviours depending on the size of the data type to which they point.

For example, consider three pointers of different types and the following arithmetic.

```
int main() {

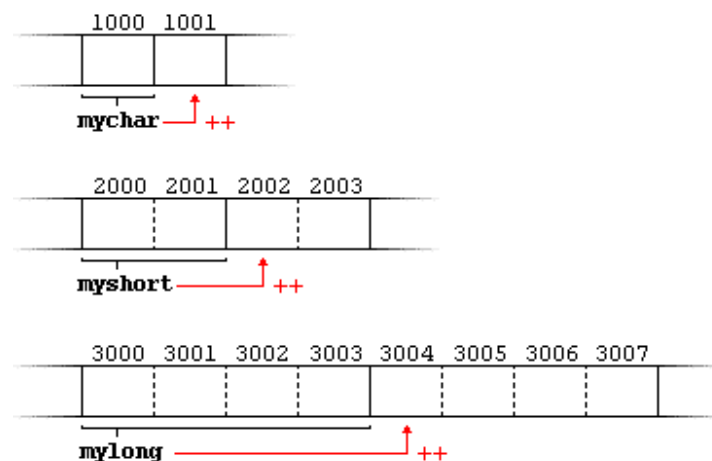
    char *mychar;
    short *myshort;
    long *mylong;

    //Suppose we know that these point to memory location 1000, 2000,
    3000 respectively

    ++mychar; //mychar now points to 1001
    ++myshort; //myshort now points to 2002
    ++mylong; //mylong now points to 3004
}
```

*Figure 6.5.1: Pointer Arithmetic Example*

The reason that the arithmetic occurs in this way is that adding one to a pointer causes the pointer to point to the following element of the same type.



*Figure 6.5.2 Pointer Arithmetic Example 2*

The placement of (++) and (--) is particularly important with pointers and should be read left to right.

```
*p++; //Same as *(p++): increment pointer, and dereference incremented address
*++p; //Same as *(++p): increment pointer, and dereference incremented address
++*p; //Same as ++(*p): dereference pointer, and increment the value it points to
(*p)++; //dereference pointer and post increment the value it points to
```

*Figure 6.5.3 Placement of Operators*

If you don't understand this then please navigate to the following website for more depth

<http://www.cplusplus.com/doc/tutorial/pointers/>

## 6.6 Pointers and const

Sometimes it is important to allow the pointer to be read-only (i.e. the value of the address can not be modified). In this scenario we use the operator **const**.

```
int main() {
    int x;
    int y = 10;
    const int *p = &y;
    x = *p; //Legal operation, reading the value
    *p = x; //Illegal operation
}
```

*Figure 6.6.1: Pointer and const Example*

## 6.7 Pointers and String Literals

Pointers to strings act In a very similar way to pointers and arrays. The pointer points to the beginning of the memory location where the string is located.

```
const char *p = "hello";
```

*Figure 6.7.1: Pointer and String Example*

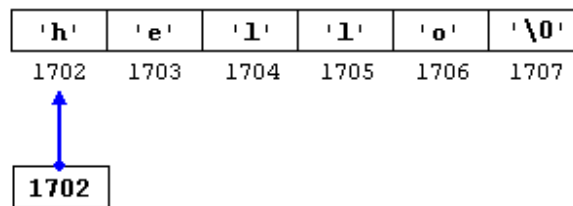


Figure 6.7.2 Pointer to String Explanation

Therefore, the elements of the string can be accessed in the same way that the elements of arrays can be accessed.

```
*(p + 4); // = 0
p[4]; // also = 0
```

Figure 6.7.3: Pointer Arithmetic for String Literals

## 6.8 Void Pointers

**Void** represents the absence of type. Therefore, a void pointer is a pointer to a value that has no type. This gives it flexibility to point to any data type, however the data type still needs to be determined and then the pointer transformed in order to use it.

```
void dosomething(void *p, int size) {
    if (size == sizeof(char)) {
        //do something
    }
    else if (size == sizeof(int)) {
        //do something
    }
}

int main() {

    int a = 1000;
    char b = 'x';

    dosomething(&a, sizeof(a));
    dosomething(&b, sizeof(b));

    return 0;
}
```

*Figure 6.8.1: Void Pointer Example*

## 6.9 Invalid and NULL pointers

Pointers can be *uninitialized* or point to an unknown address out of bounds. However, these pointers can't be dereferenced (access the values) as they will cause undefined behaviour. Despite this, sometimes it is important to set a pointer to nowhere called the nullptr or NULL. This is done in one of three ways:

```
int * p = 0;  
int * p = nullptr;  
int * p = NULL;
```

*Figure 6.9.1: NULL pointer Example*

## 7. Creating User Defined Structures and Classes

### 7.1 Structs, Unions and Classes

#### 7.1.1 Structs (Data Structures)

A data structure is a group of data elements (known as **members**) under the one name (called the **type name**). The members can have different types and lengths. Variables can be declared as **objects** to the struct, allowing them to operate.

```
int main() {  
  
    //Two different ways of declaring objects in structs  
    //Number 1  
    struct product {  
        int weight;  
        double price;  
    };  
  
    product apple;  
    product banana, melon;  
  
    //Number 2  
    struct product {  
        int weight;  
        double price;  
    }; apple, banana, melon;  
  
    //The objects allow the members to be accessed using a (.) operator  
    apple.weight = 1;  
    apple.price = 1.5;  
}
```

*Figure 7.1.1: Example on Data Structures*

#### 7.1.2 Unions

Unions are constructed in a very similar way to that of structs, however unlike structs where each member receives its own allocated memory, members of unions have to share memory. This means that if we set the value

of one member of a union, and then set the value of another member of the union, the value in the first member will be lost. Often unions are used over structs for better memory allocation purposes.

```
int main() {

    //Declare unions in the same way as structs
    union product {
        int weight;
        double price;
    };
    product apple;
    //Since the largest member element is double, the union will occupy 8
bytes

    apple.weight = 1;
    apple.price = 1.5; //apple.weight is now destroyed
}
```

*Figure 7.1.2: Unions Example*

### 7.1.3 Classes

Classes are an expanded version of structs that not only contain data members but also functions. An **object** is an instantiation of a class. They have the following format.

```
class class_name {
    access_specifier_1:
        member1;
    access_specifier_2:
        member 2;
}; object_names;
```

*Figure 7.1.3: Construction of Classes*

## 7.2 Public, Protected and Private Access Attributes

Classes contain **access specifiers** that modify the access rights for members that follow them:

- 1) **private** - accessible only from within other members of the same class (or from **friends**).
- 2) **protected** - accessible from other members of the same class and also members of **derived classes**.
- 3) **public** – accessible from anywhere

```
class class_name {
private:
    member1; //accessed by only other members in the class (or friends)
protected:
    member 2; //accessed by other members an also derived classes
public:
    member 3; //accessible anywhere
}; object_names;
```

*Figure 7.2.1: Access Attribute Example*

## 7.3 Member data and Member Functions

The following rectangle function is a simple example of how to use a combination of member data and functions within a class.

A few key notes about this code:

- The (::) is called a **scope\_operator**
- Members *width* and *height* have only private access

```
class Rectangle {
private:
    int width;
    int height;
public:
    void set_values(int x, int y);
    int area();
};

void Rectangle::set_values(int x, int y) {
    width = x;
    height = y;
}
```



```

int Rectangle::area() {
    return width*height;
}

int main() {
    Rectangle rect;
    rect.set_values(3, 4);
    cout << "area: " << rect.area(); // area: 12
    return 0;
}

```

*Figure 7.3.1: Member Data/Function Example*

## 7.4 Constructor, Copy Constructor, Destructor

### 7.4.1 Constructors

Classes usually contain a special function called a **constructor** which is automatically called whenever a new object of the class is created. This allows the initialisation of member variables and allocation of storage. Without it, accessing a function in the class may return an error if the members of it are not initialised first.

In the rectangle example above, the **set\_values** function can be easily converted into a constructor.

```

class Rectangle {
private:
    int width;
    int height;
public:
    Rectangle(int, int);
    int area() {
        return height*width;
    }
};

Rectangle::Rectangle(int x, int y) {
    width = x;
    height = y;
}

```

```
int main() {
    Rectangle rect(3,4);
    cout << "area: " << rect.area(); // area: 12
    return 0;
}
```

*Figure 7.4.1: Constructor Example*

#### 7.4.2 Overloaded Constructors

An overloaded constructor is a different version to other constructors, with different parameters and types.

```
class Rectangle {
private:
    int width;
    int height;
public:
    Rectangle();
    Rectangle(int, int);
    int area() {
        return height*width;
    }
};

Rectangle::Rectangle() {
    width = 10;
    height = 10;
}

Rectangle::Rectangle(int x, int y) {
    width = x;
    height = y;
}

int main() {
    Rectangle rect(3,4);
    Rectangle rect_2();
    cout << "area: " << rect.area(); // area: 12
    cout << "area: " << rect_2.area(); //area: 100
    return 0;
}
```

```
}

```

*Figure 7.4.2: Overloaded Constructor Example*

As you can see there is now two constructors. **Rectangle()** is the **default constructor** whilst **Rectangle(int, int)** is the overloaded constructor.

