C,C++ Hand Book

Introduction To C

History

The C language was developed at AT&T Bell Labs in the early 1970s by Dennis Ritchie. It was based on an earlier Bell Labs language "B" which itself was based on the BCPL language (Basic Computer Programming Language). Since early on, C has been used with the Unix operating system, but it is not bound to any particular O/S or hardware.

C has gone through some revisions since its introduction. The American National Standards Institute(ANSI) developed the first standardized specification for the language in 1989, commonly referred to as C89. Before that, the only specification was an informal one from the book "The C Programming Language" by Brian Kernighan and Dennis Ritchie.

The next major revision was published in 1999. This revision introduced some new features, data types and some other changes. This is referred to as the C99 standard.

Objectives

C has been used successfully for every type of programming problem imaginable from operating systems to spreadsheets to expert systems - and efficient compilers are available for machines ranging in power from the Apple Macintosh to the Cray supercomputers. The largest measure of C's success seems to be based on purely practical considerations:

- the portability of the compiler;
- the standard library concept;
- a powerful and varied repertoire of operators;
- an elegant syntax;
- ready access to the hardware when needed;

The ease with which applications can be optimized by hand-coding isolated procedures C is often called a "High Level" programming language. This is not a reflection on its lack of programming power but more a reflection on its capability to access the system's low level functions.
Most high-level languages (e.g. Fortran) provides everything the programmer might want to do already built into the language. A low level language (e.g. assembly) provides nothing other than access to the machines basic instruction set. A middle level language, such as C, probably doesn't supply all the constructs found in high-languages - but it provides you with all the building blocks that you will need to produce the results you want!

Use of C

C was initially used for system development work, in particular the programs that make-up the operating system. Why use C? Mainly because it produces code that runs nearly as fast as code written in assembly language. Some examples of the use of C might be:

Operating Systems
Language Compilers
Assemblers
Text Editors
Print Spoolers
Network Drivers
Modern Programs
Data Bases
Language Interpreters
Utilities

In recent years C has been used as a general-purpose language because of its popularity with programmers. It is not the world's easiest language to learn and you will certainly benefit if you are not learning C as your first programming language! C is trendy (I nearly said sexy) - many well established programmers are switching to C for all sorts of reasons, but mainly because of the portability that writing standard C programs can offer.

C for Personal Computers

With regards to personal computers Microsoft C for IBM (or clones) PC's and Borland C are seen to be the two most commonly used systems. However, the latest version of Microsoft C is now considered to be the most powerful and efficient C compiler for personal computers. We hope we have now managed to convince you to continue with this online C course and hopefully in time become a confident C programmer. The Edit-Compile-Link-Execute Process Developing a program in a compiled language such as C requires at least four steps:

1. Editing (or writing) the program
2. Compiling it
3. Linking it
4. Executing it

Editing

You write a computer program with words and symbols that are understandable to human beings. This is the editing part of the development cycle. You type the program directly into a window on the screen and save the resulting text as a separate file. The custom is that the text of a C program is stored in a file with the extension .c for C programming language

Compiling

You cannot directly execute the source file. To run on any computer system, the source file must be translated into binary numbers understandable to the computer's Central Processing Unit (for example, the 80*87 microprocessor). This process produces an intermediate object file - with the extension .obj, the.obj stands for Object.

Linking

The first question that comes to most people's minds is why is linking necessary? The main reason is that many compiled languages come with library routines which can be added to your program. The subroutines are written by the manufacturer of the compiler to perform a variety of tasks, from input/output to complicated mathematical functions. In the case of C the standard input and output functions are contained in a library (stdio.h) so even the most basic program will require a library function. After linking the file extension is .exe which is executable files.

Executable files

Thus the text editor produces .c source files, which go to the compiler, which produces .obj object files, which go to the linker, which produces .exe executable file. You can then run .exe files as you can other applications.

Structure of C Programs

The form of a C Program All C programs will consist of at least one function, but it is usual (when your experience grows) to write a C program that comprises several functions. The only function that has to be present is the function called main. For more advanced programs the main function will act as a controlling function calling other functions in their turn to do the dirty work! The main function is the first function that is called when your program executes.C makes use of only 32 keywords which combine with the formal syntax to the form the C programming language. Note that all keywords are written in lower case - C, like UNIX.A keyword may not be used for any other purposes. For example, you cannot have a variable name called auto.

