C++ Revision

- Extension to the C programming language
- Object-Oriented language

*The **main** Function*

- Any program written in C and C++ begins with a function called main, the first statement in the main function is the first statement to be executed. C++ programs can only have one **main** function.
- { and } specify when main starts and when it ends.

```
void main()
 {    // Statements
 }
```

Semi-colons

- At the end of every line of code (statement) comes a semi colon. The most common error in C++ is omitting a ;
- This causes the compiler to get confused and it won't compile your program.

Comments

- Single Line: Type // and anything that follows is ignored by the compiler

    // This is a comment

- Multi-line: Type /* and anything that follows is ignored by the compiler until you type */

    /* Multi- line comments are also
    ignored by the compiler */

C++ Revision

Section One:
Types and Operators

Types

- Types can be used to define variables that hold data.
- A type is a template describing the kind of data that variables of the specified type can contain and how these variables can behave
- C provides two kinds of type, intrinsic and user-defined.

Intrinsic Data & Operations

- Numeric Data:
  - Integers int long
  - Real Numbers float double
- Textual Data:
  - Characters char
  - Strings arrays of char
- Operations yield values: 13+14 = 27

Intrinsic Types

Type Name	Bytes	Range of Values
int	2 *	–22,768 **to** 32,767 *
unsigned int	2 *	0 **to** 55,535 *
char	1	-128 **to** 127
unsigned char	1	0 **to** 255
short	2	-32,768 **to** 32,767
unsigned short	2	0 **to** 65,535
long	4	-2,147,483,648 **to** 2,147,483,647
unsigned long	4	0 **to** 4,294,967,295
float	4	3.4E +/- 38 (7 digits)
double	8	1.7E +/- 308 (15 digits)
long double	10	1.2E +/- 4932 (19 digits)

* The size of the integer type (int) depends on the operating system.

- **Type:** The template which specifies the data
- **Instance:** A particular location in memory containing data as a type
- **Variable:** The name used by the programmer to refer to the memory location containing data as a type

Using Types

- To declare an integer to be used as a counter use the syntax:

```
void main()
{   int counter;
 }
```

- C allows us to give a variable, of an intrinsic type, an initial value.

```
void main()
{   int counter = 0;
 }
```

- To assign the letter ‘A’ to a variable of type char we would write:

```
void main()
  {
   char LetterA;
   LetterA = 'A';
  }
```

User Defined Types

- **Structure:**
  - a group of types which form a new type
  - a collection of simple data-types, some can be float, char, int and so on. The data items in a structure are called *members* of the structure.

Structure Example

```
struct night_out
  {
  int     drink;
  int     cigarettes;
  float   cost;
  };
```

Arrays

- An array is a series of variables, which is referred to with the same name using a number, called an index, to reference individual elements.
- The lower bound of an array is 0, and the upper bound is the amount of elements in the array less one.
- The range of an array is 0 to N-1, N being the amount of elements in the array.

Defining Arrays

- `int Numbers[10];`
  defines an array of ten integers
- To reference element 5 of the array we use `Numbers[4]`.
- To assign a value of one hundred to the first element in the above array we would say `Numbers[0] = 100;`

Defining Arrays

- `int Numbers[10];`
  defines an array of ten integers
- To reference element 5 of the array we use `Numbers[4]`.
- To assign a value of one hundred to the first element in the above array we would say `Numbers[0] = 100;`

...Constants

- **Note:** It is not possible to copy the entire contents of one array to another using assignment - each element must be copied individually.
- **Constants:** It is possible to give an element of any type a value which cannot be changed during execution of the program.

```
const int OneHundred = 100;
int Numbers[OneHundred];
```

Operators (1)

- Numeric operators include:
  - addition +
  - subtraction −
  - multiplication *
  - division /
  - modular division % (returns remainder)
  - increment ++
  - decrement --
  - addition assignment +=
  - also … -=, *=, /= , %=

Basic Arithmetic in C++

```
void main()
{
        int a = 1+1;
        int b = a *2;
        int c = b/2;
        a += 3;
        c *= 5;
        int d = 10 % 4;
}
```

++ and -- Operators

- The statements
  x = x +1; and x +=1;
  can be written using the ++ operator as
  x ++;
- The statements
  x = x -1; and x -=1;
  can be written using the -- operator as
  x --;

Post fix or Prefix ?

```
#include <iostream.h>
void main()
{    int x = 0;
     int y = 10;
     x =  ++ y;
     // What value has x ?
}
```

If we used x = y ++ instead of x = ++ y what value would x have ?

The Difference!

- x = ++ y gives x the value 11 as it is equivalent to : y = y +1; x = y;
- x = y ++ gives x the value 10 as it is equivalent to : x = y; y = y +1;

Operators (2)

- Relational operators include:
  - equal to '=='
  - not equal to '!='
  - less than '<'
  - greater than '>'
  - less than or equal to '<='
  - greater than or equal to '>='

Operators (3)

- The expression `(x = y)` will result in the value held by the variable `y` being assigned to the variable `x`.
- Logical operators allow conjunction and disjunction of relational expressions, these operators are:
  - **and** denoted by the symbol '`&&`'
  - **or** denoted by the symbol '`||`'
- Precedence : !, &&, ||

Boolean Results

- Any expression, using the relational operators, evaluates to a Boolean result, either true or false, 1 or 0 in C/C++.
- Example:
  - The result of `10 < 20` is 1 or true
  - The result of `20 < 10` is 0 or false.
- The relational operators also work with variables. If we take two integers `x` and `y` we can test them for equivalence by using the result of `(x == y)`.