### 7.4.3 Copy Constructors

A copy constructor is a member function that initializes an object using another object of the same class. It has the form

```
ClassName (const ClassName &old_obj);
```

```
class Rectangle {
private:
    int width;
    int height;
public:
    Rectangle();
    Rectangle(int, int);
    //Copy Constructor
    Rectangle(const Rectangle &rect);
    int area() {
        return height*width;
    }
};

Rectangle::Rectangle() {
    width = 10;
    height = 10;
}

Rectangle::Rectangle(int x, int y) {
    width = x;
    height = y;
}

Rectangle::Rectangle(const Rectangle &rect) {
    width = rect.width + 10;
}
```

```

        height = rect.height + 10;
    }

    int main() {
        Rectangle rect(3,4);
        Rectangle rect2 = rect; //copy constructor called here
        cout << "area: " << rect.area(); // area: 12
        cout << "area: " << rect2.area(); //area: 182
        return 0;
    }

```

*Figure 7.4.3: Copy Constructor Example*

#### 7.4.4 Destructor

Destructor functions are called when objects are destroyed (deallocated). A destructor is designated by placing the class name with a tilde (~). For example Rectangle has the destructor ~Rectangle(). The destructor uses a delete operator to deallocate the space dynamically allocate for text storage.

### 7.5 Virtual, Friend, Operator and Pure Virtual Functions

#### 7.5.1 Friend Functions and Classes

As talked about previously, private and protected members in classes can be accessed outside the class by friend functions or classes. The following is a simple example of how a **friend function** can be utilised.

```

class Rectangle {
private:
    int width;
    int height;
public:
    Rectangle();
    Rectangle(int ,int);
    int area() {
        return height*width;
    }
    friend Rectangle duplicate(const Rectangle&); //declaration of friend
function
};

```

```

Rectangle::Rectangle() {

}

Rectangle::Rectangle(int x, int y) {
    width = x;
    height = y;
}

Rectangle duplicate(const Rectangle& p) {
    Rectangle res;
    res.width = p.width * 2;
    res.height = p.height * 2;
    return res;
}

int main() {
    Rectangle rect(3,4);
    Rectangle rect2;
    rect2 = duplicate(rect);
    cout << "area: " << rect.area(); // area: 12
    cout << "area: " << rect2.area(); //area: 48
    return 0;
}

```

*Figure 7.5.1 Friend Function Example*

Notice how the function **duplicate** has not been declared a member of **Rectangle** but is able to access its **private** members.

A friend class is a class that have access to its private and protected member functions. A class is declared a friend of another before its member functions are defined.

```

class Square {
    friend class Rectangle;
private:

```

```
        int side;
public:
    Square(int a) { side = a; }
};

class Rectangle {
private:
    int width;
    int height;
public:
    Rectangle();
    Rectangle(int width, int height);
    int area() {
        return height*width;
    }
    void convert(Square a);
};

Rectangle::Rectangle() {
    // have default parameters or don't have a default constructor
    Eg.
    width = 1;
    height = 1;
}

Rectangle::Rectangle(int x, int y) {
    width = x;
    height = y;
}

void Rectangle::convert(Square a) {
    width = a.side;
    height = a.side;
}

int main() {
    Rectangle rect;
    Square sqr(4);
    rect.convert(sqr);
    cout << "area: " << rect.area(); // area: 16
    return 0;
}
```

*Figure 7.5.2: Friend Class Example*

### 7.5.3 Inheritance

Classes (called **derived classes**) can *inherit* the other members of another class (called the **base class**) whilst also adding their own members. The inheritance relationship is declared through the following format:

```
class derived_class_name: public base_class_name
```

A publicly derived class inherits access to every member of the base class except:

- its constructor and destructor
- assignment operator members
- friends
- private members

```
class circle {
protected:
    double radius, area;
public:
    circle() {}
    void set_values(double r) {
        radius = r;
        area = 3.14*radius*radius;
    }
};

class cylinder : public circle {
protected:
    double length;
public:
    cylinder(double l) {
        length = l;
    }
    double volume() {
        return area*length;
    }
};

int main() {
    cylinder cyl(20);
    cyl.set_values(10);
}
```

```

    cout << cyl.volume() << endl; //Output: 6280
    return 0;
}

```

*Figure 7.5.3: Inheritance Example*

A class may inherit from multiple base classes by separating the base classes by commas.

```

class cylinder : public Circle, public SurfaceArea {};

```

*Figure 7.5.4: Multiple Inheritance Example*

#### 7.5.4 Virtual Members

The virtual keyword allows a member of a derived class to be redefined with the same name as one in the base class, whilst still preserving its calling properties. A class that declares or inherits a virtual function is called a **polymorphic class**.

```

class Polygon {
protected:
    int width, height;
public:
    void set_values(int a, int b)
    {
        width = a; height = b;
    }
    virtual int area() {
        return 0;
    }
};

class Rectangle : public Polygon {
public:
    int area()
    {
        return width*height;
    }
};

class Triangle : public Polygon {
public:

```



```

    int area()
    {
        return width*height / 2;
    }
};

int main() {
    Rectangle rect;
    Triangle trgl;
    Polygon poly;
    Polygon * ppoly1 = &rect;
    Polygon * ppoly2 = &trgl;
    Polygon * ppoly3 = &poly;
    ppoly1->set_values(4, 5);
    ppoly2->set_values(4, 5);
    ppoly3->set_values(4, 5);
    cout << rect.area() << '\n'; //20
    cout << trgl.area() << '\n'; //10
    cout << poly.area() << '\n'; //0
    return 0;
}

```

*Figure 7.5.5 Virtual Member Example*

#### 7.5.5 Pure Virtual Functions

Pure virtual functions are virtual members without definition. Their definition is instead replaced by (=0). Classes that contain at least one pure virtual function are known as **abstract base classes**. As seen below, the abstract classes can no longer be used to instantiate objects but instead used to create pointers to it and use its polymorphic abilities.

```

class Polygon {
protected:
    int width, height;
public:
    void set_values(int a, int b)
    {
        width = a; height = b;
    }
    virtual int area() = 0 {
        return 0;
    }
}

```

```

};

class Rectangle : public Polygon {
public:
    int area()
    {
        return width*height;
    }
};

class Triangle : public Polygon {
public:
    int area()
    {
        return width*height / 2;
    }
};

int main() {
    Rectangle rect;
    Triangle trgl;
    //Polygon poly; CAN NO LONGER INSTANTIATE OBJECTS IN ABSTRACT CLASSES
    Polygon * ppoly1 = &rect;
    Polygon * ppoly2 = &trgl;
    //Polygon * ppoly3 = &poly;
    ppoly1->set_values(4, 5);
    ppoly2->set_values(4, 5);
    //ppoly3->set_values(4, 5);
    cout << rect.area() << '\n'; //20
    cout << trgl.area() << '\n'; //10
    //cout << poly.area() << '\n'; //0
    return 0;
}

```

*Figure 7.5.6: Pure Virtual Functions Example*

### 7.5.6 Overloading Operators

#### *As member functions*

Overloading operators refers to redefining the built in operators available in C++. Below is an example of how the (+) operator in a class can be overloaded so that two objects of type 'Mark' can be added together.

```
class Marks {
    int internal_mark;
    int external_mark;
public:
    Marks() {
        internal_mark = 0;
        external_mark = 0;
    }
    Marks(int im, int em) {
        internal_mark = im;
        external_mark = em;
    }

    void display() {
        cout << internal_mark << ", " << external_mark << endl;
    }

    //Overloading the (+) operator
    Marks operator+(Marks m) {
        Marks temp;
        temp.internal_mark = internal_mark + m.internal_mark;
        temp.external_mark = external_mark + m.external_mark;
        return temp;
    }
};

int main() {
    Marks m1(10, 20), m2(30, 40);
    Marks m3 = m1 + m2;
    m3.display(); // Output: 40, 60
    return 0;
}
```

*Figure 7.5.7: Overloading Operator (Member)*

As you can see, the overloading has occurred inside the class a member function. This is a very useful tool, but difficult to explain in words alone. If unsure, watch this YouTube video as it walks through how to create an overloaded operator.

<https://www.youtube.com/watch?v=tFYRTWFXSgY>

## Overloadable/Non-overloadableOperators

Following is the list of operators which can be overloaded –

+	-	*	/	%	^
&		~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	
+=	-=	/=	%=	^=	&=
=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Following is the list of operators, which can not be overloaded –

::	.*	.	?:
----	----	---	----

Figure 7.5.7: Overloadable/Non-Overloadable Operators

Sometimes it's useful to define a function outside a class. When doing this we must use the scope resolution operator (::). The below example works in the same way as the previous (Marks) example but uses the overload as a non-member.

```
class Marks {
    int internal_mark;
    int external_mark;
public:
    Marks() {
        internal_mark = 0;
        external_mark = 0;
    }
    Marks(int im, int em) {
        internal_mark = im;
        external_mark = em;
    }

    void display() {
        cout << internal_mark << ", " << external_mark << endl;
    }

    Marks operator+(Marks m); //Must include the operator+ definition in
the class
};
```

```

Marks Marks::operator+(Marks m) {
    Marks temp;
    temp.internal_mark = internal_mark + m.internal_mark;
    temp.external_mark = external_mark + m.external_mark;
    return temp;
}

int main() {
    Marks m1(10, 20), m2(30, 40);
    Marks m3 = m1 + m2;
    m3.display(); // Output: 40, 60
    return 0;
}

```

*Figure 7.5.8: Overloading Operator*

*Use of friend functions (non-member functions)*

Using friend whilst overloading an operator in a non-member function allows the access of private data members. The example below demonstrates this use.

```

class Marks {
    int internal_mark;
    int external_mark;
public:
    Marks() {
        internal_mark = 0;
        external_mark = 0;
    }
    Marks(int im, int em) {
        internal_mark = im;
        external_mark = em;
    }

    void display() {
        cout << internal_mark << ", " << external_mark << endl;
    }

    friend void operator+=(Marks &m , int bonus); //Must include the
operator+ definition in the class

```

```
};

void operator+=(Marks &m, int bonus) {
    m.internal_mark = m.internal_mark + bonus;
    m.external_mark = m.external_mark + bonus;
}

int main() {
    Marks m1(10, 20);
    m1 += 20;
    m1.display(); // Output: 30, 40
    return 0;
}
```

*Figure 7.5.9 Overloading Operators (using friend functions)*

## 7.6 Static Member Data and Static Member Functions

### 7.6.1 Static Member Data

I find the best way to understand the difference between static member data and normal variables is to look at their differences.