The layout of C Programs

A C program basically has the following form:

- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comments

The following program is written in the C programming language. Open a text file ***hello.c*** using editor and put the following lines inside that file.

```
#include <stdio.h>
int main()
{
  printf("Hello, World! \n");
  return 0;
}
```

Preprocessor Commands: This command tells the compiler to do preprocessing before doing actual compilation. Like *#include <stdio.h>* is a preprocessor command which tells a C compiler to include stdio.h file before going to actual compilation.

Functions: are main building blocks of any C Program. Every C Program will have one or more functions and there is one mandatory function which is called *main()* function. This function is prefixed with keyword *int* which means this function returns an integer value when it exits. This integer value is returned using *return* statement.

The C Programming language provides a set of built-in functions. In the above example *printf()* is a C built-in function which is used to print anything on the screen. Check Built-in function section for more detail.

You will learn how to write your own functions and use them in Using Function session.

Variables: are used to hold numbers, strings and complex data for manipulation. You will learn in detail about variables in C Variable Types.

Statements & Expressions: Expressions combine variables and constants to create new values. Statements are expressions, assignments, function calls, or control flow statements which make up C programs.

Comments: are used to give additional useful information inside a C Program. All the comments will be put inside /*...*/ as given in the example above. A comment can span through multiple lines.

Preprocessor Statements

C is a small language but provides the programmer with all the tools to be able to write powerful programs. C uses libraries of standard functions which are included when we build our programs. For the novice C programmer one of the many questions always asked is does a function already exist for what I want to do? Only experience will help here but we do include a function listing as part of this course. All pre-processor directives begin with a # and the must start in the first column. The commonest directive to all C programs is:

```
#include <stdio.h>
```

Global Declaration

The variables are declared before the main ( ) function as well as user defined functions is called global variables. These global variables can be accessed by all the user defined functions including main ( ) function.

Main () Function

Each and Every C program should contain only one main ( ). The C program execution starts with main ( ) function. No C program is executed without the main function. The main ( ) function should be written in small (lowercase) letters and it should not be terminated by semicolon. Main ( ) executes user defined program statements, library functions and user defined functions and all these statements should be enclosed within left and right braces.

() are used in conjunction with function names whereas

{} are used as to delimit the C statements that are associated with that function. Also note the semicolon - yes it is there, but you might have missed it! a semicolon (;) is used to terminate C statements. C is a free format language and long statements can be continued, without truncation, onto the next line. The semicolon informs the C compiler that the end of the statement has been reached. Free format also means that you can add as many spaces as you like to improve the look of your programs. A very common mistake made by everyone, who is new to the C programming language, is to miss off the semicolon. The C compiler will concatenate the various lines of the program together and then tries to understand them - which it will not be able to do. The error message produced by the compiler will relate to a line of you program which could be some distance from the initial mistake.

```
Int GlobalVariable =10;
Void main ()
{
}
```

Local Declarations

The variable declaration is a part of C program and all the variables are used in main ( ) function should be declared in the local declaration section is called local variables.

```
Void main ()
```

```
{
        Int LocalVariable =10;
}
```

Program Statements

These statements are building blocks of a program. They represent instructions to the computer to perform a specific task (operations). An instruction may contain an input-output statements, arithmetic statements, control statements, simple assignment statements and any other statements and it also includes comments that are enclosed within /* and */ . The comment statements are not compiled and executed and each executable statement should be terminated with semicolon.

Objectives

Having read this section you should have an understanding of:

- A pre-processor directive that must be present in all your C programs.
- A simple C function used to write information to your screen.
- How to add comments to your programs

Now that you've seen the compiler in action it's time for you to write your very own first C program. All your first program is going to do is print the message. "Hello World" on the screen.

```
#include <stdio.h>
void main()
{
printf("Hello World\n");
}
```

The first line is the standard start for all C programs - main (). After this comes the program's only instruction enclosed in curly brackets {}. The curly brackets mark the start and end of the list of Instructions that make up the program - in this case just one instruction. Notice the semicolon marking the end of the instruction. You might as well get into the habit of ending every C instruction with a semicolon - it will save you a lot of trouble! Also notice that the semicolon marks the end of an instruction - it isn't a separator as is the custom in other languages.
For example, you could enter the Hello World program as:

```
main()
{
        printf("Hello World\n");
}
```