C++ Revision

Section Two:

Functions

Functions

If we were to create a function in order to add two integers producing an integer result we would use the following notation:

```
int add(int x, int y)
{
  int returnValue = 0;
  returnValue = x + y;
  return returnValue;
}
```

Return Types

- **Return Type:** The type which the function returns. If you ask someone to add two number you would expect them to give you/ return a value.
- If you write a function to add two integers together, you would expect it to return an integer. In this case the return type is int
- Functions return one value
- Function which have no return type are labelled void

Using Functions

```
int add(int x, int y);
void main()
{
     int first = 10;
     int second = 2;
     int result;
     result= add(first, second);
}
```

Problems with Functions

- Return Values are the only way to retrieve information from a function
- **Problem 1:** only one item of data can be retrieved from a function call.
- **Problem 2:** no variables passed into a function can have their values changed. Take a situation where we might want to write a function to swap the values held in two variables

The Swap Function

```
void swap(int x, int y)
{
  // Create a temporary variable
  int temp;

  temp = x;
  x = y;
  y = temp;
}
```

The Swap Program

```
// Prototype for our swap function
void swap(int x, int y);

void main()
{
  int FirstInt = 10;
  int SecondInt = 20;
  swap(FirstInt, SecondInt);
}
```

Is there a problem here?

Passing values by reference

- **The swap doesn't happen! Why?**

- C/C++ passes parameters by value, i.e. a copy of the variable is passed to the function, not the actual value itself.
- C++ can pass the actual variables themselves - known as *passing parameters by reference*.
- To pass a parameter by reference we place & between the parameters type name and the parameter tag.

The New Swap Program

```
void swap(int& x, int& y);

void main()
{
  int FirstInt = 10;
  int SecondInt = 20;
  // Call the swap() function
  swap(FirstInt, SecondInt);
}
```

The New Swap Function

```
void swap(int& x, int& y)
{
  // Create a temporary variable
  int temp;

  temp = x;
  x = y;
  y = temp;
}
```

What about functions and arrays / structures?

Function Overloading(1)

- C++ makes it possible for functions with the same name to have different parameter lists.
- Say we wanted to write a function to output the value of a variable along with it's type we want to use the same function for `ints` as well as `floats`.

Function Overloading (2)

```
// Function Prototypes
void OutputVariable(int value);
void OutputVariable(float value);

void main()
{
  int i = 10;
  float f = 1.3;

  OutputVariable(i);
  OutputVariable(f);
}
```

C++ Revision

Section Three:
Input & Output

Libraries

- A library is a binary file that is compiled into a program.
- An include file, or header file is a C/C++ file that describes functions without actually implementing them.
- Structures can also be defined in a header file so that many source files can use them.
- A function description in a header file is called a function prototype.
  - Example: `int add(int x, int y);`

Input & Output

- Uses a header file called iostream.h - this provides two objects `cin` and `cout`.
- `cin` means “console in” and `cout` means “console out”.
- To output text with the `cout` object we use what are known as the stream out operators ‘<<’.
- To input with `cin` we use the stream in operators ‘>>’.

Output Example 1

```
#include <iostream.h>

// The main function
void main()
{    // Stream some text to cout
 cout << "Hello World";
}
```

Output Example 2

```
#include <iostream.h>
void main()
{
 int x = 10;

  // Use the << operators to stream
  out each piece of data
 cout << "The value of x is " << x;
}
```

A more useful Adding Program

```
#include <iostream.h>
int add(int x, int y);
void main()
{
  int first;
  int second;
  int result;

  cout << "Enter the first number : ";
  cin >> first;
```

```
  cout << "Enter the second number : ";
  cin >> second;

  result = add(first, second);

  cout << "The result is : " << result <<
  endl;
}
```

- Note: endl sets the cursor to the start of the next line on the screen.

C++ Revision

Section Four:

Selection

the if statement

the switch statement

Selection Example (1)

```
if (condition)
{ statement1; }
else if (condition)
{ statement2;}
else if (condition)
{ statement3;}
    :
    :
else
{ statementN; }
```

Selection (2)

- The `else` and `else if` clauses are optional.
- If an expression evaluates to true the set of statements immediately following it, i.e. those between the nearest { } parenthesis, are executed.
- Once a set of statements is selected no other statements in the `if - else if - else` clause, will be executed.
- The form of selection is sequential and deterministic.

Selection (3)

- In a case where there are no expressions that evaluate to true, the statements following the `else` statement are executed.
- If more than one expression in the `if - else if` clause evaluate to true only the first will be selected.
- An example follows ...

A Selection Example

```
int max(int x, int y)
{
 int maximum;
 if (x < y)
  {
   maximum = y;
  }
 else
  {
   maximum = x;
  }
 return maximum;
}
```

```
void main()
{    int grade;
     cout << "enter the grade:" << endl;
     cin >> grade;

     if ( grade >= 80 )
     cout << "A" << endl;
     else if (grade >= 70 )
     cout << "B" << endl;
     else if ( grade >= 60 )
     cout << "C" << endl;
     else if (grade >= 50 )
     cout << "D" << endl;
     else  cout << "FAIL" << endl;}
```

Nested if / else

```
if ( i == 10 ) {                          // this if
       if ( j < 20 )
      a = b;
       if ( k > 100 )   // this if
      c = d;
      else              //with this else
      a = c;
}
else
      a = d;                              // with this else
```

A Ternary Operator

- The conditional operator (`?:`) is closely related to the **if / else** structure.
- The `?:` is C++'s only ternary operator. Whereas a binary operator takes two parameters, a ternary operator takes three.