Name	Static Data Member	Normal Member Data
Declaration	static int a;	int a;
Initialisation	Initialises to zero when first object of class is created	Initialises to any garbage value
Description	Only one copy is created and it is available for the entire program	Available till the end of block

*Table 7.6.1: Difference Between Static and Normal Member Data*

```
void counter() {  
    static int count;  
    cout << count++ << endl;  
}  
  
int main() {  
    for (int i = 0; i < 3; i++) {  
        counter();  
    }  
}  
  
//Output  
//0  
//1  
//2
```

```
void counter() {  
    int count = 0;  
    cout << count++ << endl;  
}  
  
int main() {  
    for (int i = 0; i < 3; i++) {  
        counter();  
    }  
}  
  
//Output  
//0  
//0  
//0
```

*Figure 7.6.1 Difference Between Static and Normal Member Data*

Consider the two examples above. Since static is always initialized to zero and created once, it will continue to increment. However for the normal member data, it will be created and deleted every time that the function counter() is called. Therefore it will never increment.

### 7.6.2 Static Member Functions

A static function can be created in a similar way and has the following properties:

- Can only access static variables and static functions
- Can be called using the class name.

```
class demo {
    static int counter; //static variable
public:
    demo() {}
    void increase_count() {
        counter++;
    }
    //Static function which shows static variable
    static void showcounter() {
        cout << "Count = " << counter << endl;
    }
};

int demo::counter; //Must write this, defines the counter as a static
variable of the class

int main() {
    demo p;
    p.increase_count();
    p.increase_count();
    demo::showcounter(); //Calling the static variable using the class
name
}

//Output: Count = 2
```

*Figure 7.6.2: Implementation of Static Member Functions Into a Class*



## 8. Developing Class Hierarchies

### 8.1 Use of Virtual Functions

Please refer to 7.5

### 8.2 Use of Virtual Destructors

When deleting a base class pointer when there is no virtual destructor, an **undefined behaviour** will result. Whenever a base class is meant to be manipulated polymorphically, it is best to make the base class's destructor virtual. The following example explains the difference.

Without a virtual destructor

```
class base {
public:
    base() {
        cout << "base constructor called" << endl;
    }
    ~base() {
        cout << "base destructor called" << endl;
    }
};

class derived : public base {
public:
    derived() {
        cout << "derived constructor called" << endl;
    }
    ~derived() {
        cout << "derived destructor called" << endl;
    }
};

int main() {
    base *b = new derived;
    delete b;
}
```

*Figure 8.2.1: Without Virtual Destructor*

The output of this function is:

Base constructor called  
Derived constructor called  
Base destructor called

With a virtual destructor

```
class base {
public:
    base() {
        cout << "base constructor called" << endl;
    }
    virtual ~base() {
        cout << "base destructor called" << endl;
    }
};

class derived : public base {
public:
    derived() {
        cout << "derived constructor called" << endl;
    }
    ~derived() {
        cout << "derived destructor called" << endl;
    }
};

int main() {
    base *b = new derived;
    delete b;
}
```

*Figure 8.2.2: With a Virtual Destructor*

The output of this function is:

Base constructor called  
Derived constructor called  
Derived destructor called  
Base destructor called

### 8.3 Use of Abstract Classes

Please refer to 7.5.5

## 9. Pass Through objects

### 9.1 Pass by reference

Pass-by-reference means to pass the address of a variable instead of making a copy of it. This allows functions to modify the value and also avoid making copies of an object that might hinder performance.

```
void swap(int &i, int &j) {  
    int temp = i;  
    i = j;  
    j = temp;  
}  
  
int main() {  
    int a = 10;  
    int b = 20;  
  
    swap(a, b);  
    cout << "A = " << a << " and B = " << b << endl;  
    return 0;  
}  
  
//output: A = 20 and B = 10
```

*Figure 9.1.1: Pass by Reference Example*

If we do not want to modify the object but would still like to avoid copying the object, then we can use const.

### 9.2 Pass by Value

When you pass by value, you pass a copy of the object, and therefore modification of this 'clone' will not affect the original object.

```
void swap(int i, int j) {  
    int temp = i;  
    i = j;  
    j = temp;  
}  
  
int main() {  
    int a = 10;  
    int b = 20;
```

```

    swap(a, b);
    cout << "A = " << a << " and B = " << b << endl;
    return 0;
}

//output: A = 10 and B = 20

```

*Figure 9.2.1: Pass by Value Example*

### 9.3 Return by Reference

When a function returns by reference, it returns a pointer to its return value. This allows a function to be used on the left side of an assignment.

```

int num; //Global variable

int& test() {
    return num;
}

int main() {
    test() = 5; //test() returns num, therefore this is actually num = 5
    cout << num << endl;
    return 0;
}

//Output: 5

```

*Figure 9.3.1: Return by Reference Example*

Note: Do not return a local variable by reference

### 9.4 Return by Value

Return by value is much safer than return by reference. When returned by value a copy of that value is returned to the caller. Unlike return by reference, it can return variables that involve local variables declared within the function.

```

int test(int num) {
    return num;
}

```

```
}

int main() {
    int num = test(5); //test() returns 5, therefore this is actually num
= 5
    cout << num << endl;
    return 0;
}

//Output: 5
```

*Figure 9.4.1:: Return by Value Example*

## 10. Class Derivation and Inheritance

Please refer to 7.5.3

## 11. Abstract Classes. How to Distinguish between abstract classes and non-abstract classes.

Please refer to 7.5.4 and 7.5.5

## 12. Overloading

### 12.1 Function Overloading

You can have multiple definitions of the same function as long as they differ from each other by the types and/or arguments in their list. This works in a similar way to overloading constructors (7.4.2)

```
class printData {
public:
    printData() {}
    void print(int i) {
        cout << "Printing int " << i << endl;
    }
    void print(double f) {
        cout << "Printing double " << f << endl;
    }
    void print(char *c) {
        cout << "Printing character " << c << endl;
    }
};

int main() {
    printData p;

    p.print(5);
    p.print(500.22);
    p.print("Hello World");
}

//Output
//Printing int 5
//Printing double 500.22
//Printing character Hello World
```

*Figure 12.1.1 Function Overloading Example*

### 12.2 Operator Overloading

Please refer to 7.5.6

## 13. Function Templates and Class Templates

Function and class templates are special functions/classes that can operate with generic types (i.e. adapt to more than one type or class without repeating the entire code for each type). We call this generic programming.

### 13.1 Syntax of Templates

### 13.2 Use of Templates

#### *Function Templates*

The general forms of a function template is:

```
template <class identifier> function_declaration  
template <typename identifier> function_declaration
```

Note: The keywords class and typename have the same meaning and can be used interchangeably.

You have to write the template above the function.

```
template <class T>  
  
void addNumbers(T a, T b) {  
    cout << "Addition is = " << a + b << endl;  
}  
  
int main() {  
    int a, b;  
    a = 10;  
    b = 10;  
    addNumbers(a, b);  
    return 0;  
}  
  
//Output: Addition is = 20
```

*Figure 13.1.1 Example of Simple Template*

In the above example, 'T' acts as the data type and will therefore adjust to whatever data type is passed through the function addNumbers.



This works for single data types, however if we are using multiple data types like (int and double) then we must define multiple parameters

```
template <class_first, class_second>
```

```
template <class T, class P>

void addNumbers(T a, P b) {
    cout << "Addition is = " << a + b << endl;
}

int main() {
    int a;
    double b;
    a = 10;
    b = 10.56;
    addNumbers(a, b);
    return 0;
}

//Output: Addition is = 20.56
```

*Figure 13.1.2 Example of Multiple Template*

### *Class Templates*

Class templates differ from function templates in that the compiler cannot deduce the template parameter type(s) for a class template. We instead have to tell the data type that we will be using.

Important things to note:

- Must redefine the template every time a function is declared outside the class
- Must tell the function (outside the class) what parameter you will be passing through it (i.e. “Math<T>::addnumber()”)
- Must tell the class what data type you are using (i.e. “ Math<int> obj(10,20)”)

```
template <class T>

class Math {
private:
    T first, second;
public:
    Math() {}
```

```

    Math(T a, T b) { //Parameter type is T because we want it to be
generic
        first = a;
        second = b;
    }
    void addNumbers();
    void multiply();
};

template <class T> //Have to redefine template every time a function uses
it
void Math<T>::addNumbers() { //Have to define the type <T>
    cout << "Addition of two numbers is " << first + second << endl;
}

template <class T> //Have to redefine template every time a function uses
it
void Math<T>::multiply() { //Have to define the type <T>
    cout << "Multiplication is " << first*second << endl;
}

int main() {
    Math<int> obj(10, 20); //Have to tell the class what data type you
are using i.e. <int>
    obj.addNumbers();
    obj.multiply();
    return 0;
}
//Output
//Addition of two numbers is 30
//Multiplication is 200

```

*Figure 13.1.3 Class Template Example*

### 13.3 Container Class Vector

Vector is the most useful class in stl (Standard Template Library). Vectors are kind of similar to arrays but without the limitations that are attached to arrays. You can insert or delete an element from vector because it can change size. An array is a data structure, whilst a vector is a class, allowing it to have built in functions.

