Overview to C++

C++ is a general-purpose, case-sensitive, free-form programming language that supports procedural, object-oriented, and generic programming.

C++ is regarded as a middle-level language, as it comprises a combination of both high-level and low-level language features.

C++ was developed by Bjarne Stroustrup of AT&T Bell Laboratories in the early 1980's, and is based on the C language. The "++" is a syntactic construct used in C (to increment a variable), and C++ is intended as an incremental improvement of C. Most of C is a subset of C++, so that most C programs can be compiled (i.e. converted into a series of low-level instructions that the computer can execute directly) using a C++ compiler.

C++ is a superset of C, and that virtually any legal C program is a legal C++ program.

Object-Oriented Programming

C++ fully supports object-oriented programming, including the four pillars of object-oriented development:

1. Encapsulation
2. Data hiding
3. Inheritance
4. Polymorphism

Use of C++

1. C++ is used by hundreds of thousands of programmers in essentially every application domain.
2. In Adobe Systems All major applications are developed in C++: Photoshop & ImageReady, Illustrator, Acrobat, InDesign, GoLive
3. C++ is widely used for teaching and research because it is clean enough for successful teaching of basic concepts.
4. Anyone who has used either an Apple Macintosh or a PC running Windows has indirectly used C++ because the primary user interfaces of these systems are written in C++.
5. Amazon.com, Facebook, Google, HP, IBM, Microsoft, Mozilla, Nokia & many more companies uses C++ language.

Structure of a Program.....

```
/* This Program prints Hello World on screen */
#include<iostream.h>
using namespace std;
int main (){
cout << "Hello World!"; return 0;}
```

1. **/* This program ... */**
   The symbols/* and*/ used for comment. This Comments are ignored by the compiler, and are used to provide useful information about program to humans who use it.
2. **#include<*iostream.h*>**
   This is a preprocessor command which tells compiler to include iostream.h file.
3. **using namespace std;**
   All the elements of the standard C++ library are declared within what is called a namespace, the namespace with the name std. So in order to access its functionality we declare with this expression that we will be using these entities. This line is very frequent in C++ programs that use the standard library.
4. **main()**
   C++ programs consist of one or more functions. There must be one and only one function called main. The brackets following the word main indicate that it is a function and not a variable.
5. **{ }**
   braces surround the body of the function, which may have one or more instructions/statements.
6. **Cout<<**
   it is a library function that is used to print data on the user screen.
7. **''Hello World''** is a string that will be displayed on user screen.
8. **;** a semicolon ends a statement.
9. **return 0;** return the value zero to the Operating system.

Variables

A variable in C++ is a name for a piece of memory that can be used to store information. There are many types of variables, which determines the size and layout of the variable's memory;

Variable Names

We can use any combination of letters and numbers for Variable and function names but it must start with a letter.
We can use Underscore (_) as a letter in variable name and can begin with an underscore But Identifiers beginning with an underscore are reserved, And identifiers beginning with an underscore followed by a lower case letter are reserved for file scope identifiers Therefore using underscore as starting letter is not desirable.

Akki and **akki** are different identifiers because upper and lower case letters are treated as different identifiers.

Variable Types

There are many 'built-in' data types in C.

1. short int -128 to 127 (1 byte)
2. unsigned short int 0 to 255 (1 byte)
3. char 0 to 255 or -128 to +127 (1 byte)
4. unsigned char 0 to 255 (1 byte)
5. signed char -128 to 127 (1 byte)
6. int -32,768 to +32,767 (2 bytes)
7. unsigned int 0 to +65,535 (2 bytes)
8. long int -2,147,483,648 to +2,147,483,647 (4 bytes)
9. unsigned long int 0 to 4,294,967,295 (4 bytes)
10. float single precision floating point (4 bytes)
11. double double precision floating point (8 bytes)
12. long double extended precision floating point (10 bytes)

Definition, Declaration & Initialization

Definition is the place where variable is created (allocated storage).

Declaration is a place where nature (type) of variable is stated, but no storage is allocated.

Initialization means assigning a value to the variable.

Variables can be declared many times, but defined only once. Memory space is not allocated for a variable while declaration. It happens only on variable definition.

Variable declaration Syntax
data_type variable_name;
example
int a, b, c;
char flag;

Variable initialization syntax
data_type variable_name = value;
example
int a = 50;
char flag = 't';

external and static
initialisation done once only.

auto and register
initialisation done each time block is entered.

external and static variables cannot be initialised with a value that is not known until run-time; the initialiser must be a constant expression.

A variable that has not been assigned a value is called an uninitialized variable. Uninitialized variables are very dangerous because they cause intermittent problems (due to having different values each time you run the program). This can make them very hard to debug.

Variables Scope

Variables Scope refers to where variables is declared.

It can be Inside a function or a block which is called local variables, In the definition of function parameters which is called formal parameters or Outside of all functions which is called global variables.

Global variables

Global variable are declared outside any functions, usually at top of program. they can be used by later blocks of code:

```
int g;     //global
int main(void)
{
g = 0;
}
```

Local variables

Variables that are declared inside a function or block are local variables. The scope of local variables will be within the function only. These variables are declared within the function and can't be accessed outside the function.

```
void main()
{
int g;     //local
g=2;
cout << g;
}
```

Constants

Constants refer to fixed values in the code that you can't change and they are called literals. Constants can be of any of the basic data types and can be divided into Integer literals, Floating-Point literals, Strings, Characters and Boolean Values.

Integer literals

An Integer literal can be a decimal, octal, or hexadecimal constant.
A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

Example

```
45     //decimal
0213    //octal
0x4b    //hexadecimal
```

Floating-point literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

Example

```
3.14159
314159E-5L
```

Boolean literals

There are two Boolean literals and they are part of standard C++ keywords:
A value of true representing true. ?
A value of false representing false.

You should not consider the value of true equal to 1 and value of false equal to 0.

Character literals

A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

Escape sequence & Meaning
There are several character escape sequences which can be used in place of a character constant or within a string.