- It takes three operands:
  1. Boolean expression (the condition)
  2. value if condition is true
  3. **value** if condition is false
- The general syntax of the ternary operator is:
  – <Boolean expr> **?** <value if true> **:** <value if false>

Ternary operator: example

cout << (grade >= 50 **?** "Passed" **:** "Failed");

boolean expression **value if true** **value if false**

switch

- The `switch` statement is C++'s multiway branch statement.
- It provides an easy way to dispatch execution to different parts of your code based on the value of an integer expression.
- If often provides a better alternative than a large series of **if-then-else** statements.

- The general form of a switch statement is:-

switch(expression**) {**
case value1 **:** <appropriate C++ statements>
        break;
case value2 **:** <appropriate C++ statements>
        break;
    :
default **:** <appropriate C++ statements>
        break;
}

- The `switch` expression must produce a value that is an **integer** or **character**.
- The case values are of the same type as the switch expression.
- The **switch** can only test for equality.
- There may be one or more case values for a given statement.
- The case values do not have to be in any order
- A case value may occur only **once**.

Example: Currency Converter

- We're going to develop a simple currency converter, and we'll use the switch statement to help.
- First we declare the rates as constants in our program.

```
const float dollarRate = 1.3516;
const float yenRate = 163.34;
const float euroRate = 1.2697;
const float sterlingRate = 0.83511;
int main()
{
```

... contd. ...

Next, we declare variables to hold the amount to be converted and the user's choice of currency. We can then ask the user for this information using `cin` and `cout`.

```
float amount;
 char choice;

 cout << "enter the amount in punts:" ;
 cin >> amount;
 cout << "what currency? (d)ollar,
  (y)en, (e)uro or (s)terling? ";
 cin >> choice;
```

```
switch (choice) {
 case 'd':
   cout << amount << " = " << (amount * dollarRate)
    << " dollars." << endl;
   break;
 case 'e':
   cout << amount << " = " << (amount * euroRate)
    << " euros." << endl;
   break;
 case 'y':
   cout << amount << " = " << (amount * yenRate)
    << " yen." << endl;
   break;
```

```
case 's':
   cout << amount << " = " << (amount * sterlingRate)
    << " sterling." << endl;
   break;
 default:
   cout << "Sorry - can't do that currency!!" << endl;
 }
```

Why switch?

- If you're thinking, you'll have realised that that last switch statement could have been written using a chained if-then-else statement! So, why bother with switch?
- It's really just a matter of taste and style. Sometimes, if there's a lot of possibilities all hinging on the value in one variable, a switch statement better describes what's going on.

C++ Revision

Section Five:

Iteration

The While loop

The Do While loop

The For loop

Iteration

- Iteration is the means by which a set of statements can be repeated while some clause is not fulfilled.
- C/C++ provides three methods for the implementation of iteration
  - the `while` loop
  - the `do .. while` loop
  - the `for` loop

Iteration

While Loop

```
initialisation
while (guard)
 {   // Body
     Progress
 }
```

Do Loop

```
Initialisation
do
{ // Body
  Progress
}while(guard);
```

For Loop

```
for(initialisation;guard;
  progress)
{    // Body
}
```

The While Loop (1)

```
int x = 1;
int sum = 0;
while (x != 10) // The guard
    {
      // The body
      sum = sum + x;
      x = x + 1;
    }
```

The While Loop (2)

- In this example the result of the loop is that sum contains the sum of the number 1 …9
- The loop terminates when x equals 10.
- If the guard evaluates to true the associated set of statements (the loop body) are executed, this is repeated while the guard remains true.

***When** to use while*

- Use the while loop when you **do not know in advance** how many times the loop is to execute.
- A useful example is when you get information from the user. For example, you want the user to give you a 'y' for yes, or a 'n' for no. Try this:

```
char choice;
 cout << "enter your choice" << endl;
 cin >> choice;
 while (choice == 'y') {
   cout << "enter again, y or n: " << endl;
   cin >> choice;
 }
```

The Do While Loop (1)

- The statements within the body of the loop are executed once.
- The expression is evaluated, if it evaluates to true the statements in the body are executed and the expression evaluated again.
- This process continues until the expression no longer remains true.

The Do While Loop (2)

```
void main()
{
  int x = 1;
  int sum = 0;
  do
  {
     sum = sum + x;
     x = x + 1;
   }  while (x != 10)
}
```

Another Do While Example

```
#include <iostream.h>
void main()
{
  int x = 0;
  do
  { cout << "Enter a number between 0 and 9:";
    cin >> x;

  } while (x < 0 || x > 9);
    cout << "You entered " << x << endl;
}
```

while or do-while?

- Any `do...while` loop can be performed by just writing the code to do the operation once, and then using a `while` loop to handle the subsequent looping.
- As a result, many coders just stick to `for` loops and `while` loops, and don't bother using `do...while`.
- Its your choice!

The For Loop (0)

```
for(initialisation;guard;progress)
{
  // Body
}
```

The initialisation section allows “progress” variables to be declared and initialised for use within the loop.

while is equivalent to for

- Any **for**-loop can be rewritten as a while-loop.

```
for(initialisation; condition; update) {
    statement;
}
```

  – This loop can be rewritten as a while loop like this.

```
initialisation;
while(condition) {
  statement;
  update;
}
```

The For loop (1)

Declare control variable

Continuation condition

```
int counter;
for ( counter = 1; counter <= 10; counter++) {
    // body of loop
}
```

for keyword

Initialise control variable

increment of control variable

The For Loop (2)

- The body of the loop is executed while the guard evaluates to true.
- The progression section allows statements to be executed in order for the loop to progress.
- Progress is performed after the first iteration of the loop, and every subsequent time that the loop is executed.
- In all but the first iteration of the loop the progress section is executed before the guard is evaluated.