Note: To use vector we must include the header file <vector>

The syntax is

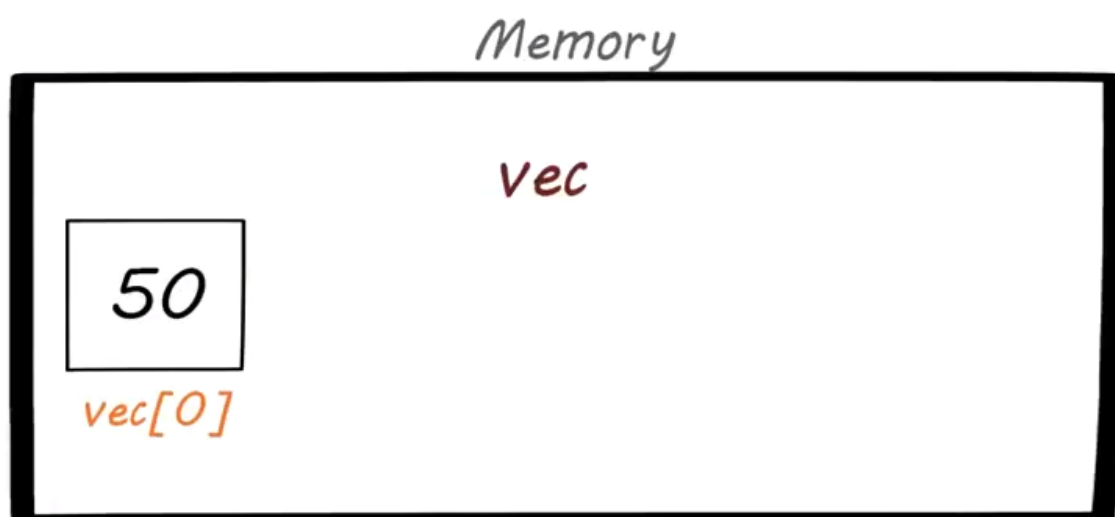
```
Vector <data_type> vector_name
```

E.g.

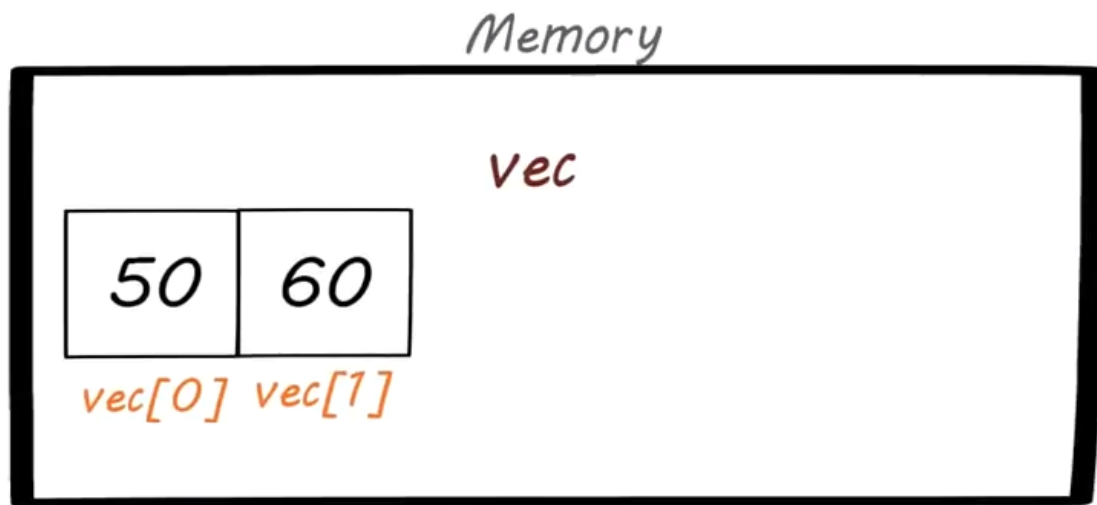
```
Vector <int> vec; //A vector called vec which will store int values
```

If we want to insert an element into it we must use the `push_back()` function.

```
vec.push_back(50);
```



```
vec.push_back(60);
```



If we want to remove an element we must use the `pop_back()` function.

```
vec.pop_back();
```

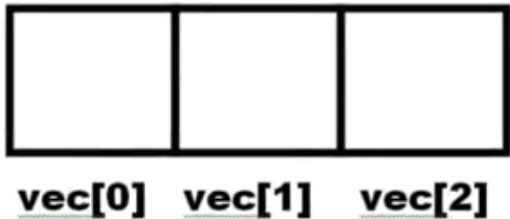
If we want to know the size of our vector then we should use `vec.size()`.

Since vector is a class in the STL it has many different already defined functions. If you would like to know more about these then navigate to <http://www.cplusplus.com/reference/vector/vector/>

And watch the following video

<https://www.youtube.com/watch?v=2b9Mdh3L5Qs>

Below is a table of a few important methods of creating vectors (i.e their constructors)

Method	Description
<code>vector &lt;int&gt; vec(3);</code>	Creates a three element vector 
<code>vector &lt;int&gt; vec{ 10, 20, 30 };</code>	Create a vector with these values.

<code>vector&lt;int&gt; vec(3,0);</code>	<p>Create a vector with 3 elements and put 0 in each element.</p>
<code>vector&lt;char&gt; vec(3,'a');</code>	<p>Creates a vector with 3 elements with 'a' in it.</p>

*Table 13.3.1: Ways of Creating Vectors*

### 13.4 Iterating Through Vector of Objects

### 13.5 Accessing Object Data Using Iterators

Iterators are a pointer like object in STL that is able to be incremented with (++), dereferenced with (\*) and compared against another iterator with (!=). Iterators allow you to get elements from and store elements to various **containers**.

The syntax for iterators is:

`class_name<template_parameters>::iterator name`

Where the name is the name of the iterator variable you wish to create and the class\_name is the name of the STL container.

For example, if you were passing an iterator through a vector of ints the syntax would be:

```
vector<int>::iterator iter
```

Below is an example on how an iterator can be used to show the values of a vector.

```
void printOut(vector<int> x) {
    vector<int>::iterator pos;
    for (pos = x.begin(); pos != x.end(); pos++) {
        cout << *pos << endl;
    }
}

int main() {
    vector<int> vec;

    //fill the vector with values
    for (int i = 0; i < 10; i++) {
        vec.push_back(3 * (i + 1));
    }

    //print out the value
    printOut(vec);

    cin.get();

    return 0;
}

//Output:
//3
//6
//9
//12
//15
//18
//21
//24
//27
```



```
//30
```

*Figure 13.5.1: Example of Iterator use in Vectors*

## 14. Type Casting

### 14.1 Ordinary Type Casting

#### *Implicit Conversion*

Implicit conversion do not require any operator but are performed when a value is copied to a compatible type. For example:

```
int main() {  
    short a = 2000;  
    int b;  
    b = a;  
    cout << b << endl; //Output: 2000  
}
```

*Figure 14.1.1: Example of Implicit Conversion*

#### *Explicit Conversion*

Explicit conversion uses an operator in order to explicitly convert two data types. For example:

```
int main() {  
    short a = 2000;  
    int b;  
    b = (int) a; //c-like cast notation  
    b = int(a); //functional notation  
    cout << b << endl; //Output: 2000  
}
```

*Figure 14.1.2: Example of Explicit Conversion*

### 14.2 Dynamic Casting

`dynamic_cast` can only be used with pointers and references to objects. It ensures that the result of the type conversion is a valid complete object of the requested class. It is always successful when we cast a class to one of its base classes (i.e. can only go through public inheritance).

The `dynamic_cast` does a check before it does the conversion, and if successful it will convert the classes. If unsuccessful it will return a NULL pointer. This is more type safe.



```
class Parent {
protected:
    int x, y;
public:
    void SetPosition(int x, int y) { this->x = x; this->y = y; }
    void Display() { cout << x << " " << y << endl; }
};

class Child : public Parent {

};

int main() {
    Child c;
    Parent *p;
    //p = &c; THIS MAY CAUSE ERRORS IF UNSUCCESSFUL
    p = dynamic_cast<Parent*>(&c); //Will not cause errors if
    unsuccessful conversion
    return 0;
}
```

*Figure 14.2.1: Dynamic\_cast Example*

# 15. Streams

## 15.1 How Data Are Packed as Bytes

Data is packed in groups of 8 bytes (for 64-bit machine) or 4 bytes (for a 32-bit machine).

## 15.2 How to Alter Data Packing in Memory

In order to alter data packing in memory we must use pragma pack. It is given the syntax:

```
#pragma pack(number)
```

Where number specifies how many groups it should be packed into.

Often the compiler will choose to lay out memory with padding between the members. The alignment generally allows a faster access of the memory (and sometimes non alignment is not possible). This is often seen in data structures.

```
struct Test {
    char AA;
    int BB;
    char CC;
};
```

Figure 15.2.1: Example Struct

Different compilers will pack data differently. The data packing could look like this.

1	2	3	4
AA(1)	pad	pad	pad
BB(1)	BB(2)	BB(3)	BB(4)
CC(1)	pad	pad	pad

Table 15.2.1: Data Packing of Simple Struct

As you can see, the size of the data structure is 12 bytes (4 x 3). However, it only contains 6 bytes of data with the rest being padding. This is where `#pragma` can come into use, in order to pack the data in ways that maximises memory. For instance, `#pragma pack(1)` will look like this:

1
AA(1)
BB(1)
BB(2)
BB(3)
BB(4)
CC(1)

*Table 15.2.2: Data Packing #pragma pack(1)*

As you can see the size of the data structure is now 6 bytes, and memory has been conserved.

With `#pragma pack(2)` the struct would look like:

1	2
AA(1)	pad
BB(1)	BB(2)
BB(3)	BB(4)
CC(1)	pad

*Table 15.2.3 Data Packing #pragma pack(2)*

The size of the data structure will now be 8 bytes.