\a alert (bell)
\b backspace
\f formfeed
\n newline
\r carriage return
\t tab
\v vertical tab
\ backslash
\? question mark
\' quote
\'' double quote
\ooo character specified as an octal number
\xhh character specified in hexadecimal

String literals

String literals are enclosed in double quotes. A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separate them using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

"hello, dear"
"hello, \ dear"
"hello, ""d""ear"

Defining Constants

There are two ways in C++ to define constants:

1. Using #define preprocessor.
2. Using const keyword.

The #define Preprocessor:
Following is the form to use #define
preprocessor to define a constant:
#define identifier value

Example:

```
#include
using namespace std;
#define WIDTH 5
#define LENGTH 10
```

The const Keyword:
You can use const prefix to declare constants with a specific type as follows:
const type variable = value;

Example:

```
#include&ly;iostream>
using namespace std;
int main()
{
const int LENGTH =10;
const int WIDTH =5;
}
```

Variables Storage Classes

auto: The default class. Automatic variables are local to their block. Their storage space is reclaimed on exit from the block.

register: If possible, the variable will be stored in a processor register. May give faster access to the variable. If register storage is not possible, then the variable will be of automatic class.

Use of the register class is not recommended, as the compiler should be able to make better judgement about which variables to hold in registers, in fact injudicious use of register variables may slow down the program.

static: On exit from block, static variables are not reclaimed. They keep their value. On re-entry to the block the variable will have its old value.

extern: Allows access to external variables. An external variable is either a global variable or a variable defined in another source file. External variables are defined outside of any function. (Note: Variables passed to a function and modified by way of a pointer are not external variables)

static external: External variables can be accessed by any function in any source file which make up the final program. Static external variables can only be accessed by functions in the same file as the variable declaration.

Operators

An operator is a symbol. Compiler identifies Operator and performs specific mathematical or logical operation. C provides following operators:

1. Arithmetic Operators
2. Logical Operators
3. Increment and Decrement Operators
4. Relational Operators
5. Cast Operators
6. Bitwise Operators
7. Assignment Operators
8. Misc

Arithmetic Operators

1. ***** multiplication
2. **/** division
3. **%** remainder after division (modulo arithmetic)
4. **+** addition
5. **-** subtraction and unary minus

Logical Operators

&& Called Logical AND operator. If both the operands are non-zero, then condition becomes true.

|| Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.

! Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.

Increment and Decrement Operators

Increment and decrement operators are used to add or subtract 1 from the current value of oprand.

1. **++** increment
2. **--** decrement

Increment and Decrement operators can be prefix or postfix. In the prefix style the value of oprand is changed before the result of expression and in the postfix style the variable is modified after result.

For eg.

```
a = 9;
b = a++ + 5; /* a=10 b=14 */
a = 9;
b = ++a + 5; /* a=10 b=15 */
```

Relational Operators

1. **==** equal.
2. **!=** Not equal.
3. **> <** Greater than/less than
4. **>=** greater than or equal to
5. **<=** less than or equal to

Cast Operators

Cast operators are used to convert a value from one to another type.

(float) sum; converts type to float

(int) fred; converts type to int

Bitwise Operators

Bitwise operators performs operation on actual bits present in each byte of a variable. Each byte contain 8 bits, each bit can store the value 0 or 1

1. **~** one's complement
2. **&** bitwise AND
3. **^** bitwise XOR
4. **|** bitwise OR
5. **<<** left shift (binary multiply by 2)
6. **>>** right shift (binary divide by 2)

Assignment Operators

1. **=** assign
2. **+=** assign with add
3. **-=** assign with subtract
4. ***=** assign with multiply
5. **/=** assign with divide
6. **%=** assign with remainder
7. **>>=** assign with right shift
8. **<<=** assign with left shift
9. **&=** assign with bitwise AND
10. **^=** assign with bitwise XOR
11. **|=** assign with bitwise OR

For example,
a = a + 64; is same
as a += 64;

Misc

sizeof
The sizeof operator returns the size, in bytes, of a type or a variable.
You can compile and run the following program to find out how large your data types are: cout << "bool:\t\t" << sizeof(bool) << " bytes"; bool: 1 bytes

Condition ? X : Y
Condition operator: If Condition is true ? then it returns value X : otherwise value Y

,

Comma operator causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list .

(dot) and -> (arrow)

Member operators are used to reference individual members of classes, structures, and unions.

&

Pointer operator: & returns the address of an variable. **For example** &a; will give actual address of the variable.

*****

Pointer operator: * is pointer to a variable. For example *var; will pointer to a variable var.

Arrays

Array is a fixed size collection of similar data type items. Arrays are used to store and access group of data of same data type.
Arrays can of any data type.Arrays must have constant size. Continuous memory locations are used to store array.
It is an aggregate data type that lets you access multiple variables through a single name by use of an index. Array index always starts with 0.

Example for Arrays:
int a[5]; // integer array char
a[5]; // character(string) array
In the above example, we declare an array named a. When used in an array definition, the subscript operator ([]) is used to tell the compiler how many variables to allocate. In this case, we're allocating 5 integers/character. Each of these variables in an array is called an element.

Types of Arrays:

1. One Dimensional Array
2. Two Dimensional Array
3. Multi Dimensional Array

One Dimensional Array

Array declaration
int age [5];

Array initialization
int age[5]={0, 1, 2, 3, 4, 5};

Accessing array
age[0]; /0_is_accessed/

age[1]; /1_is_accessed/
age[2]; /2_is_accessed/ 2

Two Dimensional Array

Two dimensional array is combination of rows n columns.
Array declaration
int arr[2][2];

Array initialization
int arr[2][2] = {{1,2}, {3,4}};

Accessing array
arr [0][0] = 1;
arr [0][1] = 2;
arr [1][0] = 3;
arr [1][1] = 4;

Multi-dimensional Array

C++ programming language allows programmer to create arrays of arrays known as multidimensional arrays.
For example:
float a[2][4][3];

Pointer to an array

Please go through pointers chapter first to understand this
An array name is a constant pointer to the first element of the array. Therefore, in the declaration:
double balance[50];

balance is a pointer to &balance[0], which is the address of the first element of the array balance. Thus, the following program fragment assigns p the address of the first element of balance:
double *p;

```
double balance[10];
p = balance;
```

It is legal to use array names as constant pointers, and vice versa. Therefore, *(balance + 4) is a legitimate way of accessing the data at balance[4].

Passing Array To Function

We can pass entire Arrays to functions as an argument.

For eg.

```
#include void display(int a)
{
int i;
for(i=0;i < 4;i++)
{
cout << a[i];
}
}
int main()
{
int c[]={1,2,3,4};
display(c); //Passing array to display.
return 0;
}
```

Return array from functions

C++ does not allow to return an entire array as an argument to a function. However, You can return a pointer to an array by specifying the array's name without an index.