The For Loop (3)

```
include <iostream.h>
void main()
{
 int sum = 0;
  for (int x=1; x<10; x ++)
  {
      sum = sum +x;
  }
}
```

Declaring variables inside the loop

Often a variable that controls a **for** loop is only needed for the purposes of the loop and is not used elsewhere. If so, you can declare the loop control variable in the intialisation part of the loop:

```
for(int i = 0; i < 5; i++ )
{
     cout << "The value is" << i;
}
```

Scope of A Variable

- **Remember :** when you declare a variable **inside** a **for-**loop, the scope of that variable **ends** when the **for** statement does. This program won't compile, because the variable i exists only inside the for-loop.

```
void  main()
{ for (int i = 0; i < 7; i++)
   { cout << i << endl;
   }
 cout << i << endl;
}
```

- This is an example of block scope. When you declare a variable inside a block of code, it exists from the point of its declaration until the closing '}' of the block.

Using the comma

- You can make the initialisation of more than one loop variables explicit by using the comma

```
int main()
{
 int a, b;

 for( a = 1, b = 4; a < b; a++, b--) {
   cout << "a = " << a << endl;
   cout << "b = " << b << endl;
 }
}
```

Keep it Simple

- Loops with more than one controlling variable are very hard to understand and walk through. You can usually find another way to rewrite the loop so that you don't have to use more than one controlling variable.
- Don't make your code any more complicated than it has to be!

Infinite loops

- You can intentionally create an infinite loop (a loop that never terminates)
  - if you leave all three parts of the **for** empty.

```
for( ; ;) {
cout << "blah blah blah" << endl;
}
```

  - if you place a ; after the for as follows:

```
for ( x = 0; x < N;  x++) ;
```

Nested loops

- A nested loop is one loop inside another loop.

```
void main()
{ int width;
  cout << "enter the width of the square: ";
  cin >> width;
  for(int i = 0; i < width; i++) {
    for( int j = 0; j < width; j++){
      cout << "* " ;
    }
    cout << endl;;
  }
}
```

When to use the for-loop

- Use the for loop when you **know before the loops starts** how many times the loop is to execute.

Some Simple Loop Problems

Given an array of 10 elements populate the array and

- print out the contents of the array
- sum the elements in the array
- sum the even elements of the array
- print the largest value in the array
- locate the first occurrence of the value 9 in the array (if it exists)

C++ Revision

Section Six:

Arrays

Arrays

- An array is a series of variables, which is referred to with the same name using a number, called an index, to reference individual elements.
- The lower bound of an array is 0, and the upper bound is the amount of elements in the array less one.
- The range of an array is 0 to N-1, N being the amount of elements in the array.

Defining Arrays

- `int Numbers[10];`

  defines an array of ten integers
- To reference element 5 of the array we use `Numbers[4]`.
- To assign a value of one hundred to the first element in the above array we would say `Numbers[0] = 100;`

Loops and Arrays

```
#include <iostream.h>
void main()
{
  int age[4];

  for (int j = 0; j < 4;j++)
     {
     cout <<"Enter an age : ";
     cin >> age[j];
     }
  for (j = 0; j < 4; j++)
  cout <<" \nYou entered " <<age[j];
}
```

Average Sales Example

```
#include <iostream.h>

const int SIZE = 6;

void main()
{ float sales[SIZE];
  float total = 0;
  float average;
```

```
cout <<"\nEnter sales for six days \n";
  for (int j=0; j < SIZE; j++)
  {   cin >>sales[j];
  }
  for (j=0; j < SIZE; j++)
  { total += sales[j];
  }
  average = total / SIZE;
  cout <<"Average = " <<average;
}
```

Initialising Arrays

```
#include <iostream.h>
void main()
{ int month, day, total_days;
  int days_per_month[12] = {31, 28, 31, 30,
  31, 30, 31, 31, 30, 31, 30, 31};
  total_days = 0;

  for (int j=0; j < 12; j++)
     total_days += days_per_month[j];

    cout <<"\nTotal days:" << total_days;
}
```

2-D Arrays ...

- It is possible to define arrays with multiple dimensions, `int 2DArray[10][10];`
- To reference elements in this array we use two sets of square brackets
  - `2DArray [3][4] = 20;`
- The above statement assigns 20 to the element at position 3, 4. All array dimensions have zero based indexes.

```
#include <iostream.h>
const int BARS = 4;       //Rows
const int MONTHS = 3;     //Columns
void main()
{ int b, m;
  float sales[BARS][MONTHS];

  for (b=0; b<BARS; b++)
     for (m=0; m<MONTHS; m++)
     {
     cout <<"Enter sales for bar " <<b+1;
     cout <<", month " << m+1<< " : ";
     cin >> sales[b][m];
     }
}
```

Initializing 2D Arrays

```
float sales[BARS][MONTHS] =
     {
          {1432.07, 234.50, 654.01},
          {322.00,13838.32, 17589.01},
          {9328.34, 934.00, 4492.30},
          {12838.29,2332.63,  32.93}
     };
```