### 15.3 Instantiating Stream Objects for Input and Output

In order to understand this topic, we must first understand what ‘streams’ are. A stream is something that allows you to send or received an unknown number of bytes. A common metaphor used to explain them is:

*A stream of water. You take the data as it comes, or sent it as it's needed.*

If you compare this to an array, that is limited by its fixed and known length. Streams work with built-in data types and you can also make your own user-defined types by working with streams by overloading the operator (<<) to write objects into streams, or the (>>) operator to read objects from streams.

You have probably been using streams without even realising it (#include <iostream>). The most basic stream types are the standard input/output streams:

istream cin - built-in input stream variable, by default hooked to keyboard

ostream cout - built-in output stream variable, by default hooked to console

Important Note: Always include the pre-compiler #include <iostream>

*Output: The Insertion Operator (<<)*

In order to get information out of a file or a program, we need to explicitly instruct the computer using the insertion operator (<<). For example, to output to a screen we use the statement:

```
cout << "X = " << X;
```

*Input: The Extraction Operator (>>)*

Similarly to the insertion operator, to get information into a file or a program we need to explicitly instruct the computer using the extraction operator (>>).

```
cin >> X;
```

## 15.4 Reading and Writing Object Data to an ASCII file

When reading and writing to files stored on disk, we must use the pre-compiler:

```
#include <fstream>
```

There are no pre-defined file stream variables therefore we must declare them

```
ifstream inFile; //input file stream object
ofstream outFile; //output file stream object
```

In order to establish a connection between the file stream and a file we may use open().

```
inFile.open("readme.data");
outFile.open("writeme.data");
```

If the file does not exist then the input stream variable will contain an error flag. For the output stream, if the file does not exist it will create one on the OS. We can also compact the above two equations together to make:

```
ifstream inFile("readme.data");
ofstream outFile("writeme.data");
```

When we are done with the file we must close using:

```
inFile.close();
outFile.close();
```

Some useful formatting tools:

Manipulator	Description
setw()	Sets the field width (number of spaces in which the value is displayed)
setprecision() Must also include (ostream << fixed << showpoint) Outstream is usually cout	Sets the precision, the number of digits shown after the decimal point.
ostream << left	Will justify the alignment of data to the left
ostream << right	Will justify the alignment of data to the right (with padding on the left).
setfill()	Allows you to fill unused space

*Table 15.4.1: Manipulators*

Note: Make sure that you include the header <iomanip> and if using strings <string>.

The below example combines all these concepts we just learned

```
#include <fstream>
#include <iomanip>
#include <string>

using namespace std;

int main() {

    ofstream oFile("AreaData.txt");
    oFile << fixed << showpoint;
    const double PI = 3.14159265;
    string figName = "Ellipse";
    int majorAxis = 10;
    int minorAxis = 2;
    double Area = PI*majorAxis*minorAxis;

    oFile << setw(20) << "Area" << endl;
    oFile << left << setw(10) << figName;
    oFile << right << setw(10) << setprecision(4) << Area << endl;

    oFile.close();
    return 0;
}
```

*Figure 14.5.1: Manipulators Example*

The output is a file (AreaData.txt) that looks like this:

```

                Area
Ellipse      62.8319
```

Some useful member functions:

Member Functions	Description
getline() getline(file, string_name)	Provides a way to read character input into a string variable
fail()	Provides a way to check the status of the last operation. Returns true if the last operator failed and false if successful.

<code>clear()</code>	Provides a way to reset the status flags of an input stream after an input failure has occurred.
<code>eof()</code>	Returns true if the last input operation attempted to read the end-of-file mark, and false otherwise
<code>get()</code> <code>*get(someChar)</code>	Is a member function of cin and will return the next single character in the stream (whether it is a whitespace or not). It will also remove the character from cin and place it into a character.

*Table 15.4.2: Member Functions*

## 15.5 Reading and Writing Object Data To Binary

### *Writing Object Data to Binary File*

1. Include <fstream> header file.
2. Create an object of the ofstream class and open this using the binary flag.
3. Write the bytes of the variable or object of the ofstream object

```
int main() {

    const float f = 3.14f;
    ofstream ofile("binary.bin", ios::binary);
    ofile.write((char*)&f, sizeof(float));
    ofile.close();
    return 0;
}
```

*Figure 15.5.1 Writing Object to Binary File*

In this course, it is expected that we know how to overload the operator (<<) in order to write data to a Binary File. First you will define a friend function in the class definition that contains the data. In this case the class is Drive.

```
friend ofstream& operator<<(ofstream& os, const Drive& d);
```

```
ofstream& operator<<(ofstream& os, Drive d) {
    unsigned char * BytePtr = (unsigned char*)&d.D; // (a)
```

```

    for (int i = 0; i < sizeof(d.D); i++) { //(b)
        os.put(*(BytePtr + i));
    }
    return os;
}

```

*Figure 15.5.2: Overloading << operator to write to binary file*

What is happening:

- (a) A pointer is made that points to the first element in the struct of the data stored in Data struct D.
- (b) Iterators through each BYTE of the Drive class and puts the data stored from the data structure into a binary file using `os.put(*(BytePtr + i));`

We can then use the operator in the main to write to a Binary file.

#### *Reading Object Data to Binary File*

1. Include <fstream> header file
2. Create an object of the ifstream class and open using the binary flag.
3. Read the bytes of the variable or object of the ifstream to a variable.

```

int main() {

    float f;
    ifstream ofile("binary.bin", ios::binary);
    ofile.read((char*)&f, sizeof(float));
    cout << f << endl; //Output: 3.14
    ofile.close();
    return 0;
}

```

*Figure 15.5.3 Reading Object to Binary File*

Again in this course they would like you to know how to overload the (>>) operator in order to read a binary file. It is very similar to writing to a binary file, but we obviously use ifstream instead of ofstream, including its different member functions.

```

friend ifstream& operator>>(ifstream& is, Drive d);

```

```

ifstream& operator>>(ifstream& is, Drive d) {
    unsigned char * BytePtr = (unsigned char*)&d.D; //(a)
    for (int i = 0; i < sizeof(d.D); i++) { //(b)
        *(BytePtr + i) = is.get();
    }
}

```



```

    }
    return is;
}

```

*Figure 15.5.4 Overloading >> operator to read Binary File*

What is happening:

- (a) A pointer is made that points to the first element in the struct of the data stored in Data struct D.
- (b) Iterators through each BYTE of the Drive class and extracts the data from a binary file using `*(BytePtr + i) = is.get();`

## 15.6 How to Randomly Access Data in Binary File

- No extra operator overloads are needed

```

#define OBJECT_NO 5

int main() {
    int j = 0;
    Storage S;

    ifstream is("BinaryData.dat", ios::binary);
    is.seekg(OBJECT_NO * sizeof(Data));

    is >> S;

    cout << S << endl;

    is.close();
    return 0;
}

```

- In a binary file there are always a fixed number of bytes, thus we don't have to loop through and find the data, but we can skip through it
- **OBJECT\_NO\*sizeof(Data)** allows to skip through by `5*sizeof(Data)` bytes

## Question 1 - Example

A robotic manipulator is controlled by controlling its individual joints, one joint at a time. The joint control algorithms were developed by the robot manufacture and are given in a file named `asea1230.lib` which is a library file. The header file `asea1230.h` is as follows:

```
#ifndef ASEA1230_H
#define ASEA1230_H
int DriveJoint1(int absolutePosition);
int DriveJoint2(int absolutePosition);
int DriveJoint3(int absolutePosition);
int DriveJoint4(int absolutePosition);
int DriveJoint5(int absolutePosition);
int DriveJoint6(int absolutePosition);
#endif // ASEA1230_H
```

Each of the functions in the header file are written to drive a particular joint. For example, `DriveJoint1()` takes in the encoder count values specified in and drives the Joint Number 1 to the specified position. The return value indicates the joint positioning status by returning 0 when complete and 1 when still in motion.

A production facility uses this robot to carry out a certain task. The commands to the task are in a binary file which contains the joint number as an int and the position it need to be positioned to also specified as an int, stored in the binary file in that order. The number of moves to be carried out are determined by the length of the file and the length of the file varies from task to task. Hence the number of data records are usually not specified.

1. Develop a data structure to represent the data to be read from the file representing a single joint motion. (2 marks).

This questions is quite simple. In the question you can identify two variables, the joint number and its position. Both of these variables are integers, therefore the data structure will look like so.

```
struct Data {
    int jointNumber;
    int pos;
};
```

The question stated “contains joint number... and the position...stored in binary file in that order”

Therefore it is important that the joint number is the first variable and the position is the second (otherwise they will be mixed up).

2. Develop a class to deal with data extraction from the file. (4 marks)

The Data we need to extract from the data structure is the jointNumber and pos. These are in binary form and therefore we have to read them byte by byte. To do this we will create a simple class that uses getters to access the data from the data structure. There is nothing too special about this so i won't go into detail.

```
#include <iostream>

using namespace std;

struct Data {
    int jointNumber;
    int pos;
};

class Drive {
private:
    Data D;
public:
    Drive();
    int getPosition();
    int getJointNumber();
    ~Drive();
};

Drive::Drive() {

}

int Drive::getPosition() {
    return D.pos;
}

int Drive::getJointNumber() {
    return D.jointNumber;
}

Drive::~Drive() {

}
```

3. Overload the stream extraction operator '>>' to read one binary data record from the file. (4 marks).

This part of the question is directly related to 15.5 - Reading Object Data to Binary File.