If you want to return a single-dimension array from a function, you would have to declare a function returning a pointer as in the following **example:**

```
int * myFunction(){
int c[]={1,2,3}
.

return c}
```

Strings

A string is simply an array of characters which is terminated by a null character '\0' which shows the end of the string.
Strings are always enclosed by double quotes. Whereas, character is enclosed by single quotes in C.

This declaration and initialization create a string with the word "AKKI".

To hold the \0 (null character) at the end of the array, the size of array is one more than the number of characters in the word "AKKI".

char my_name[5] = {'A', 'K', 'K', 'I','\0'};

we can also write the above statement as follows:
char my_name[] = "AKKI";

C String functions

To use string functions programmer must import String.h header file.
String.h header file supports all the string functions in C language.

All the string functions are given below.

a. **strcat( )**
   Concatenates str2 at the end of str1.
b. **strcat( )**
   Appends a portion of string to another .
c. **strcpy( )**
   Copies str2 into str1 .
d. **strncpy( )**
   Copies given number of characters of one string to another .
e. **strlen( )**
   Gives the length of str1.

f. **strcmp( )**
    a. Returns 0 if str1 is same as str2.
    b. Returns <0 if strl < str2.
    c. Returns >0 if str1 > str2.

g. **strcmpi( )**
   Same as strcmp() function. But, this function negotiates case. ''A'' and ''a'' are treated as same.

h. **strchr( )**
   Returns pointer to first occurrence of char in str1.

i. **strrchr( )**
   Last occurrence of given character in a string is found .

j. **strstr( )**
   Returns pointer to first occurrence of str2 in str1.

k. **strrstr( )**
   Returns pointer to last occurrence of str2 in str1.

l. **strdup( )**
   Duplicates the string .

m. **strlwr( )**
   Converts string to lowercase .

n. **strupr( )**
   Converts string to uppercase .

o. **strrev( )**
   Reverses the given string .

p. **strset( )**
   Sets all character in a string to given character .

q. **strnset( )**
   It sets the portion of characters in a string to given character .

r. **strtok( )**
   Tokenizing given string using delimiter.

Pointers

Pointer is a variable that stores the address of another variable. They can make some things much easier, help improve your program's efficiency, and even allow you to handle unlimited amounts of data.

Pointer is used to allocate memory dynamically i.e. at run time. The variable might be any of the data type such as int, float, char, double, short etc.

Syntax :

To declare a pointer, we use an asterisk between the data type and the variable name Pointers require a bit of new syntax because when you have a pointer, you need the ability to both request the memory location it stores and the value stored at that memory location. data_type *ptr_name;

Example :

int *a; char *a;
Where, * is used to denote that ''a'' is pointer variable and not a normal variable. In this context, the asterisk is not a multiplication

Key points to remember about pointers:

1. Normal variable stores the value whereas pointer variable stores the address of the variable.
2. The content of the pointer always be a whole number i.e. address.
3. Always pointer is initialized to null, i.e. int *p = null.
4. The value of null pointer is 0. # & symbol is used to get the address of the variable.
5. * symbol is used to get the value of the variable that the pointer is pointing to.
6. If pointer is assigned to NULL, it means it is pointing to nothing.

7. Two pointers can be subtracted to know how many elements are available between these two pointers.
8. But, Pointer addition, multiplication, division are not allowed.
9. The size of any pointer is 2 byte (for 16 bit compiler).

Since pointers only hold addresses, when we assign a value to a pointer, the value has to be an address. To get the address of a variable, we can use the address-of operator (&)

Example program for pointer:

```
#include<iostream.h>
int main()
{
int *ptr, q;
q = 50;
/* address of q is assigned to ptr */
ptr = &q;
// prints address held in ptr, which is &q
cout << ptr;
/* display q's value using ptr variable */
cout << *ptr;
return 0;
}
```

The null pointer

Sometimes we need to make our pointers point to nothing. This is called a null pointer. We assign a pointer a null value by setting it to address 0: int
*ptr;
ptr = 0;
// assign address 0 to ptr or simply
int *ptr = 0;
// assign address 0 to ptr

C++ Pointer Arithmetic

As you understood pointer is an address which is a numeric value; therefore, you can perform arithmetic operations on a pointer just as you can a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -.

Example:

```
ptr++;
ptr--;
ptr+21;
ptr-10;
```

If a char pointer pointing to address 100 is incremented (ptr++) then it will point to memory address 101

C++ Pointers vs Arrays

Pointers and arrays are strongly related. In fact, pointers and arrays are interchangeable in many cases. For example, a pointer that points to the beginning of an array can access that array by using either pointer arithmetic or array-style indexing.

```
int main()
{
int var[3] = {1, 2, 3};
int *ptr;
cout << *ptr << endl;
ptr++;
cout << *ptr << endl;
return 0;
}
```

this code will return :

```
1
2
```

C++ Pointer to Pointer

A pointer to a pointer is a form of multiple indirection or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value.

```
int main()
{
int var;
int *ptr;
int **pptr;
var = 3000;
ptr = &var;
pptr = &ptr;
cout << "Value of var :" << var << endl;
cout << "Value available at *ptr :" << *ptr << endl;
cout << "Value available at **pptr :" << **pptr << endl;
return 0;
}
```

this code will return

```
Value of var :3000
Value available at *ptr :3000
Value available at **pptr :3000
```

Structure

In Array we can store data of one type only, but structure is a variable that gives facility of storing data of different data type in one variable.

Structures are variables that have several parts; each part of the object can have different types. Each part of the structure is called a member of the structure.

Declaration

Consider basic data of student : roll_no, class, name, age, address.

A structure data type called student can hold all this information:

```
struct student
{
int roll_no;
char class ;
char name[25];
int age;
char address[50];
};
```

before the final semicolon, At the end of the structure's definition, we can specify one or more structure variables.

There is also another way of declaring variables given below,
struct student s1;

Initialization

Structure members can be initialized at declaration. This is same as the initialization of arrays; the values are listed inside braces. The structure declaration is preceded by the keyword static.

```
static struct student akki ={1234,"comp","akki",20,"mumbai"};
```

Accessing structure data

To access a given member the dot notation is use. The "dot" is called the member access operator

```
struct student s1;
s1.name = "Akki";
s1.roll_no = 1234 ;
```

scope

A structure type declaration can be local or global, depending upon where the declaration is made.