Arrays of Structures

```
#include <iostream.h>
const int SIZE = 4;

struct part
  {
  int modelnumber;
  int partnumber;
  float cost;
  };
```

```
void main()
{ int n;
  part apart[SIZE]; //array of structures

  for (n=0; n < SIZE; n++)
     {     cout << endl;
           cout << "Enter model number : ";
           cin >> apart[n].modelnumber;
           cout <<"Enter partnumber : ";
           cin >> apart[n].partnumber;
           cout <<"Enter cost : ";
           cin >> apart[n].cost;
     }
}
```

Arrays & Functions (1)

- When passing arrays to functions as parameters they always act as if they were passed as reference.
- To declare a function that takes an array as a parameter we use the square brackets after the parameter tag as follows:
  `int Fname(int Array[], int ArrayCount);`

Arrays & Functions (2)

- Although not syntactically necessary, it is good practice to pass the amount of elements contained within the array into the function, as it is rare that the function will know the amount of elements in advance.
- We might write a function that initialises all elements of an array to zero

```
#include <iostream.h>
void InitArray(int Array[], int ArrayCount);
  //Function Prototype

void main()
{
 int MyArray[10];
 InitArray(MyArray, 10); //Function call

 for (int i = 0; i < 10; i++)
  {cout <<"Element at: "<< i << "= ";
   cout << MyArray[i] << endl;
  }
}
```

```
void InitArray(int Array[], int ArrayCount)
{
 for (int i = 0; i < ArrayCount; i++)
  {
   Array[i] = 0;
  }
}
```

Non Partitioned Array Processing

- Processing every element of the array
- Example: Sum all values in an array

```
void main()
{const int N = 10;
   int Array[N] = {1,2,3,4,5,63,5,7,5,63};
   int Sum=0; int k;
   for (k=0; k<N; k++)
   {          Sum = Sum + Array[k]
   }
   cout << " The sum of the array elements is: " << Sum;
   }
```

Partitioned Array Processing

- Processing some elements of the array
- Example: Sum all even values in an array

```
void main()
{  const int N = 10;
   int Array[N] = {1,2,3,4,5,63,5,7,5,63};
   int Sum=0;
   for (int k=0; k<N; k++)                    The partition
   {          if (Array[k] % 2 == 0)
              {Sum = Sum + Array[k];}
   }
   cout << " The sum of the array elements is: " << Sum;
}
```

Linear Search.

– Guaranteed to find the item you are searching for:

```
#include<iostream.h>
void main()
{  const int N = 10;
   int Array[N] = {1,2,3,4,5,63,5,7,5,63};
   int x = 63, i= 0;          When I still haven't found what I'm
                              looking for - keep searching
while(Array[i] != x)
   { i = i+1;
   }
   cout << " x is at position: " << i;
   }
```

Bounded Linear Search

- In the Linear Search e.g. we were searching for the position of an element which we know is present
- In the Bounded Linear Search we are searching for something which may not be there
  – e.g. the position of an element which we may not be present

Bounded Linear Search.

Not guaranteed to find the item you are searching for:

```
#include<iostream.h>
void main()
{  const int N = 10;
   int Array[N] = {1,2,3,4,5,63,5,7,5,63};
   int x = 63, i= 0;          When I still haven't found what I'm
                              looking for and I haven't ran out of
                              search space - keep searching
while((Array[i] != x) && (i <N))
   { i = i+1;
   }
```

Bounded Linear Search ctd.

```
If (Array[i] == x)
{ cout << "Element found";
}
else   // (i <N))
{ cout << "Element not found";
}

}
```

If statement indicates if the element is found or if the search space is exhausted

Binary Search

```
void main ()
{   int array[6] = {1,2,3,4,5,6};
    int h, x = 4, a = 0, b = 5;
        while (a+1 != b)
        {       h = (a+b) /2;
                if (array[h] <= x )
                {       a = h;}
                else
                {       b = h;}
        }
cout<< "x is " <<x<<endl;
cout <<"It is stored at position "<<h<<endl;
}
```

C++ Revision

Section Seven:
Pointers

What is a Pointer?

- One of the major features that distinguishes the C++ language from most others is the abundant use of pointers.
- A pointer is a special kind of variable, one that does not hold the data item itself but its memory address.
- Only when a special statement is executed is memory set aside for this data value, and the pointer assigned the data item's address. Until this time, the pointer does not hold a meaningful address.

- When the location that the pointer references is no longer needed, the area is freed so that the computer can use it for other purposes.
- Thus, pointers enable the use of ***dynamic memory allocation.***

Dynamic Memory Allocation

- the expansion and contraction of memory allotments as needed during program execution.

Static Memory Allocation

- the setting aside of memory requirements before program execution begins. Throughout execution, the area remains allocated.

Pointers

What are they for ?

- Accessing array elements
- Passing arguments to a function when the function needs to modify the original argument.
- Passing arrays and strings to functions
- Obtaining memory from the system
- Creating data structures from linked lists

Pointers

- It is possible to do without pointers:
  - arrays can be accessed with array notation rather than pointer notation
  - a function can modify arguments passed, by reference, as well as those passed by pointers.
- However, in order to obtain the most from the language it is essential to use pointers.

The Address Operator &

It is possible to find out the address occupied by a variable by using the address of the operator &.

```
void main()
{
  int var1 = 11;
  int var2 = 22;
  int var3 = 33;

  cout     << endl << &var1
           << endl << &var2
           << endl << &var3;
}
```

Addresses

The actual addresses occupied by variables in a program depend on many factors, such as

- the computer the program is running on,
- the size of the operating system,
- and whether any other programs are currently in memory.