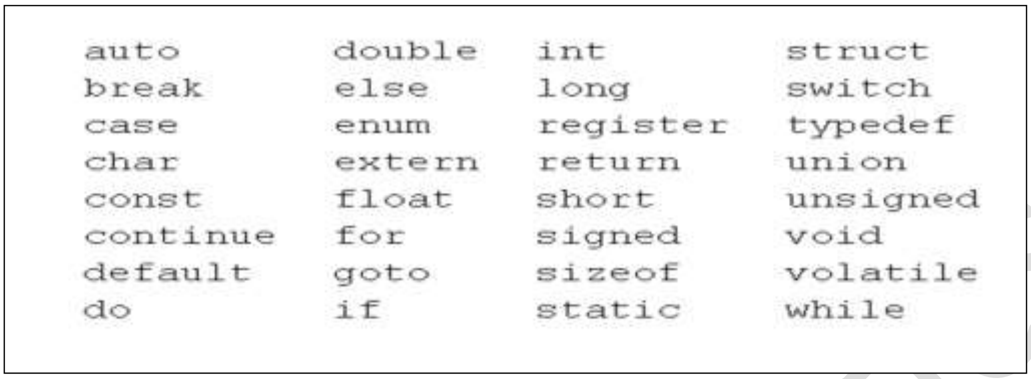
But this is unusual.
The printf function does what its name suggest it does: it prints, on the screen, whatever you tell it to. The "\n" is a special symbol that forces a new line on the screen. Then use the compiler to compile it, then the linker to link it and finally run it. The output is:

```
Hello World
```

Keyword and Identifier

Keywords

Keyword is the built-in words which are stored in library. We cannot use these word as a identifiers. Keywords like if, for, continue, break, etc....which have some meaning in library. Keywords are predefined reserved identifiers that have special meanings. They cannot be used as identifiers in your program.in c language 32 keywords.



auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Identifiers:

Identifiers are user define name or word like variable name, array name, function name and structure ,class name etc...

For example:

```
Int var1=90;
```

Here var1 is identifier because its variable name.

Rules for Identifiers

The identifiers must conform to the following rules.

1. First character must be an alphabet (or underscore)
2. Identifier names must consists of only letters, digits and underscore.
3. A identifier name should have less than 31 characters.
4. Any standard C language keyword cannot be used as a variable name.
5. A identifier should not contain a space.

Constants:

Constants refer to fixed values that may not be altered by the program. All the data types we have previously covered can be defined as constant data types if we so wish to do so. The constant data types must be defined before the main function.A variable is called constant when its value not changes during the program. Constants are 2 types:

1) const keyword
2) #define constant (symbolic constant)

1) Const keyword:

We can assign this keyword to any varible.and make it to const varible.For example:

```
Const int a=90;
```

Here, the value of a will be 90 to whole program,doesn't change its value.

2) #define constant (symbolic constant):

```
#define CONSTANTNAME value
```

For example:

```
#define SALESTAX 0.05
```

The constant name is normally written in capitals and does not have a semi-colon at the end. The use of constants is mainly for making your programs easier to be understood and modified by others and yourself in the future. An example program

```
#include <stdio.h>
#define SALESTAX 0.05
Main ()
{
        float amount, taxes, total;
        printf("Enter the amount purchased : ");
        scanf("%f",&amount);
        taxes = SALESTAX*amount;
        printf("The sales tax is £%4.2f",taxes);
        printf("\n The total bill is £%5.2f",total);
}
```

The float constant SALESTAX is defined with value 0.05. Three float variables are declared amount, taxes and total. Display message to the screen is achieved using printf and user input handled by scanf. Calculation is then performed and results sent to the screen. If the value of SALESTAX alters in the future it is very easy to change the value where it is defined rather than go through the whole program changing the individual values separately, which would be very time consuming in a large program with several references. The program is also improved when using constants rather than values as it improves the clarity.