We begin by creating a friend function in the class Drive that overloads the operator (>>).

```
friend ifstream& operator>>(ifstream& is, Drive& d);
```

The function is declared outside the class and looks like this.

```
ifstream& operator>>(ifstream& is, Drive& d) {
    unsigned char * BytePtr = (unsigned char*)&d.D;
    for (int i = 0; i < sizeof(d.D); i++) {
        *(BytePtr + i) = is.get();
    }
    return is;
}
```

It creates a pointer (BytePtr) that points towards the first element in the struct D. It then iterates through each Byte of data and use the get() function in order to extract the binary data and store their values into the Struct Data we made. In the question i told us the the

4. Show how function pointers can be used to drive the joints as specified by the data in the file until the end of the file. (6 marks)

The first step is to create an object of the class Drive, and also a stream variable that opens a file containing the binary data. We weren't given the name of any files so let's just make one up.

```
Drive Joint;
ifstream is("BinaryData.dat");
```

The next step is to create the function pointer. A function pointer allows you to create a reference to a function, whilst also passing arguments through function if needed.

```
int (*fp[6])(int) = {DriveJoint1, DriveJoint2, DriveJoint3,
DriveJoint4, DriveJoint5, DriveJoint6};
```

Therefore `fp[0](1)` will point towards `DriveJoint1(1)`;

Finally, we must iterate through the stream variable (`is`) until it reaches the end of file.

Whilst iterating, the data must be stored in the class using our overloaded operator (`>>`) which will store the data into the struct. It must also use the function pointer that we created to pass arguments through the joint control algorithms (pre-defined in the question) and therefore drive the robot.

```
while(!is.eof())
{
    is >> Joint;
    while(fp[Joint.GetJointNo() - 1](Joint.GetPosition()));
}
```

The reason why the function pointer is encased in a while loop is because the question states that the functions return 0 when finished and 1 when in motion. Therefore we continue calling the function pointer until it returns 0.

5. Give a complete `main()` function showing how the robotic manipulator motion can be carried out. (4 marks)

This part is just free marks. The previous question we did most of the work so you just need to wrap it up into a main function. Make sure to include all the headers.

```
#include <iostream>
#include <fstream>
#include "asea1230.h"
#include "drive.h"

using namespace std;

//Part 5
int main()
{
    Drive Joint;
    ifstream is("BinaryData.dat", ios::binary);
    //Part 4: Describe the following line
    int (*fp[6])(int) = {DriveJoint1, DriveJoint2, DriveJoint3,
    DriveJoint4, DriveJoint5, DriveJoint6};

    while(!is.eof())
    {
        is >> Joint;
```

```
        while(fp[Joint.GetJointNo() -1 ](Joint.GetPosition()));  
    }  
  
    return 0;  
}
```

## Question 2 - Example

I think this is a really good example and possibly could be something that we see in the exam. It manages to assess a wide range of knowledge in just one question (i.e. Fundamental data types, data structures, classes, operator overloading (both member and non-member), copy constructors, pointers, reading and writing object data to binary files, for loops)

In a certain complex process control system, a set of sensor data collected by the process are transmitted to a SCADA system as a binary data record of 75 bytes. The data corresponds to the binary form of six double type variables, three char type variables, one int type variable followed by two more double type variable and the checksum formed as a four byte entity. The data records are continuously arriving and one are sorted and extracted by the manufacturers software and is presented to the user as a 75 byte data record. Do the following:

- (a) Develop a suitable data structure to store the arriving data

This part is pretty much given to you in the question. You need to make a data structure that is able to store the variables in the order given. The only tricky part about is that `#pragma pack(1)` is needed. This is because the alignment of the different data type leaves padding that we don't want to use. If you don't get this look at the lecture Friday Oct 27, at around 20 minutes.

```
#pragma pack(1)
struct Data {
    double A[6];
    char B[3];
    int C;
    double D[2];
    unsigned int checksum;
};
```

- (b) Develop an object class (only the class definition is required) to manage this data by incorporating the following functionality.
1. Constructor
  2. Copy Constructor
  3. A function to store the receiving data in your data structure
  4. A function to write the receiving data to binary data file
  5. A member function to overload the operator '<<' that will print all the data in the above mentioned order
  6. Destructor

I was a little confused with the question originally. I thought that he want us to write the class with function definitions included. However, all this question is asking for is the class structure.

The constructor, copy constructor and destructor are just free marks and i won't go into detail about them.

The `store_data()` function uses the pointer 'p' to store the data into our array. This question originally through me off as i thought that the data will be coming from a binary file. However in (b) q3, it just says 'receiving data'. We instead consider the 75 bytes as an array of characters and therefore use an unsigned char pointer to access it.

The `write_data()` function will write the data from our data structure into a binary file.

Finally in the lecture, Friday October 27, Dr Jay says that he got number 5 wrong. He instead meant to say "a non- member function". In fact, it is not possible to write a member function to overload `<<`. I won't go into it because it is not necessary but if you would like to know why, read up!

<https://stackoverflow.com/questions/9814345/cant-overload-operator-as-member-function>

Therefore as a non-member function, we can use a friend function and overload the operator `<<`.

```
class Storage {
private:
    Data D;
public:
    Storage();
    Storage(const Storage &d);
    void store_data(unsigned char *p);
    void write_data(ofstream& is);
    //Non - Member function that overloads << to print to stdout
    friend ostream& operator<<(ostream& out, const Storage& d);
    ~Storage();
};
```

(c) Give the complete definitions of the functions b)3, b)4, and b)5.

b)3

For this function we are casting a pointer to an array (p) through to a pointer to our data structure (BytePtr). As we iterate through the 75 bytes, each byte is getting stored into our data structure. As long as our data structure is in the correct order, and we use `#pragma pack(1)`, it will store in the correct places. It is important that we increase both the array pointer and the data structure pointer for each iteration.



```

void Storage::store_data(unsigned char *p){
    unsigned char * BytePtr = (unsigned char *)&D;

    for (int i = 0; i < 75; i++) {
        *(BytePtr + i) = *(p + i);
    }
}

```

b)4

Again, we will make a pointer 'BytePtr' that points towards the first element in the data structure. We will then iterate through each byte and use the put() function to store the data in the ofstream variable 'os'.

```

void Storage::write_data(ofstream& os) {
    unsigned char * BytePtr = (unsigned char *)&D;

    for (int i = 0; i < sizeof(D); i++) {
        os.put(*(BytePtr + i));
    }
}

```

b)5

Here we do a simply non-member function overload of the operator '<<'. This allows us to return a chain of outputs to stdout.

```

ostream& operator<<(ostream& out, const Storage& d) {
    for (int i = 0; i < 6; i++) {
        out << d.D.A[i] << endl;
    }

    for (int i = 0; i < 3; i++) {
        out << d.D.B[i] << endl;
    }

    out << d.D.C << endl;
}

```

```

        for (int i = 0; i < 2; i++) {
            out << d.D.D[i] << endl;
        }

        out << d.D.checksum << endl;

        return out;
    }

```

### Question 3 - Example

*This is a question made by myself, styled off DrJays previous questions and in attempt to incorporate as many elements of the course that i could. It is quite similar to question 2 above but with a few extra bits. **If there is anything wrong with the solution please let me know so i can fix it.***

UNSW is attempting to design a fully animated world. The platform for the world is already built, but there are no buildings as of yet. In order to create these buildings, UNSW is going to send you a binary file that contains the buildings height as an int, two double type variables, one that represents its length and the other breadth, three double type variables that represent its RGB values, two double type variables that represent its position (x and y) and one character variable that represents the building's name, all of which is in that order.

Note: Assume the building's name is between 0 and 50 characters.

1. Develop a data structure to represent the data to be read from the file representing the building.
2. Develop an object class to manage this data by incorporating the following functionality:
  - a. Constructor
  - b. Copy Constructor that increases the length of the building by a multiple of 2.
  - c. A function to calculate the volume of space that the building will occupy. This must be returned as a long double variable.
  - d. Overload the operator '>>' to read the receiving data into your data structure
  - e. Overload the operator '<<' to write the data in your data structure to a binary file
  - f. A member function to overload the operator '<<' that will print all the data in the above mentioned order.
  - g. A function that overloads the operator '<' for the class. Building A < Building B if:
    - i. A.length < B.length
    - ii. A.breadth < B.breadth and A.length = B.length
  - h. A destructor that prints out "I <3 Mechatronics" when it is called.
3. Write a main function that creates two objects of your class ( B and C). B is going to contain data read from a file "BinaryFile.dat", whilst C is going to be copy of B with a length twice of that of B.

Demonstrate that your overload of operator '<' works. Print the values of Bs parameters and its volume to stdout. Finally, write C's data to a binary file named "OutputFile.dat".