- For these reasons no two computers will give the same answer to the above program.

Creating Pointers

- For creation of a pointer variable that can hold the address of a data type *int*, an asterisk is used to write the type definition of *ptr* such as

  `int * ptr;`
- As a result of this declaration, room for an address is allocated to *ptr*.

The NULL Pointer

- No address has been placed in *ptr*, so its value is undefined. With *ptr* being undefined, a comparison involving *p* would be an error, although most C++ compilers would not flag the mistake.
- It is possible to assign the value constant NULL to indicate that *ptr* does not point to a memory allocation (*ptr* is no longer undefined).

Printing out variables that hold address values are useful.

```
void main()
{ int   var1 = 11;
  int*  ptr;  //defines a pointer
  ptr = Null; //sets pointer equal to Null
  cout <<endl <<&var1;
  ptr = &var1;
  cout <<endl <<ptr;
}
```

The * means *pointer to.* **ptr** is a pointer to an **int** (it can hold the address of integer variables).

Defining pointer variables

- defining pointer variables

```
char*  cptr;
int*   iptr;
float* fptr;
```

- defining multiple pointer variables

```
char*  ptr1, * ptr2, * ptr3;
```

Putting values in pointers

- Before a pointer is used a specific address must be placed in it :

```
ptr = &var1;
```

- A pointer can hold the address of any variable of the correct type;
- **Warning!**
  - it must be given some value, otherwise it will point to an arbitrary address (because it has to point to something).

Accessing the variable pointed to

```
void main()
  {int     var1 = 22;
   int*    ptr;

   ptr = &var1;
   cout <<endl <<*ptr;
  }
```

- **ptr** holds the address of var1
- *ptr holds the contents of var1

Using Pointers to Modify Variables

```
void main()
{  int  var1, var2;
   int* ptr;

   ptr = &var1;
   *ptr = 37;
   var2 = *ptr;
   cout <<endl<< var2;
}
```

Problems with Functions

- Return Values are the only way to retrieve information from a function
- **Problem 1:** only one item of data can be retrieved from a function call.
- **Problem 2:** no variables passed into a function can have their values changed. Take a situation where we might want to write a function to swap the values held in two variables

The Swap Function

```
void swap(int x, int y)
{
  // Create a temporary variable
  int temp;

  temp = x;
  x = y;
  y = temp;
}
```

The Swap Program

```
// Prototype for our swap function
void swap(int x, int y);

void main()
{
  int FirstInt = 10;
  int SecondInt = 20;
  swap(FirstInt, SecondInt);
}
```

Remember there was a problem here!

The New Swap Program

```
void swap(int& x, int& y);

void main()
{
  int FirstInt = 10;
  int SecondInt = 20;
  // Call the swap() function
  swap(FirstInt, SecondInt);
}
```

The New Swap Function

```
void swap(int& x, int& y)
{
  // Create a temporary variable
  int temp;

  temp = x;
  x = y;
  y = temp;
}
```

The Swap Program with reference

```
void swap(int& x, int& y);

void main()
{
  int FirstInt = 10;
  int SecondInt = 20;
  // Call the swap() function
  swap(FirstInt, SecondInt);
}
```

The Swap Function with reference

```
void swap(int& x, int& y)
{
  // Create a temporary variable
  int temp;

  temp = x;
  x = y;
  y = temp;
}
```

The Swap Program with Pointers

```
#include <iostream.h>
// Prototype for our swap function
// Use pointers instead of references
void swap(int *x, int *y);

void main()
{ int FirstInt = 10;
  int SecondInt = 20;
  // We must pass the address of each
  //variable when we call the swap
  swap(&FirstInt, &SecondInt);
}
```

The Swap Function with Pointers

```
// The swap() function
void swap(int * x, int * y)
{
  // Create a temporary variable
  int temp;
  temp = * x;
 * x =  * y;
 *y = temp;
}
```

Indirect Addressing

The * is used in declaration is different to the * used in assignment.

Using the * to access the value stored in an addesss is called *indirect addressing*, or sometimes dereferencing the pointer.

Be Careful!!!

Note : by declaring a pointer as

```
void* ptr;
```

is declaring a general purpose pointer that can point to any data type irrespective of its type. It is possible to assign the address of an *int, float etc.,* to the same pointer variable, this eliminates the need to declare a separate pointer variable for each data type.

In Summary

```
int  v;  //defines variable v of type int
int* p;   //defines p as a pointer to int

p = &v;   //assigns address of variable v
          to pointer p
v = 3;    //assigns 3 to v
*p = 3;   //assigns 3 to v
```

`v = 3;` is normal assignment of a value into a variable

`*p = 3;` indirect addressing

Pointer Constants

- The array name `A` is the address where the system has decided to place the array, and it will stay at this address until the program terminates. `A` is a constant.

Pointers to arrays

```
void main()
{
 int j;
 int A[5] = {31,54,77,52,93};

 for (j=0; j < 5; j++)
     cout <<endl <<A[j];
}
```

Using Pointer Notation

```
#include <iostream.h>
void main()
 {int A[5] = { 31, 54, 77, 52, 93};

  for(int j=0; j<5; j++)
  {       // for each element,
       cout << endl << *(A + j);
            // print value
  }
}
```

Using Pointer Notation

- Remember that **the name of an array is its address.**
- The expression `A+j` is the address with something added to it.
- What does ***(A+3)** yield?
- **Why can't we write *(A++)?** Can we write 7++?
- The array name `A` is the address where the system has decided to place the array, and it will stay at this address until the program terminates. **`A` is a constant.**
- We can't increment an address but we can increment a pointer that holds an address - see following slide.