Structures as Function Arguments

You can pass a structure as a function argument in very similar way as you pass any other variable or pointer. You would access structure variables in the similar way as you have accessed in the above example

Pointers to Structures

You can define pointers to structures in very similar way as you define pointer to any other variable as follows:

```
struct student *struct_pointer;
```

Reference

References are another variable type of C++ that act as an alias or short name to another variable. A reference variable acts just like the original variable it is referencing. References are declared by using an ampersand (&) between the reference type and the variable name.

Declaration

```
int distance = 5;
// normal integer
int &ref = distance;
// reference to distance
```

The ampersand in this context does not mean “address of”, it means “reference to”.

Use of reference

```
distance = 6;
//distance is now 6
ref = 7;
//distance is now 7

cout << distance;
// prints 7
distance++;
cout << ref;
//prints 8
```

Using the address-of operator on a reference returns the address of the value being referenced:

```
cout << &distance;
//prints 0012FF7C
```

```
cout << &ref;
//prints 0012FF7C
```

Const references

It is possible to declare a const reference. A const reference will not let you change the value it references:

```
int distance = 5;
const int &ref = distance;
ref = 6;   // illegal -- ref is const
```

Enumeration and Typedef

C++ allows us to create own own data types. enumerated type is the simplest method for doing so.
An enumerated type is a data type where every possible value is defined as a symbolic constant (called an enumerator). enum keyword is used to declare Enumerated types.

Example:

```
//define a new enum named Brand
enum Brand
{
//Here are the enumerators
//These define all the possible values this type can hold
Brand_LEE,
Brand_Reebok,
Brand_Cooper,
Brand_Fila
};

//Declare a variable of enumerated type Brand
Brand a_Brand = Brand_Fila;
```

Memory is not allocated while defining an enumerated type.
When a variable of the enumerated type is declared (such as a_Brand in the example above), memory is allocated for that variable at that time.

Enum variables are the same size as an int variable.
This is because each enumerator is automatically assigned an integer value based on it’s position in the enumeration list.
By default, the first enumerator is assigned the integer value 0, and each subsequent enumerator has a value one greater than the previous enumerator:

```
enum Brand
{
Brand_LEE,    // assigned 0
Brand_Reebok,    // assigned 1
Brand_Cooper,    // assigned 2
Brand_Fila    // assigned 3
};

Brand a_Brand= Brand_Fila;
cout << a_Brand;
```

The cout statement above prints the value 3.

We can explicitly define the value of enumerator. Value of enumerator can be positive or negative and can be non-unique. Any non-defined enumerators are given a value one greater than the previous enumerator.

```
//define a new enum named Brand
enum Brand
{
Brand_LEE = -3,
Brand_Reebok,    // assigned -2
Brand_Cooper,    // assigned -1
Brand_Fila = 5,
Brand_Nike    // assigned 6
};
```

typedef

Typedefs allow the programmer to create an alias for a data type, and use the aliased name instead of the actual type name. To declare a typedef, simply use the typedef keyword, followed by the type to alias, followed by the alias name:

```
typedef long Binary;
//define Binary as an alias for long
```

//The following two statements are equivalent: long number;
Binary number;

A typedef does not define new type, but is just another name for an existing type. A typedef can be used anywhere a regular type can be used.

Typedefs are used mainly for documentation and legibility purposes. Data type names such as char, int, long, double, and bool are good for describing what type of variable something is, but more often we want to know what purpose a variable serves.

In the above example, long number does not give us any clue what type number is holding. Is it Binary, Hexadecimal, Octal or some other type? Binary number makes it clear what the type of number is.

OUTPUT STREAM (cout)

On most program environments, the standard output by default is the screen, and the C++ stream object defined to access it is cout. cout is an instance of iostream class.

For formatted output operations, cout is used together with the insertion operator, which is written as << (i.e., two "less than" signs).

```
cout << "this is Output";         //prints this is Output sentence on screen
cout << 50;           //prints number 50 on screen
cout << x;             //prints the value of x on screen
```

The << operator inserts the data that follows it into the stream that precedes it. In the examples above, it inserted the literal string Output sentence, the number 120, and the value of variable x into the standard output stream cout. Notice that the sentence in the first statement is enclosed in double quotes (") because it is a string literal, while in the last one, x is not. The double quoting is what makes the difference; when the text is enclosed between them, the text is printed literally; when they are not, the text is interpreted as the identifier of a variable, and its value is printed instead.

For example,

These two sentences have very different results:

```
cout << "Hello";
//prints Hello
cout << Hello;
//prints the content of variable Hello
```

Multiple insertion operations (<<) may be chained in a single statement:

```
cout << "This " << " is a " << "single C++ statement";
```

This last statement would print the text This is a single C++ statement. Chaining insertions is especially useful to mix literals and variables in a single statement:

```
cout << "I am " << age << " years old and my zipcode is " << zipcode;
```

INPUT STREAM (cin)

In most program environments, the standard input by default is the keyboard, and the C++ stream object defined to access it is cin. cin is an instance of iostream class

For formatted input operations, cin is used together with the extraction operator, which is written as >> (i.e., two "greater than" signs). This operator is then followed by the variable where the extracted data is stored.
For example:

```
int age;
//declares a variable of type int called age
cin >> age;
//extracts a value to be stored in it
```

This operation makes the program wait for input from cin; generally, this means that the program will wait for the user to enter some sequence with the keyboard.

Extractions on cin can also be chained to request more than one datum in a single statement:

```
cin >> a >> b;
```

This is equivalent to:

```
cin >> a;
cin >> b;
```

In both cases, the user is expected to introduce two values, one for variable a, and another for variable b. Any kind of space is used to separate two consecutive input operations; this may either be a space, a tab, or a new-line character.

ERROR STREAM (cerr)

The predefined object cerr is an instance of iostream class. The cerr object is said to be attached to the standard error device, which is also a display screen but the object cerr is un-buffered and each stream insertion to cerr causes its output to appear immediately.

The cerr is also used in conjunction with the stream insertion operator as shown in the following example.

```
int main( ){
char str[] = "Unable to read....";
cerr << "Error message : " << str << endl;}
```

When the above code is compiled and executed, it produces the following result:

```
Error message : Unable to read....
```

LOG STREAM (clog)

The predefined object clog is an instance of ostream class. The clog object is said to be attached to the standard error device, which is also a display screen but the object clog is buffered. This means that each insertion to clog could cause its output to be held in a buffer until the buffer is filled or until the buffer is flushed. The clog is also used in conjunction with the stream insertion operator as shown in the following example.

```
int main( ){
char str[] = "Unable to read....";
clog << "Error message : " << str << endl;}
```

When the above code is compiled and executed, it produces the following result:

```
Error message : Unable to read....
```

You would not be able to see any difference in cout, cerr and clog with these small examples, but while writing and executing big programs then difference becomes obvious. So this is good practice to display error messages using cerr stream and while displaying other log messages then clog should be used.