## Solutions

1. Develop a data structure to represent the data to be read from the file representing the building.

Pretty simple, just make a data structure that fits the question.

```
#pragma pack(1)
struct Data {
    int height;
    double length;
    double breadth;
    double RGB[3];
    double x;
    double y;
    char name[50];
};
```

*Note: You may feel the need to use a string for the name. Strings within a struct are not contiguous and therefore when you try to use the >> operator to put a binary file into the struct, it will produce an error.*

2. Develop an object class to manage this data by incorporating the following functionality

- Constructor (Going to skip this because if you don't know how to make a constructor by now you're doomed)
- Copy Constructor that increases the length of the building by a multiple of 2.

```
class Building {
private:
    Data D;
public:
    Building();
    Building(const Building &b);
}

Building::Building() {
}
```

```
Building::Building(const Building &b) {
    D = b.D;
    D.length = D.length * 2;
}
```

- c. A function to calculate the volume of space that the building will occupy. This must be returned as a long long variable.

```
class Building {
private:
    Data D;
public:
    Building();
    Building(const Building &b);
    long double volume();
}

Building::Building() {

}

Building::Building(const Building &b) {
    D = b.D;
    D.length = D.length * 2;
}

long double Building::volume() {
    return (long double)D.length*D.height*D.breadth;
}
```

*Note: In order to return the volume in the correct data type make sure to return (long double)*

- d. Overload the operator '>>' to read the receiving data into your data structure

Similar to question 2, but we are instead overloading and not using a member function. if you need an explanation please look there.

```
class Building {
private:
    Data D;
public:
```

```

    Building();
    Building(const Building &b);
    long double volume();
    friend ifstream& operator>>(ifstream& is, Building& b);
}

Building::Building() {

}

Building::Building(const Building &b) {
    D = b.D;
    D.length = D.length * 2;
}

long double Building::volume() {
    return (long double)D.length*D.height*D.breadth;
}

ifstream& operator>>(ifstream& is, Building& b) {
    unsigned char * BytePtr = (unsigned char *)&b.D;

    for (int i = 0; i < sizeof(b.D); i++) {
        *(BytePtr + i) = is.get();
    }

    return is;
}

```

- e. Overload the operator '<<' to write the receiving data from your data structure

Again look at question 2 if you need help with this concept.

```

class Building {
private:
    Data D;
public:
    Building();
    Building(const Building &b);
    long double volume();
    friend ifstream& operator>>(ifstream& is, Building& b);
}

```

```

    friend ostream& operator<<(ostream& os, Building& b);
}

Building::Building() {

}

Building::Building(const Building &b) {
    D = b.D;
    D.length = D.length * 2;
}

long double Building::volume() {
    return (long double)D.length*D.height*D.breadth;
}

ifstream& operator>>(ifstream& is, Building& b) {
    unsigned char * BytePtr = (unsigned char *)&b.D;

    for (int i = 0; i < sizeof(b.D); i++) {
        *(BytePtr + i) = is.get();
    }

    return is;
}

ostream& operator<<(ostream& os, Building& b) {
    unsigned char *BytePtr = (unsigned char *)&b.D;

    for (int i = 0; i < sizeof(b.D); i++) {
        os.put(*(BytePtr + i));
    }
    return os;
}

```

- f. A member function to overload the operator '<<' that will print all the data in the above mentioned order.

Again look at question 2 if you need help with this one.

```

class Building {

```

```

private:
    Data D;
public:
    Building();
    Building(const Building &b);
    long double volume();
    friend ifstream& operator>>(ifstream& is, Building& b);
    friend ofstream& operator<<(ofstream& os, Building& b);
    friend ostream& operator<<(ostream& out, Building& b);
}

ostream& operator<<(ostream& out, Building& b) {
    out << b.D.height << endl;
    out << b.D.length << endl;
    out << b.D.breadth << endl;

    for (int i = 0; i < 3; i++) {
        out << b.D.RGB[i] << endl;
    }
    out << b.D.x << endl;
    out << b.D.y << endl;
    out << b.D.name << endl;

    return out;
}

Building::Building() {
}

Building::Building(const Building &b) {
    D = b.D;
    D.length = D.length * 2;
}

long double Building::volume() {
    return (long double)D.length*D.height*D.breadth;
}

ifstream& operator>>(ifstream& is, Building& b) {
    unsigned char * BytePtr = (unsigned char *)&b.D;

```

```

    for (int i = 0; i < sizeof(b.D); i++) {
        *(BytePtr + i) = is.get();
    }

    return is;
}

ofstream& operator<<(ofstream& os, Building& b) {
    unsigned char *BytePtr = (unsigned char *)&b.D;

    for (int i = 0; i < sizeof(b.D); i++) {
        os.put(*(BytePtr + i));
    }
    return os;
}

```

- g. A function that overloads the operator ' $<$ ' for the class. Building A  $<$  Building B if:
- A.length  $<$  B.length
  - A.breadth  $<$  B.breadth and A.length = A.length

This question is asking you to compare two of the Buildings given a set of criteria. If it fits the criteria then return true, and if not then false (i.e. use a bool statement). Since i'm not overloading a stream variable, i will just use a normal member function.

**Below is the entire code for the class and its functions.**

```

class Building {
private:
    Data D;
public:
    Building();
    Building(const Building &b);
    long double volume();

    friend ifstream& operator>>(ifstream& is, Building& b);
    friend ofstream& operator<<(ofstream& os, Building& b);
    friend ostream& operator<<(ostream& out, Building& b);
}

ostream& operator<<(ostream& out, Building& b) {

```



```
        out << b.D.height << endl;
        out << b.D.length << endl;
        out << b.D.breadth << endl;

        for (int i = 0; i < 3; i++) {
            out << b.D.RGB[i] << endl;
        }
        out << b.D.x << endl;
        out << b.D.y << endl;
        out << b.D.name << endl;

        return out;
    }

    bool operator<(Building& b) {
        if (D.length < b.D.length) {
            return true;
        }
        else if (D.breadth < b.D.breadth && D.length == b.D.length) {
            return true;
        }
        else {
            return false;
        }
    }

    ~Building();
};

Building::Building() {

}

Building::Building(const Building &b) {
    D = b.D;
    D.length = D.length*2;
}

long double Building::volume() {
    return (long double)D.length*D.height*D.breadth;
}
```

```

ifstream& operator>>(ifstream& is, Building& b) {
    unsigned char * BytePtr = (unsigned char *)&b.D;

    for (int i = 0; i < sizeof(b.D); i++) {
        *(BytePtr + i) = is.get();
    }

    return is;
}

ofstream& operator<<(ofstream& os, Building& b) {
    unsigned char *BytePtr = (unsigned char *)&b.D;

    for (int i = 0; i < sizeof(b.D); i++) {
        os.put(*(BytePtr + i));
    }
    return os;
}

Building::~~Building() {
    cout << "I <3 Mechatronics" << endl;
}

```

3. Write a main function that creates two objects of your class ( B and C). B is going to contain data read from a file “BinaryFile.dat”, whilst C is going to be copy of B with a length twice of that of B. Demonstrate that your overload of operator ‘<’ works. Print the values of Bs parameters and its volume to stdout. Finally, write C’s data to a binary file named “OutputFile.dat”.

```

int main(void) {

    Building B;

    ifstream is("BinaryFile1.dat", ios::binary);
    ofstream os("OutputFile.dat", ios::binary);

    is >> B; //storing the files data into the data structure

    Building C(B); //Creating Building C
}

```

```
//Check if C is larger than B
if (B < C) {
    cout << "True" << endl;
}

//Outputting B's parameters and volume
cout << B;
cout << "Volume of B is " << B.volume() << endl;

//Writing C's parameters to a file in binary
os << C;

os.close();
is.close();

return 0;
}
```

Output will look like this for the “BinaryFile1.dat”

```
True
10
5
4
1.2
3.4
0.2
2
0
DrJaysCrib
Volume of B is 200
I <3 Mechatronics
```

## Question 4 - Example

*This is another question made by myself in attempt to style a possible examination problem.*

*Outcomes Being Assessed: Class Structure, Data Types, Inheritance, Vectors, Iterators, Pointers, If and If-Else Statements. **If there is anything wrong with the question or solution please let me know.***

Jimmy, one of your good mates, loves to play poker but never knows what to do when all the cards are down on the table. He has designed a poker algorithm to help him make decisions but is having trouble putting it all together. He has created a vector of the 5 different classes and would like you to implement a class called “My\_Turn” that iterates through all of the classes and decides which move to choose.

The classes are below :

```
class Jimmy {
protected:
    int hole_cards[2]; //Two cards given at the start of the game
    int table_cards[5]; //5 cards on the table
public:
    Jimmy();
    void set_hole_cards(int c1, int c2);
    void set_table_cards(int c1, int c2, int c3, int c4, int c5);
    virtual bool move();
    ~Jimmy();
};
```

```
class Fold : public Jimmy {
public:
    Fold();
    virtual bool move();
    ~Fold()
};
```

```
class Allin : public Jimmy {
public:
    Allin();
    virtual bool move();
    ~Allin();
};
```

```
class check : public Jimmy {
public:
    check();
    virtual bool move();
    ~check();
};
```

```
class call : public Jimmy {
public:
    call();
    int call_amount();
    virtual bool move();
    ~call();
};
```

```
class raise : public Jimmy {
public:
    raise();
    virtual bool move();
    int raise_amount();
    ~raise();
};
```

The following is the name and syntax of the vector.

```
vector<Jimmy> jimmy;
```

Your class should contain its own move() function that stores the following encoded numbers;

- 1 = fold
- 2 = all in
- 3 = check
- 4 = call
- 5 = raise

*HInt: Think back to assignment two*

Solutions

The solutions to this problem are almost identical to that of the assignment. The move() function is the centre

function of the entire problem. It will use an iterator (it) to iterate through the predefined vector in the problem. Dynamic casting has to be used in this scenario as we are casting two pointers (a pointer to individual classes and the iterator). If the dynamic\_cast fails, it will automatically return NULL (instead of an error).