Data Types

Now we have to start looking into the details of the C language. How easy you find the rest of this section will depend on whether you have ever programmed before - no matter what the language was. The first thing you need to know is that you can create variables to store values in. A variable is just a named area of storage that can hold a single value (numeric or character).To create variables and what you store in them. It demands that you declare the name of each variable that you are going to use and its type, or class, before you actually try to do anything with it. In this section we are only going to be discussing local variables. These are variables that are used within the current program unit (or function) in a later section we will looking at global variables - variables that are available to all the program's functions. There are five basic data types associated with variables:

Keyword	**Format Specifier**	**Size**	**Data Range**

char	%c	1 Byte	-128 to +127
unsigned char	<-- -- >	8 Bytes	0 to 255
int	%d	2 Bytes	-32768 to +32767
long int	%ld	4 Bytes	-2^{31} to $+2^{31}$
unsigned int	%u	2 Bytes	0 to 65535
float	%f	4 Bytes	$-3.4e^{38}$ to $+3.4e^{38}$
double	%lf	8 Bytes	$-1.7e^{308}$ to $+1.7e^{308}$
long double	%Lf	12-16 Bytes	Same as double

One of the confusing things about the C language is that the range of values and the amount of storage that each of these types takes is not defined. This is because in each case the 'natural' choice is made for each type of machine. You can call variables what you like, although it helps if you give them sensible names that give you a hint of what they're being used for - names like sum, total, average and so on.

Qualifier

When qualifier is applied to the data type then it changes its size. Size qualifiers : short, long

Sign qualifiers : signed, unsigned

Variable

A variable is a named data storage location in your computer's memory. By using a variable's name in your program, you are, in effect, referring to the data stored there.

Rules For Create Variable

- First character should be letter or alphabet.
- Keywords are not allowed to use as a variable name.
- White space is not allowed.
- C is case sensitive i.e. UPPER and lower case are significant.
- Only underscore, special symbol is allowed between two characters.
- The length of identifiers may be upto 31 characters but only only the first 8 characters are significant by compiler.

(Note: Some compilers allow variable names whose length may be upto 247 characters. But, it is recommended to use maximum 31 characters in variable name. Large variable name leads to occur errors.)

Integer Number Variables

The first type of variable we need to know about is of class type int - short for integer. An int variable can store a value in the range -32768 to +32767. You can think of it as a largish positive or negative whole number: no fractional part is allowed. To declare an int you use the instruction:

int variable_name;

For example:

```
int a;
```

declares that you want to create an int variable called a. To assign a value to our integer variable we would use the following C statement:

a=10;

The C programming language uses the "=" character for assignment. A statement of the form a=10; should be interpreted as take the numerical value 10 and store it in a memory location associated with the integer variable a. The "=" character should not be seen as an equality otherwise writing statements of the form:

a=a+10;

will get mathematicians blowing fuses! This statement should be interpreted as take the current value stored in a memory location associated with the integer variable a; add the numerical value 10 to it and then replace this value in the memory location associated with a.

Decimal Number Variables

As described above, an integer variable has no fractional part. Integer variables tend to be used for counting, whereas real numbers are used in arithmetic. C uses one of two keywords to declare a variable that is to be associated with a decimal number: float and double.

Float

A float, or floating point, number has about seven digits of precision and a range of about 1.E-36 to. A float takes four bytes to store.

Double

A double or double precision, number has about 13 digits of precision and a range of about 1.7E-308 to 1.7E+308. A double takes eight bytes to store.

For example:

```
float total;
double sum;
```

To assign a numerical value to our floating point and double precision variables we would use the following C statement:

```
total=0.0;
sum=12.50;
```

Character Variables

C only has a concept of numbers and characters. It very often comes as a surprise to some programmers who learnt a beginner's language such as BASIC that C has no understanding of strings but a string is only an array of characters.

For example:

char c;

To assign, or store, a character value in a char data type is easy - a character variable is just a symbol enclosed by single quotes. For example, if c is a char variable you can store the letter A in it using the following C statement:

```
c='A'
```

Notice that you can only store a single character in a char variable. Later we will be discussing using character strings, which has a very real potential for confusion because a string constant is written between double quotes. But for the moment remember that a char variable is 'A' and not "A".

Void Data type

The void type basically means "nothing". A void type cannot hold any values. You can also declare a function's return type as void to indicate that the function does not return any value.

Enum Data Type

This is an user defined data type having finite set of enumeration constants. The keyword 'enum' is used to create enumerated data type.

Typedef

It is used to create new data type. But it is commonly used to change existing data type with another name.