IF-statement

if statement

An if statement contains a Boolean expression and block of statements enclosed within braces.

Structure of if statement

```
if (boolean expression )
/* if expression is true */
statements... ; /* Execute statements */
```

If the Boolean expression is true then statement block is executed otherwise (if false) program directly goes to next statement without executing Statement block.

if...else statement

If statement block with else statement is known as as if...else statement. Else portion is non-compulsory.

Structure of if...else

```
if(condition)
{
statements...
}
else
{
statements...
}
```

If the condition is true, then compiler will execute the if block of statements, if false then else block of statements will be executed.

Nested if...else statement

We can use multiple if-else for one inside other this is called as Nested if-else.

Structure of Nested if...else

```
if(condition)
{
statements...
}
else if
{
statements...
}
else
{
statements...
}
```

Switch Statement

A switch statement is used instead of nested if...else statements. It is multiple branch decision statement of C++. A switch statement tests a variable with list of values for equivalence. Each value is called a case. The case value must be a constant integer.

Structure of switch() statement

```
switch (expression)
{
case value: statements...
case value: statements...
default : statements...
}
```

Individual case keyword and a semi-colon (:) is used for each constant. Switch tool is used for skipping to particular case, after jumping to that case it will execute all statements from cases beneath that case this is called as ''Fall Through''.

In the example below, for example, if the value 2 is entered, then the program will print **two one something else!**

```
int main()
{
int i;
cout << ''Enter an integer: '';
cin>>i;
switch(i)
{
case 4: cout << ''four''; break;
case 3: cout << ''three''; break;
case 2: cout << ''two '';
case 1: cout << ''one '';
default: cout << ''something else!'';
}
return 0;
}
```

To avoid fall through, the break statements are necessary to exit the switch. If value 4 is entered, then in case 4 it will just print four and ends the switch.

The default label is non-compulsory, It is used for cases that are not present.

Loops

while loop

The while loop calculates the expression before every loop. If the expression is true then block of statements is executed, so it will not execute If the condition is initially false. It needs the parenthesis like the if statement.

```
while(expression)
/* while expression is true do following*/
statements... ;
```

do While loop

This is equivalent to a while loop, but it have test condition at the end of the loop. The Do while loop will always execute at least once.

```
do
statements ;
while ( expression );
/* while expression is true do...*/
```

for loop

This is very widely held loop.
For loops work like the corresponding while loop shown in the above example. The first expression is treated as a statement and executed, then the second expression is test or condition which is evaluated to see if the body of the loop should be executed. The third expression is increment or decrement which is performed at the end of every iteration/repetition of the loop.

```
for (expr1; expr2; expr3)
statements...;
```

In while loop it can happen that the statement will never execute But In the do-while loop, test condition is based at the end of loop therefore the block of statement will always execute at least once. This is the main difference between the while and the do-while loop.

For example, to execute a statement 5 times:

```
for (i = 0; i < 5; i++)
{
cout << i << endl;
}
```

Another way of doing this is:

```
i = 5;
while (i--)
statements;
```

While using this method, make sure that value of i is greater than zero, or make the test i-->0.

Break and Continue

Break statement

Break statement is usually used to terminate a case in the switch statement. Break statement in loops to instantly terminates the loop and program control goes to the next statement after the loop.

If break statement is used in nested loops (i.e., loop within another loop), the break statement will end the execution of the inner loop and Program control goes back to outer loop.

Syntax :
break;

Continue statement

In C++ programming language the continue statement works slightly similar to the break statement. The continue restarts the loop with the next value of item. All the line code below continue statement is skips.

Syntax : continue;
In the for loop, continue statement skips the test condition and increment value of the variable to execute again and In the while and do...while loops, continue skips all the statements and program control goes to at the end of loop for tests condition.

Random Numbers

C (and by extension C++) comes with a built-in pseudo-random number generator. It is implemented as two separate functions that live in the cstdlib header:
srand() sets the initial seed value. srand() should only be called once.
rand() generates the next random number in the sequence (starting from the seed set by srand()).

Here's a sample program using these functions:

```
int main()
{
int count;
srand(5323);
// set initial seed value to 5323

// Print 10 random numbers
for (i=0; i < 10; i++)
{
cout << rand() << "\t";
}
}
```

The output of this program will be any 10 numbers.

The range of rand()

rand() generates pseudo-random integers between 0 and RAND_MAX, a constant in stdlib that is typically set to 32767.

Generally, we do not want random numbers between 0 and RAND_MAX — we want numbers between two other values, which we'll call Low and High. **For example,** if we're trying to simulate the user rolling a dice, we want random numbers between 1 and 6.

It turns out it’s quite easy to take the result of rand() can convert it into whatever range we want:

```
// Generate a random number between Low and High (inclusive)
unsigned int GetRandomNumber(int Low, int High)
{
return (rand() % (High - Low + 1)) + Low;
}
```

Functions

A function is a group of statements that together perform a task. All C programs made up of one or more functions. There must be one and only one main function.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so each function performs a specific task.

A program will be executing statements sequentially inside one function when it encounters a function call.

A function call is an expression that tells the CPU to interrupt the current function and execute another function.

The CPU 'puts a bookmark' at the current point of execution, and then calls (executes) the function named in the function call. When the called function terminates, the CPU goes back to the point it bookmarked, and resumes execution.

Function definition

```
return_type function_name( parameter list )
{
//statements
}
```

1. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value.
2. **function name** is the identifier by which the function can be called.
3. A **parameter** is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
4. **statements** is the function's body. It is a block of statements surrounded by braces { } that specify what the function actually does.

Example

```
/* Function returning addition of 2 integers */
int add(int a, int b)
{
int total ;
total= a+b;
return total;
}
```

Declaration, Call and Argument

Function Declaration

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:
return type function_name(parameter list);

For our defined function add() in previous chapter, following is the function declaration:
int add(int a, int b);

Parameter names are not important in function declaration only their type is required, so following is also valid declaration:
int add(int, int);

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Function Call

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value.