Three integers must be defined in the MyTurn class: The amount to call, the amount to raise, the code related to which move to undertake.

```
class MyTurn : public Jimmy {
protected:
    int call_amount;
    int raise_amount;
    int no_move; //Integer between 1 and 5 depending on which move
public:
    MyTurn();
    virtual bool move();
    ~MyTurn();
};

MyTurn::MyTurn() {

}

bool MyTurn::move() {

    for (std::vector<Jimmy *>::iterator it = jimmy.begin(); it !=
jimmy.end(); it++) {

        Fold* foldptr;
        Allin* allinptr;
        check* checkptr;
        call* callptr;
        raise* raiseptr;

        foldptr = dynamic_cast<Fold*>(*it);
        allinptr = dynamic_cast<Allin*>(*it);
        checkptr = dynamic_cast<check*>(*it);
        callptr = dynamic_cast<call*>(*it);
        raiseptr = dynamic_cast<raise*>(*it);

        if (foldptr != NULL) {
            if (foldptr->move() == true) {
                no_move = 1;
            }
        }
    }
}
```

```
        return true;
    }
}
else if (allinptr != NULL) {
    if (allinptr->move() == true) {
        no_move = 2;
        return true;
    }
}
else if (checkptr != NULL) {
    if (checkptr->move() == true) {
        no_move = 3;
        return true;
    }
}
else if (callptr != NULL) {
    if (callptr->move() == true) {
        call_amount = callptr->call_amount;
        no_move = 4;
        return true;
    }
}
else if (raiseptr != NULL) {
    if (raiseptr->move() == true) {
        raise_amount = raiseptr->raise_amount();
        no_move = 5;
        return true;
    }
}

}

}

MyTurn::~MyTurn() {

}
```

## Question 5 - Example

*Another example made by myself. If there is anything wrong with it please leave a comment.*

*Outcomes being assessed: Object Class, Templates, For and While loops, Reading binary data, Writing ASCII files, Manipulators, Overloading Operators, Data structures, Data Types*

DrJay is trying to sort out everyone's marks in a logical manner. He currently has a binary file that contains the student's first name and last name, the students zid (as an integer without the z), and the students marks for assignment1 (as a double), assignment2 (as an int) and the matlab assignment (as a long double) for each individual student. The files are in that exact order. He would really love it if you could help him make a program that is able to extract the data from the file and then print it to a txt.file in the following order:

Last Name, First Name: zID      ass1\_mark      ass2\_mark      mat\_mark      mark\_total

There has to be a comma after the Last Name, with a space before the First name. There also has to be a colon after the First Name and then 5 spaces in between each value afterwards. For example:

<b>Issa, Sharif:</b>	5555555	20.00	16	10.00	46.00
----------------------	---------	-------	----	-------	-------

Dr Jay would appreciate if those floating point numbers with decimal places be set to a precision of 2 (even if it is a whole number as you can see above).

Assume,

0 < Last Name < 20 (characters)

0 < First Name < 20 (characters)

1. Create a data structure that is able to store all of the above data
2. Develop an object class (only the class definition is required) to manage this data by incorporating the following functionality.
  - a. Constructor
  - b. Overload the operator (>>) to store the data into the data structure
  - c. A template function that adds the three marks together.
  - d. A function to print the data (in ASCII format) in Dr Jays preferred way to a txt file.
  - e. A destructor
3. Write a main function that is able to store incoming data into the data structure, and then output the stream to an ASCII file. The output should be able to add multiple names (i.e. not overwrite itself).

Solutions



Below is the header file (Question4.h)

```
#include<iostream>
#include<fstream>

using namespace std;

#ifndef QUESTION4_H
#define QUESTION4_H

#pragma pack(1)
//Question (1)
struct Data {
    char first_name[20];
    char last_name[20];
    int zID;
    double ass1_mark;
    int ass2_mark;
    long double mat_mark;
};

//Question 2
class Student {
private:
    Data D;
    double total_marks;
public:
    Student(); //Part (a)
    friend ifstream& operator>>(ifstream& is, Student& S); //Part (b)

    //Part (c)
    template<class T, class P, class C>
    void addMarks(T a, P b, C c);
    double get_a1();
    int get_a2();
    long double get_mm();

    friend ofstream& operator<<(ofstream& os, Student& S); //Part (d)

    void setup(char fn[20], char ln[20], int zid, double a1, int a2, long
```

```
double mm);

    ~Student(); //Part (e)

};

#endif
```

Below is the function file (Question4.cpp)

```
#include<iostream>
#include<fstream>
#include<iomanip>
#include "Question4.h"

using namespace std;

//Part (a)
Student::Student() {

}

//Part (b)
ifstream& operator>>(ifstream& is, Student& S) {
    unsigned char * BytePtr = (unsigned char *)&S.D;

    for (int i = 0; i < sizeof(S.D); i++) {
        *(BytePtr + i) = is.get();
    }

    return is;
}

//Part ©
//if you're not comfortable with templates please see 13.2
template<class T, class P, class C>
void Student::addMarks(T a, P b, C c) {
    total_marks = (a + b + c);
}

double Student::get_a1() {
    return D.ass1_mark;
```

```

}
int Student::get_a2() {
    return D.ass2_mark;
}
long double Student::get_mm() {
    return D.mat_mark;
}

//Part (d)
ofstream& operator<<(ofstream& os, Student& S) {

    os << fixed << showpoint << setprecision(2); //If you're not
    comfortable with manipulators please see section 15.4

    int i = 0;
    while (S.D.last_name[i] != '\0' ) {
        os << S.D.last_name[i];
        i++;
    }
    i = 0;
    os << ", ";

    while (S.D.first_name[i] != '\0') {
        os << S.D.first_name[i];
        i++;
    }
    os << ":      " << S.D.zID;
    os << "      " << S.D.ass1_mark;
    os << "      " << S.D.ass2_mark;
    os << "      " << S.D.mat_mark;
    os << "      " << S.D.total_marks << endl;

    return os;
}

//Part (e)
Student::~Student() {
}

```

Below is the main function

//Question (3)

```
int main() {  
  
    Student S;  
  
    ifstream is("Sharif_Issa.dat", ios::binary); //Make sure to use  
binary flag  
    ofstream os("DrJaysStudentData.txt", ios::app); //Make sure to use  
append(ios::app) allows you to add to a file  
  
    is >> S; //Storing data in structure  
  
    S.addMarks(S.get_a1(), S.get_a2(), S.get_mm()); //Adding the marks  
  
    os << S; //Outputting to file  
  
    is.close();  
    os.close();  
  
    return 0;  
  
}
```

## Question 6 - Example

*Another example created by myself. Outcomes being accessed: Object Class, Writing ASCII files, overloading operators, copy constructors, static data types, Data types, Manipulators*

You currently work at a car dealership that sells used cars of all types. Your boss has asked you to create a program that is able add certain information about the cars they have as they receive them into their inventory. In particular, he would like to know the the cars manufacturers name and the number of seats it has. He would also like you to keep a record of how many cars they have as they come into the dealership.

To do the following he would like you to:

1. Create a class that declares the following functions
  - a. A variable for the manufacturers name, number of seats and car count
  - b. A default constructor
  - c. An overloaded constructor that sets the name and number of seats.
  - d. Overload the operator “=” that will be used to copy the classes (called a copy assignment operator)
  - e. Overload the operator “<<” that will output the data to an ASCII file.
  - f. A destructor
2. Define the each of the above functions. Remember to continuously update the car count variable with each new object of the class created. The output file should look like this:

Manufacturer Name	No. of Seats	Car Count
<b>Toyota</b>	<b>4</b>	<b>1</b>
<b>Ford</b>	<b>5</b>	<b>2</b>

There are ten spaces between each variable. Also the names of each car manufacturer should be line up as shown in the figure. Give 20 spaces for the names of the manufacturer. (*Hint use setw()*)

3. Create a main function that creates the creates the following five cars:

Toyota	4	1
Toyota	4	2
Ford	5	3
Toyota	2	4
Lamborghini	2	4

Note that after the 2 seat Toyota, the car count does not increase. This is because this car was sold before the lamborghini came into the dealership and was therefore deleted from the system. (*Hint, in order to delete a class manually it must be allocated in heap i.e. use a pointer allocated by new*).

Also note that the two top Toyotas are copies of each other.

## Solutions

1. Create a class that declares the following functions

Most of this is fairly regular syntax. It's important that you declare the car count as a static variable. If you are not familiar with static data types please refer to 7.6 of these notes. The copy assignment operator (=) is also interesting, it is a combination of a copy constructor and a overloaded operator together. We have defined it as a member function here.

```
#include <iostream>
#include <string>
#include <fstream>

using namespace std;

#ifndef CAR_H
#define CAR_H

class Car {
private:
    //(a)
    string man_name;
    unsigned int numSeats; //unsigned because we cant have <0 seats
    static unsigned int CarCount; //Static because we want to increment
it each class
public:
    Car(); //(b)
    Car(string mn, unsigned int ns); //(c)
    Car& operator=(const Car& c); //(d)
    friend ostream& operator<<(ostream& os, Car &c); //(e)
    ~Car(); //(f)
};
#endif
```

2. Define the each of the above functions. Remember to continuously update the car count variable with each new object of the class created. The output file should look like this:

It is important that you initialize the CarCount to 0 here, and then increment it and decrement it when a car is created or sold.

In order to align the data correctly use setw(), read 15.4 of these notes if you don't understand what this does. It is important that you include the header file #include <iomanip>.

```
#include <iostream>
#include <string>
#include <fstream>
#include <iomanip>
#include "Dealership.h"

unsigned int Car::CarCount = 0; //Important to initialise the CarCount

Car::Car() {
    CarCount++;
}

Car::Car(string mn, unsigned int ns) {
    man_name = mn;
    numSeats = ns;
    CarCount++;
}

Car& Car::operator=(const Car& c) {
    man_name = c.man_name;
    numSeats = c.numSeats;
    return *this; //Returns the object (type Car&)
}

ofstream& operator<<(ofstream&os, Car &c) {
    os << setw(20) <<c.man_name << " " << c.numSeats << " "
    << c.CarCount << endl;
    return os;
}

Car::~~Car() {
    CarCount--;
}
```

3. Create a main function that creates the creates the following five cars:

```
int main() {  
  
    ofstream os("Dealership.txt", ios::app); //The flag app allows you to  
    append the file  
  
    Car one("Toyota", 4);  
    os << one;  
  
    Car two;  
    two = one; //uses the overloaded copy operator  
    os << two;  
  
    Car three("Ford", 5);  
  
    os << three;  
  
    Car* four = new Car("Toyota", 2); //Must declare in heap so it can be  
    deleted  
    os << *four;  
  
    delete four;  
  
    Car five("Lamborghini", 2);  
    os << five;  
  
    os.close();  
    return 0;  
}
```