What does the following do?

```
#include <iostream.h>
void main()
{ int A[] = { 31, 54, 77, 52, 93 };
  int* ptrint = A;
  for(int j=0; j<5; j++)
      {
            cout << endl << *(ptrint++);
      }
 }
```

`Prints the values of the array. We can increment a pointer that holds an address. We cannot increment an address itself.`

Passing by Reference

```
void main()
{ void centimise(double&);
  double var = 10.0;
  cout <<endl<< 'var = '<<var<<'inches';
  centimise(var);
  cout <<endl<<'var = ' <<var<<'centimeters';
}

void centimise(double& v)
{
  v *= 2.54;          //v is the same as var
}
```

Using Pointers

```
void main()
{ void centimise(double*);
  double var = 10.0;
  cout << endl << 'var = '<<var<<'inches';
  centimise(&var);  //&var = address of var
  cout <<endl<<'var = '<<var<<'centimeters';
}

void centimise(double* ptr)
{
  *ptr    *= 2.54;       //*ptr = *ptr * 2.54;
}
```

Arrays & Functions

- When passing arrays to functions as parameters they always act as if they were passed as reference.
- To declare a function that takes an array as a parameter we use the square brackets after the parameter tag as follows:
  `int Fname(int Array[], int ArrayCount);`

```
#include <iostream.h>
void InitArray(int A[], int ArrayCount);
  //Function Prototype

void main()
{
 int A[10];
 InitArray(A, 10); //Function call

 for (int i = 0; i < 10; i++)
  {cout <<"Element at: "<< i << "= ";
   cout << A[i] << endl;
  }
}
```

Function definition

```
void InitArray(int A[], int ArrayCount)
{
 for (int i = 0; i < ArrayCount; i++)
  {
   A[i] = 0;
  }
}
```

Passing Arrays by Pointer

```
#include <iostream.h>
void InitArray(int*, int); //Prototype
void main()
{
 int A[10];
 InitArray(A, 10); //Function call
 for (int i = 0; i < 10; i++)
  {cout <<"Element at: "<< i << "= ";
   cout << A[i] << endl;
  }
}
```

Declaring the function

```
void InitArray(int *A, int ArrayCount)
{
 for (int i = 0; i < ArrayCount; i++)
  {
   *(A+i) = 0;
  }
}
```

Pointers to Structures

- To declare a structure using pointers we use the keyword new
  - Declare a pointer to a structure
- To reference the fields we use the arrow (->) notation i.e.
  pointername -> structure part

An Example of pointers and structures

```
#include <iostream.h>

struct Complex
{
  int real;
  char imaginary;
};
void GetComplex(Complex *pCX);
void PrintComplex(Complex CX);
```

Takes a pointer to the structure as its parameter

```
void main()
{ Complex MyComplex;
  PrintComplex(MyComplex);
  GetComplex(&MyComplex);
}
void PrintComplex(Complex CX)
{
  cout << "Complex Number = ";
  cout << CX.real;
  cout << CX.imaginary << endl;
}
```

Pass in the address

GetComplex

```
void GetComplex(Complex *pCX)
{ int realPart;
  char imaginaryPart;
  cout << "Enter real part : ";
  cin >> realPart;
  cout << "Enter imaginary part : ";
  cin >> imaginaryPart;
  pCX->real = realPart;
  pCX->imaginary = imaginaryPart;
}
```

Using pointers => arrows instead of dots

Strings.

- Strings in C++ are represented as:
  - arrays of type char
  - pointers of type char

Arrays of type char.

- **char astring[10]**
- This declares a string of length 9. Note that all strings must be terminated by the symbol '/0'.
- Example: H e l l o /0

- If /o is not used to terminate a string, any attempt at displaying or copying the string will cause problems as the program will display or copy all the memory locations after the string until a /0 is encountered.

- **Initializing Character Arrays.**

  char astring[] = hello

  We cannot assign a string literal to a character array after the string has been declared - it can only be done one character at a time.(See next slide)

```
void main()
{char astring[4];
astring = 'abc';  // will NOT compile
astring[0] = 'a';
astring[1] = 'b';
astring[2] = 'c';
}
```

We cannot copy a string as follows: bstring = astring;

We have to use #include <string.h> where **strcpy** and **strncpy** are defined.

strcpy(dest, source);
strcpy(bstring, astring); // copies string including /0

strncpy(dest, source, stringlength);
strcpy(bstring, astring,9); // If bstring is of length 10

bstring[9] = '/0'; may be explicitly added.

```
include<iostream.h>
include<string.h>
void main()
{char astring[20];
   char bstring[10];
   cout << " Enter a string (up
   to    19 characters):";
   cin >> astring;
   cout <<astring<<endl;

   strncpy(bstring,astring, 9);
   bstring[9] = '/0';
   cout << bstring<<endl;
}
```

This program runs as follows:

Enter a string (up to 19 characters):
Superconductivity
Superconductivity
Supercond

Using pointers to strings.

- Using arrays as strings has drawbacks as arrays are difficult to manipulate and cannot be returned as a return value from a function.
- A complementary approach is to use a pointer to reference a string of characters.
- To declare a pointer to a string it must be of type char:
  char string_pointer*
- This declares a pointer of type char called string_pointer which is able to point to the first character of a string of characters.

- We can use char pointer to reference an array of characters quite easily:

```
void main()
{
   char string[11] = "char array";
   char* string_pointer = string;
   cout<<string_pointer;
}
```

- The assignment operator directs the pointer to the address of the first character in the array, and the program displays the text 'char array' using the pointer.