Input and Output Functions

```
a=100;
```

Stores 100 in the variable a each time you run the program, no matter what you do. Without some sort of input command every program would produce exactly the same result every time it was run. There are a number of different C input commands, the most useful of which is the scanf command. To read a single integer value into the variable called a you would use:

```
scanf("%d",&a);
```

For the moment doesn’t worry about what the %d or the &a means - concentrate on the difference between this and:

```
a=100;
```

When the program reaches the scanf statement it pauses to give the user time to type something on the keyboard and continues only when users press <Enter>, or <Return>, to signal that he, or she, has finished entering the value. Then the program continues with the new value stored in a.
To display the value stored in the variable a you would use:

```
printf("The value stored in a is %d",a);
```

The %d, both in the case of scanf and printf, simply lets the compiler know that the value being read in, or printed out, is a decimal integer - that is, a few digits but no decimal point.Note: the scanf function does not prompt for an input. You should get in the habit of always using a printf function, informing the user of the program what they should type, before a scanf function.

The % Format Specifiers

The % specifiers that you can use in ANSI C are: Usual variable type Display
%c char single character
%d (%i) int signed integer
%e (%E) float or double exponential format
%f float or double signed decimal

Formatting Your Output

flag width.precision number.
\b backspace
\f formfeed
\n new line
\0 null
Example:

```
#include <stdio.h>
main ()
{
        int a,b,c;
        printf("\nThe first number is ");
        scanf("%d",&a);
        printf("The second number is ");
        scanf("%d",&b);
        c=a+b;
        printf("The answer is %d \n",c);
}
```

Single character input output:

The getchar function can be used to read a character from the standard input device. The scanf can also be used to achieve the function. The getchar has the following form.
Character _variable = getchar();
Variable name is a valid 'C' variable, that has been declared already and that possess the type char.
For example:

```
char ch;

ch = getchar();
```

The putchar function which in analogus to getchar function can be used for writing characters one at a time to the output terminal. The general form is

putchar (variable name);

```
for example:

putchar (ch);
```

Displays the value stored in variable ch to the standard screen.

Storage Classes

A storage class defines scope(visibility) and life time of varibles and function. Types of storage classes:

1) auto
2) register
3) static
4) extern

1) Auto:

Auto is default storage class for all local variables.
Keyword: auto
Storage location: main memory
Scope: local to that block when it declare.
For example:

```
#include <stdio.h>
#include <conio.h>
void main()
{
        auto inti=10;
        clrscr();
        {
                auto inti=20;
                printf("\n\t %d",i);
        }
        printf("\n\n\t %d",i);
        getch();
}
```

Output:

```
20

10
```

Here both are same.

2) Register:

Register is fast access variable because its store in CPU register.
Keyword: register
Storage location: CPU registers
Scope: local to that block when it declares.
For example:

```
#include <stdio.h>
#include <conio.h>
void main()
{
```

Output:

```
20

10
```

```
          register inti=10;
          clrscr();
          {
                    register inti=20;
                    printf("\n\t %d",i);
          }
          printf("\n\n\t %d",i);
getch();
}
```

3) Static:

Keyword: static
Storage Location: Main memory
Initial Value: Zero and can be initialize once only.
Life: depends on function calls and the whole application or program.
Scope: Local to the block.
Syntax:

```
static [data_type] [variable_name];
```

Example:

```
Static int a;
```

There are two types of static variables as :
a) Local Static Variable
b) Global Static Variable

Static storage class can be used only if we want the value of a variable to persist between different function calls.

Example

```
#include <stdio.h>
#include <conio.h>
void main()
{
inti;
void incre(void);
clrscr();
for (i=0; i<3; i++)
{
          incre();  getch();
}}
```

```
void incre(void)
{
          intavar=1;
          static intsvar=1;
          avar++;
          svar++;
          printf("\n\n Automatic variable
          value : %d",avar);
          printf("\t Static variable value :
          %d",svar);
}
```

4) extern:

Variables can be declared as either local variables which can be used inside the function it has been declared in (more on this in further sections) or global variables which are known throughout the entire program. Global variables are created by declaring them outside any function. For example:

Example:

```
#include <stdio.h>
#include <conio.h>
extern inti=10;
void show(void)
{
            printf("\n\n\t %d",i);
}
void main()
{
            int i=20;
             void show(void);
            clrscr();
            printf("\n\t %d",i);
            show();
            getch();
}
```

The int max can be used in both main and function f1 and any changes made to it will remain consistent for both functions. The understanding of this will become clearer when you have studied the section on functions but I felt I couldn't complete a section on data types without mentioning global and local variables.