For example:

```
int main(){
int a = 100;
int b = 200;
int ret;
// calling a function to get addition.
ret = add(a, b);
cout << "addition is : " << ret << endl;
return 0;}
```

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.
The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function:

Call by value
This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

Call by pointer
This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Call by reference
This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Inline Function

Calling a function generally causes a certain overhead (stacking arguments, jumps, etc...), and thus for very short functions, it may be more efficient to simply insert the code of the function where it is called, instead of performing the process of formally calling a function.

Preceding a function declaration with the inline specifier informs the compiler that inline expansion is preferred over the usual function call mechanism for a specific function. This does not change at all the behavior of a function, but is merely used to suggest the compiler that the code generated by the function body shall be inserted at each point the function is called, instead of being invoked with a regular function call.

For example, the concatenate function above may be declared inline as:

```
inline string concatenate (const string& a, const string& b)
{
return a+b;
}
```

This informs the compiler that when concatenate is called, the program prefers the function to be expanded inline, instead of performing a regular call. inline is only specified in the function declaration, not when it is called.

Note that most compilers already optimize code to generate inline functions when they see an opportunity to improve efficiency, even if not explicitly marked with the inline specifier. Therefore, this specifier merely indicates the compiler that inline is preferred for this function, although the compiler is free to not inline it, and optimize otherwise. In C++, optimization is a task delegated to the compiler, which is free to generate any code for as long as the resulting behavior is the one specified by the code.

Recursion

Recursion is a programming technique that allows the programmer to express operations in terms of themselves.

In C++, this takes the form of a function that calls itself. A useful way to think of recursive functions is to imagine them as a process being performed where one of the instructions is to "repeat the process". This makes it sound very similar to a loop because it repeats the same code, and in some ways it is similar to looping. On the other hand, recursion makes it easier to express ideas in which the result of the recursive call is necessary to complete the task. Of course, it must be possible for the "process" to sometimes be completed without the recursive call.

```
void CountDown(int nValue)
{
using namespace std;
cout << nValue << endl;
CountDown(nValue-1);
}

int main(void)
{
CountDown(10);
return 0;
}
```

When CountDown(10) is called, the number 10 is printed, and CountDown(9) is called. CountDown(9) prints 9 and calls CountDown(8). CountDown(8) prints 8 and calls CountDown(7). The sequence of CountDown(n) calling CountDown(n-1) is continually repeated, effectively forming the recursive equivalent of an infinite loop.

This program illustrates the most important point about recursive functions: you must include a termination condition, or they will run “forever” (or until the call stack runs out of memory).

Stopping a recursive function generally involves using an if statement. Here is our function redesigned with a termination condition:

```
void CountDown(int nValue)
{
using namespace std;
cout << nValue << endl;

// termination condition
if (nValue > 0)
CountDown(nValue-1);
}

int main(void)
{
CountDown(10);
return 0;
}
```

Now when we run our program, CountDown() will count down to 0 and then stop!

Basics of Classes

Classes are an expanded concept of data structures: like data structures, they can contain data members, but they can also contain functions as members.
An object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Classes are defined using either keyword class or keyword struct, with the following syntax:

```
class class_name
{
access_specifier_1:
member1;
access_specifier_2:
member2;
...
} object_names;
```

Where class_name is a valid identifier for the class, object_names is an optional list of names for objects of this class. The body of the declaration can contain members, which can either be data or function declarations, and optionally access specifiers.

Classes have the same format as plain data structures, except that they can also include functions and have these new things called access specifiers. An access specifier is one of the following three keywords: private, public or protected. These specifiers modify the access rights for the members that follow them:
private members of a class are accessible only from within other members of the same class (or from their "friends").
protected members are accessible from other members of the same class (or from their "friends"), but also from members of their derived classes.
Finally, **public** members are accessible from anywhere where the object is visible.

By default, all members of a class declared with the class keyword have private access for all its members. Therefore, any member that is declared before any other access specifier has private access automatically.

Defining C++ Objects

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box: Box Box1;
//Declare Box1 of type Box
Box Box2;
//Declare Box2 of type Box

Both of the objects Box1 and Box2 will have their own copy of data members

Constructor-Destructor

Constructor

A constructor is a special kind of class member function that is executed when an object of that class is instantiated.
Constructors are typically used to initialize member variables of the class to appropriate default values, or to allow the user to easily initialize those member variables to whatever values are desired.

Unlike normal functions, constructors have specific rules for how they must be named:

1. Constructors should always have the same name as the class (with the same capitalization)
2. Constructors have no return type (not even void)

A constructor that takes no parameters (or has all optional parameters) is called a default constructor but if you need, a constructor can have parameters. This helps you to assign initial value to an object at the time of its creation.

Destructor

A destructor is another special kind of class member function that is executed when an object of that class is destroyed. They are the counterpart to constructors. When a variable goes out of scope, or a dynamically allocated variable is explicitly deleted using the delete keyword, the class destructor is called (if it exists) to help clean up the class before it is removed from memory.

For simple classes, a destructor is not needed because C++ will automatically clean up the memory for you. However, if you have dynamically allocated memory, or if you need to do some kind of maintenance before the class is destroyed (eg. closing a file), the destructor is the perfect place to do so.

Like constructors, destructors have specific naming rules:

1. The destructor must have the same name as the class, preceded by a tilde (~).
2. The destructor can not take arguments.
3. The destructor has no return type.

Note: Rule 2 implies that only one destructor may exist per class, as there is no way to overload destructors since they can not be differentiated from each other based on arguments.

Example

```
class abc
{
private :
int a,b;

public :
abc() //default constructor
{
a=0;
b=1;
}

abc(int x, int y) //Parametrized constructor
{
a=x;
b=y;
}

~abc(); //destructor
.
.
.
};
```

Friend Function

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword friend as follows:

```
class ABC
{
double a;
public:
double b;
friend void printWidth( ABC abc );
void setWidth( double c );
};
```

To declare all member functions of class XYZ as friends of class ABC, place a following declaration in the definition of class ABC:

```
friend class XYZ;
```

Inheritance

Inheritance is one of the key feature of object-oriented programming including C++ which allows user to create a new class(derived class) from a existing class(base class). The derived class inherits all feature from a base class and it can have additional features of its own.

Inheritance makes the code reusable. When we inherit an existing class, all its methods and fields become available in the new class, hence code is reused.
Note : All members of a class except Private, are inherited.