- It is also easy to assign literals at declaration, similar to arrays:
  char* string_pointer= "A string";
- The assignment operator may be used at any time after declaration to make one char pointer equal to another.
- Attempts to assign one array to another, or to assign a string literal to an array after declaration will lead to compiler errors. Both of these operators are legal with pointers.

```
void main()
{
   char a[10] = "spring";
   char b[10] = "summer";
   char* c = "autumn";
   char* d = “winter”;

   a =  b;                 //won't compile
   a =  "seasons"          //won't compile
   c =  d;                 //will compile
   c = "fall";             //will compile
}
```

Copying pointers to strings

- Note that using the assignment operator with two pointers will not copy the data itself but will copy the address of the data.
- Hence a = b above leaves the 2 pointers pointing to the same memory address.
- It is possible to use strcpy with character pointers to copy data rather than addresses.
  strcpy(a,b)

What's the error?

```
void main()
{   char str1[] = "Defined as an array";
    char* str2 = "Defined as a pointer";

    cout << endl << str1;
    cout << endl << str2;

    str1++; str2++;
    cout << endl << str2;
}
```

What's the output?

The Answers

- **The error:**
  - str1++;

    // can't do this; str1 is a constant
  - str2++;

    // this is OK, str2 is a pointer
- **The output:**

```
Defined as an array
Defined as a pointer
efined as an array
```

Strings as arguments

```
void main()
   {
   void dispstr(char*);
   char str[] = "Strings and pointers ";
   dispstr(str);
   }
void dispstr(char* ps)
   {  cout << endl;
      while( *ps )
      cout << *ps++;
   }
```

Strings as arguments

```
// displays a string with pointer notation
void main()
   {
   void dispstr(char*);      // prototype
   char str[] = "Strings and pointers ";
   dispstr(str);             // display the string
   }

void dispstr(char* ps)
   {  cout << endl;
      while( *ps )           // until null character,
      cout << *ps++;         // print characters
   }
```

A copy program

```
// copies one string to another with pointers
void main()
   {
   void copystr(char*, char*);
   char* str1 = "";
   char str2[80];

   copystr(str2, str1);      // copy str1 to str2
   cout << endl << str2;
   }
```

How would we write the copystr function using pointers?

The copy function

```
void copystr(char* dest, char* src)
   {
   while( *src )        // until null character,
     *dest++ = *src++; // copy chars
     *dest = '\0';      // terminate dest
   }
```

Explain the following:

```
#include <iostream.h>
const int DAYS = 7;
void main()
{  char* arr[DAYS] = { "Sunday", "Monday",
  "Tuesday", "Wednesday", "Thursday","Friday",
  "Saturday"  };

   for(int j=0; j<DAYS; j++)
   {cout << arr[j] << endl;
   }
}
```

Explain the following:

```
#include <iostream.h>
const int DAYS = 7;
void main()
{  char* arr[DAYS] = { "Sunday", "Monday",
  "Tuesday", "Wednesday", "Thursday","Friday",
  "Saturday"  };

   for(int j=0; j<DAYS; j++)
   {cout << arr[j] << endl;
   }
}
```

The above strings are displayed!

The new operator

```
void main()
{  char* str = "Using new with arrays.";
   int len = strlen(str);
   char* ptr;
   ptr = new char[len+1];  // set aside
                             memory: string + '\0'

   strcpy(ptr, str);    // copy str to new
                             memory area ptr
   cout << endl << "ptr=" << ptr;
   delete[] ptr; }
```

Using Pointers to Objects:
A Distance Class

```
#include <iostream.h>
class Distance
   {
   private:
      int feet;
      float inches;
   public:
        void getdist();
        void showdist();
   };
```

```
void getdist()
  {
  cout << "\nEnter feet: ";
  cin >> feet;
  cout << "Enter inches: ";
  cin >> inches;
   }
 void showdist()
  // display distance
  {
  cout << "feet" << feet;
  cout << "inches" << inches;
  }
```

Using Pointers to Objects

```
void main()
{   Distance dist;      // create an Object dist      //
  define a named object
   dist.getdist();           // access object members
   dist.showdist();          //    with dot operator

   Distance* distptr;    //create a pointer to an object
  // pointer to Distance
   distptr = new Distance;  // points to new object
   distptr->getdist();      // access object members
   distptr->showdist();     //    with -> operator
 }
```

If we replaced `distptr->getdist();` **with** *`(*distptr).getdist();`* **would anything change?**

The Class Person

```
class person
   // class of persons
{   private:
       char name[40];
        // person's name

    public:
        void setName();
        // set the name
        void printName();
        // get the name
};
```

```
void setName()
{
    cout << "Enter name: ";
    cin >> name;
}

void printName()
{
 cout << "\n   Name is: "
         << name;
}
```

Arrays of Pointers to Objects

```
void main()
{ person* PPtr[100];   // array of pointers to persons
  int n = 0;           // number of persons in array
  char choice;
  do  {                              // put persons in array
      PPtr[n] = new person;         // make new object
      PPtr[n]->setName();          // set person's name
      n++;                             // count new person
      cout << "Enter another (y/n)? "; // enter another
      cin >> choice;                    // person?
      }
   while( choice=='y' );                // quit on 'n'
```

```
for(int j=0; j<n; j++)          // print names of
      {                         // all persons
      cout << "\nPerson number " << j+1;
      PPtr[j]->printName();
      }
   }
         // end main()
```