Operators

Assignment Operator:

Once you've declared a variable you can use it, but not until it has been declared - attempts to use a variable that has not been defined will cause a compiler error. Using a variable means storing something in it. You can store a value in a variable using:

```
            name = value;
```

For example:

```
            int a=10;
            float b=90.66;
            char ch='p';
```

Stores the value 10 in the int variable a.

Arithmetic Operator:

Operator	Example
+	C=a+b
-	C=a-b
*	C=a*b
/	C=a/b
%	C=a%b

For example:

```
#include <conio.h>
void main()
{
        int a,b,c,d,e,f,g;
        printf("\n\t Enter First Number :");
        scanf("%d",&a);
        printf("\n\t Enter Second Number :");
        scanf("%d",&b);
        c = a + b;
        d = a - b;
        e = a * b;
        f = a / b;
        g = a % b;
        printf("\n\n\t Addition is : %d",c);
        printf("\n\n\t Subtraction is : %d",d);
        printf("\n\n\t Multiplication is : %d",e);
        printf("\n\n\t Division is : %d",f);
        printf("\n\n\t Modulus is : %d",g);
}
```

Logical Operator:

C has the following logical operators; they compare or evaluate logical and relational expressions.

Operator	Meaning
&&	Logical AND
!!	_ Logical OR
!	Logical NOT

Logical AND (&&):

condition1	**Condition2**	**Conditon1 && condition2**
True	true	True
True	False	False
False	True	False
False	False	False

Logical OR (||):

condition1	**Condition2**	**Conditon1 || condition2**
True	true	True
True	False	True
False	True	True
False	False	False

Logical NOT(!):

Operand	**Result**
0	1
1	0

The logical operator example, we discuss later.

Shorthand Operator:

It is used to perform mathematical operations at which the result or output can affect on operands. we use shorthand operator, when there is same variable around = operator.

Operator	**Example**	**Shorthand operator**
+=	C=c+b	C+=b
-=	C=c-b	C-=b
*=	C=c*b	C*=b
/=	C=c/b	C/=b
%=	C=c%b	C%=b

Program:

```
#include <stdio.h>
void main()
{
         int a,b;
        a = 18;
        b = 4;
        printf("\n\t Value of A : %d",a);
        printf("\n\t Using of B : %d",b);
        b += a ;                          // b = b + a
        printf("\n\n\t answer= %d",b);
}
```

Unary Operator:

1) Increment (++)

Increment operator means add 1 in the current value. There is 2 way to add 1 in current valu.like prefix and post fix. Prefix is the operator first then operand. Postfix means first take operand and then operator. Prefix example:

```
int b,a=20;

b=++a;
```

postfix example :

```
int b,a=20;

b=a++;
```

2) Decrement (--)

Decrement operator means minus or decrease 1 in the current value. There is 2 way to add 1 in current value .like prefix and post fix. Prefix & Postfix is same as above increment (++) Operator.

Prefix example:

```
int b,a=20;

b=--a;
```

Postfix example :

```
int b,a=20;

b=a--;
```

Conditional Operator:

Conditional operator is also called as 'ternary operator.' It is widely used to execute condition in true part or in false part. It operates on three operands. The logical or relational operator can be used to check conditions. Syntax:

```
(Condition)?  True statements:  false statements   ;
```

Example:

```
int a=9,b=10
 (a>b)?printf("\n a is greater"): printf("\n a is greater") ;
```

Relational Operator:

Relational operator is used when there is relation between 2 or more variables.

Operator	**Meaning**
<	Less than
<=	Less than or equal to
>	greater than
>=	greater than or equal to
!=	Not equal to
==	Is equal to

The logical operator example, we discuss later in control statement.

Special Operator:

The Comma Operator:

The comma operator can be used to link related expressions together. A comma-linked list of expressions is evaluated left to right and value of right most expression is the value of the combined expression. For example the statement

```
value = (x = 10, y = 5, x + y);
```

First assigns 10 to x and 5 to y and finally assigns 15 to value. Since comma has the lowest precedence in operators the parenthesis is necessary.

The sizeof Operator :

The operator size of gives the size of the data type or variable in terms of bytes occupied in the memory. The operand may be a variable, a constant or a data type qualifier.