Purpose of Inheritance

1. Code Re usability
2. Method Overriding (Hence, Runtime Polymorphism.)
3. Use of Virtual Keyword

Basic Syntax of Inheritance

class Subclass_name : access_mode Superclass_name

While defining a subclass like this, the super class must be already defined or at least declared before the subclass declaration.
Access Mode is used to specify, the mode in which the properties of superclass will be inherited into subclass, public, private or protected.

Access Control and Inheritance

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

Access

1. same class can access all (public, private, protected) members.
2. derived class can access only public and protected members.
3. Outside classes can access only public members.

A derived class inherits all base class methods with the following exceptions:

1. Constructors, destructors and copy constructors of the base class.
2. Overloaded operators of the base class.
3. The friend functions of the base class.

Inheritance Visibility Mode

Depending on Access modifier used while inheritance, the availability of class members of Super class in the sub class changes. It can either be private, protected or public.

1) Public Inheritance

This is the most used inheritance mode. In this the protected member of super class becomes protected members of sub class and public becomes public. class Subclass : public Superclass

2) Private Inheritance

In private mode, the protected and public members of super class become private members of derived class. class Subclass : Superclass // By default its private inheritance

3) Protected Inheritance

In protected mode, the public and protected members of Super class becomes protected members of Sub class. class subclass : protected Superclass

Function Overloading

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You can not overload function declarations that differ only by return type.

```
// overloading functions
#include
using namespace std;

int add (int a, int b)
{
return (a+b);
}

double add (double a, double b)
{
return (a+b);
}

int main ()
{
int x=5,y=2;
double n=5.0,m=2.5;

cout << add(x,y) << '\n';
cout << add(n,m) << '\n';
return 0;
}
```

o/p:

```
7
7.5
```

In this example, there are two functions called add, but one of them has two parameters of type int, while the other has them of type double. The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. If it is called with two int arguments, it calls to the function that has two int parameters, and if it is called with two doubles, it calls the one with two doubles.

Note : A function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

Operator Overloading

Operator overloading is an important concept in C++. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type.

Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Almost any operator can be overloaded in C++. However there are few operator which can not be overloaded. Operator that are not overloaded are follows

1. scope operator - ::
2. sizeof
3. member selector - .
4. member pointer selector - *
5. ternary operator - ?:

Syntax?

```
Return_type class_name :: operator operator_symbol(argument_list)
{
//function body
}
```

here operator is keyword and operator_symbol is operator to be overloaded.

Implementing Operator Overloading

Operator overloading can be done by implementing a function which can be:

1) Member Function
2) Non-Member Function
3) Friend Function

Operator overloading function can be a member function if the Left operand is an Object of that class, but if the Left operand is different, then Operator overloading function must be a non-member function.
Operator overloading function can be made friend function if it needs access to the private and protected members of class.

Restrictions on Operator Overloading

Following are some restrictions to be kept in mind while implementing operator overloading.

a. Precedence and Associativity of an operator cannot be changed.
b. Arity (numbers of Operands) cannot be changed. Unary operator remains unary, binary remains binary etc.
c. No new operators can be created, only existing operators can be overloaded.
d. Cannot redefine the meaning of a procedure. You cannot change how integers are added.

Polymorphism and Virtual Functions

Polymorphism

"Poly" means "many" and "morph" means "form". Polymorphism is the ability of an object (or reference) to assume (be replaced by) or become many different forms of object. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.
Example: Function overloading, virtual functions. Another example can be a plus + sign, used for adding two integers or for using it to concatenate two strings.

Virtual Function:

A virtual function is a function in a base class that is declared using the keyword virtual. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as dynamic linkage, or late binding.

```
class Shape
{
protected:
int width, height;

public:
```

```
Shape( int a=0, int b=0)
{
width = a;
height = b;
}

// pure virtual function
virtual int area() = 0;
};
```

Important Points to Remember

1. Only the Base class Method's declaration needs the Virtual Keyword, not the definition.
2. If a function is declared as virtual in the base class, it will be virtual in all its derived classes.
3. The address of the virtual Function is placed in the VTABLE and the copiler uses VPTR(vpointer) to point to the Virtual Function.

Data Abstraction

Object Oriented Programming has a special feature called data abstraction. Data abstraction allows ignoring the details of how a data type is represented. While defining a class, both member data and member functions are described. However while using an object (that is an instance of a class) the built in data types and the members in the class are ignored. This is known as data abstraction. This can be seen from the above example.

Benefits of Data Abstraction

1. Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
2. The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data are public, then any function that directly accesses the data members of the old representation might be broken.

Example

```
#include
#include
class base
{
private: int s1,s2;
public: void inp_val()
{
cout <<"input the values of s1 and s2 ";
cin>>sl>>s2;
}
void display()
{
cout <<<"="" "<<s2<<"\n="" ";
 }
};
void main()
{
base b;
b.inp_val();
b.display();
}
```

Above class takes numbers from user, and returns both. The public members inp_val and display are the interfaces to the outside world and a user needs to know them to use the class. The private members s1 & s2 are something that the user doesn't need to know about, but is needed for the class to operate properly.

Data Encapsulation

Encapsulation is the method of combining the data and functions inside a class. This hides the data from being accessed from outside a class directly, only through the functions inside the class is able to access the information.

This is also known as "Data Abstraction", as it gives a clear separation between properties of data type and the associated implementation details. There are two types, they are "function abstraction" and "data abstraction". Functions that can be used without knowing how its implemented is function abstraction. Data abstraction is using data without knowing how the data is stored.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes. We know that class can have private, protected and public members. By default, all items defined in a class are private.

For example:

```
class add
{
public:
double getadd(void)
{
return no1 + no2;
}
private:
double no1; double no2;
};
```

The variables no1 & no2 are private. This means that they can be accessed only by other members of the add class, and not by any other part of your program. This is one way encapsulation is achieved.

To make parts of a class public (i.e., accessible to other parts of your program), you must declare them after the public keyword. All variables or functions defined after the public specifier are accessible by all other functions in your program.

Making one class a friend of another exposes the implementation details and reduces encapsulation. The ideal is to keep as many of the details of each class hidden from all other classes as possible.

File Handling

A file is collection of related records, a record is composed of several fields and field is a group of character.

This requires another standard C++ library called fstream which defines three new data types:

Ofstream

This data type represents the output file stream and is used to create files and to write information to files.

Ifstream

This data type represents the input file stream and is used to read information from files.

Fstream

This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

File Operations

1. Open an existing file
2. Read from file
3. Write to a file
4. Moving a specific location in a file(Seeking)
5. Closing File

Opening a File

A file must be opened before you can read from it or write to it. Either the ofstream or fstream object may be used to open a file for writing and ifstream object is used to open a file for reading purpose only. Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

```
void open(const char *filename, ios::openmode mode);
```

Here, the first argument specifies the name and location of the file to be opened and the second argument of the open() member function defines the mode in which the file should be opened.

file open modes

ios::app - Append mode. All output to that file to be appended to the end.
ios::ate - Open a file for output and move the read/write control to the end of the file.
ios::in - Open a file for reading.
ios::out - Open a file for writing.
ios::trunk - If the file already exists, its contents will be truncated before opening the file.

Closing a File

When a C++ program terminates, it automatically closes flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.
Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

```
void close();
```

Writing to a File

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an ofstream or fstream object instead of the cout object.

Reading from a File

You read information from a file into your program using the stream extraction operator (<<) just as you use that operator to input information from the keyboard. The only difference is that you use an ifstream or fstream object instead of the cin object./p>

File Position Pointers

Both istream and ostream provide member functions for repositioning the file-position pointer. These member functions are seekg ("seek get") for istream and seekp ("seek put") for ofstream. The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be ios::beg (the default) for positioning relative to the beginning of a stream, ios::cur for positioning relative to the current position in a stream or ios::end for positioning relative to the end of a stream

Exception Handling

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in our program by transferring control to special functions called handlers.
C++ exception handling is built upon three keywords: try, catch, and throw.

throw: A program throws an exception when a problem shows up. This is done using a throw keyword.
catch: A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
try: A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.
Assuming a block will raise an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
// protected code
}
catch( ExceptionName e1 )
{
// catch block
}
catch( ExceptionName e2 )
{
// catch block
}
catch( ExceptionName eN )
{
// catch block
}
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

Throwing Exceptions

Exceptions can be thrown anywhere within a code block using throw statements. The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs:

```
double division(int a, int b)
{
if( b == 0 )
{
throw "Division by zero condition!";
}
return (a/b);
}
```

Catching Exceptions

The catch block following the try block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try
{
// protected code
}catch( ExceptionName e )
{
```

```
// code to handle ExceptionName exception
}
```

C++ Standard Exceptions

std::exception -
An exception and parent class of all the standard C++ exceptions.
std::bad_alloc -
This can be thrown by new.
std::bad_cast -
This can be thrown by dynamic_cast.
std::bad_exception -
This is useful device to handle unexpected exceptions in a C++ program
std::bad_typeid -
This can be thrown by typeid.
std::logic_error -
An exception that theoretically can be detected by reading the code.
std::domain_error -
This is an exception thrown when a mathematically invalid domain is used
std::invalid_argument -
This is thrown due to invalid arguments.
std::length_error -
This is thrown when a too big std::string is created
std::out_of_range -
This can be thrown by the at method from for example a std::vector and std::bitset < >::operator[]().
std::runtime_error -
An exception that theoretically can not be detected by reading the code.
std::overflow_error -
This is thrown if a mathematical overflow occurs.
std::range_error -
This is occured when you try to store a value which is out of range.
std::underflow_error -
This is thrown if a mathematical underflow occurs.

Define your own exceptions

You can define your own exceptions by inheriting and overriding exception class functionality.

Example :

```
struct MyException : public exception
{
const char * display() const throw ()
{
return "This is Exception";
}
};
```

Here, what() is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception

Dynamic Memory

Allocating memory

There are two ways that memory gets allocated for data storage:

Compile Time (or static) Allocation

1. Memory for named variables is allocated by the compiler
2. Exact size and type of storage must be known at compile time
3. For standard array declarations, this is why the size has to be constant

Dynamic Memory Allocation

1. Memory allocated "on the fly" during run time
2. dynamically allocated space usually placed in a program segment known as the heap or the free store.
3. Exact amount of space or number of items does not have to be known by the compiler in advance.
4. For dynamic memory allocation, pointers are crucial.

Dynamic Memory Allocation

We can dynamically allocate storage space while the program is running, but we cannot create new variable names "on the fly"

For this reason, dynamic allocation requires two steps:

1. Creating the dynamic space.
2. Storing its address in a pointer (so that the space can be accesed)

To dynamically allocate memory in C++, we use the new operator.

De-allocation:

1. Deallocation is the "clean-up" of space being used for variables or other data storage
2. Compile time variables are automatically deallocated based on their known extent (this is the same as scope for "automatic" variables)
3. It is the programmer's job to deallocate dynamically created space
4. To de-allocate dynamic memory, we use the delete operator

Allocating space with new

To allocate space dynamically, use the unary operator new, followed by the type being allocated.

```
new int;
// dynamically allocates an int
new double;
// dynamically allocates a double
```

If creating an array dynamically, use the same form, but put brackets with a size after the type:

```
new int[40];
// dynamically allocates an array of 40 ints
new double[size];
// dynamically allocates an array of size doubles
```

These statements above are not very useful by themselves, because the allocated spaces have no names! BUT, the new operator returns the starting address of the allocated space, and this address can be stored in a pointer:

```
int * p;
// declare a pointer p
p = new int;
// dynamically allocate an int and load address into p
```

Deallocation of dynamic memory

To deallocate memory that was created with new, we use the unary operator delete. The one operand should be a pointer that stores the address of the space to be deallocated:

```
int * ptr = new int;
// dynamically created int
...
delete ptr;
// deletes the space that ptr points to
```

Note : The pointer ptr still exists in this example. That's a named variable subject to scope and extent determined at compile time. It can be reused:

```
ptr = new int[10];
// point p to a brand new array
```

To deallocate a dynamic array, use this form:

```
delete [] name_of_pointer
```

Templates

Templates in C++ programming allows function or class to work on more than one data type at once without writing different codes for different data types. Templates are often used in larger programs for the purpose of code reusability and flexibility of program. The concept of templetes can be used in two different ways:

1. Function Template
2. Class Template

Function Template

A function templates work in similar manner as function but with one key difference. A single function template can work on different types at once but, different functions are needed to perform identical task on different data types.The general form of a template function definition is shown here:

```
template ret-type func-name(parameter list)
{
// body of function
}
```

Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition.

Class Template

Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here:

```
template class class-name
{
.
}
```

Here, type is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a comma-separated list.