Example

m = sizeof (sum);   //ans is 2 if sum is integer.
n = sizeof (long int);   //ans is 4

The size of operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer.

Expression Evolution

Arithmetic Expressions

An expression is a combination of variables constants and operators written according to the syntax of C language. In C every expression evaluates to a value i.e., every expression results in some value of a certain type that can be assigned to a variable. Some examples of C expressions are shown in the table given below.

Algebraic Expression - C Expression

a x b – c -> a * b – c
(m + n) (x + y) -> (m + n) * (x + y)
(ab / c) -> a * b / c
3x2 +2x + 1 -> 3*x*x+2*x+1
(x / y) + c -> x / y + c

Evaluation of Expressions

Expressions are evaluated using an assignment statement of the form

Variable = expression;

Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and then replaces the previous value of the variable on the left hand side. All variables used in the expression must be assigned values before evaluation is attempted. Example of evaluation statements are

x = a * b – c
y = b / c * a
z = a – b / c + d;
The following program illustrates the effect of presence of parenthesis in expressions.

```
main ()
{
        float a, b, c x, y, z;
        a = 9;
        b = 12;
        c = 3;
        x = a – b / 3 + c * 2 – 1;
        y = a – b / (3 + c) * (2 – 1);
        z = a – ( b / (3 + c) * 2) – 1;
        printf (“x = %fn”,x);
        printf (“y = %fn”,y);
        printf (“z = %fn”,z);
}
```

```
Output
x = 10.00
y = 7.00
z = 4.00
```

Precedence in Arithmetic Operators

An arithmetic expression without parenthesis will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C. High priority * / % ; Low priority + -

Rules for evaluation of expression

1. First parenthesized sub expression left to right are evaluated.
2. If parentheses are nested, the evaluation begins with the innermost sub expression.
3. The precedence rule is applied in determining the order of application of operators in evaluating sub expressions.
4. The associability rule is applied when two or more operators of the same precedence level appear in the sub expression.
5. Arithmetic expressions are evaluated from left to right using the rules of precedence.
6. When Parenthesis is used, the expressions within parenthesis assume highest priority.

Type conversion:

When variables and constants of different types are combined in an expression then they are converted to same data type.

The process of converting one predefined type into another is called type conversion.
Type conversion in c can be classified into the following two types:

Implicit Conversion:

When the type conversion is performed automatically by the compiler without programmers intervention, such type of conversion is known as implicit type conversion or type promotion.

The compiler converts all operands into the data type of the largest operand.
The sequence of rules that are applied while evaluating expressions are given below:

All short and char are automatically converted to int, then,

1. If either of the operand is of type long double, then others will be converted to long double and result will be long double.
2. Else, if either of the operand is double, then others are converted to double.
3. Else, if either of the operand is float, then others are converted to float.
4. Else, if either of the operand is unsigned long int, then others will be converted to unsigned long int.
5. Else, if one of the operand is long int, and the other is unsigned int, then
   1. if a long int can represent all values of an unsigned int, the unsigned int is converted to long int.
   2. otherwise, both operands are converted to unsigned long int.
6. Else, if either operand is long int then other will be converted to long int.
7. Else, if either operand is unsigned int then others will be converted to unsigned int.

It should be noted that the final result of expression is converted to type of variable on left side of assignment operator before assigning value to it.

Also, conversion of float to int causes truncation of fractional part, conversion of double to float causes rounding of digits and the conversion of long int to int causes dropping of excess higher order bits.

Explicit Conversion

The type conversion performed by the programmer by posing the data type of the expression of specific type is known as explicit type conversion.

The explicit type conversion is also known as type casting.
Type casting in c is done in the following form:
(data_type) expression;
Where, data_type is any valid c data type, and expression may be constant, variable or expression.
For example,
1
x= (int)a+b*d;

The following rules have to be followed while converting the expression from one type to another to avoid the loss of information:

1. All integer types to be converted to float.
2. All float types to be converted to double.
3. All character types to be converted to integer.

Decision making and branching statements

C language provides statements that can alter the flow of a sequence of instructions. These statements are called control statements. These statements help to jump from one part of the program to another.
1) If statement.
2) If else statement.
3) Nested if statement.

4) Else if ladder statement.
5) Switch statement.

1) If statement.

It is very frequently used in decision making and allowing the flow of program execution. The If structure has the following syntax

```
If (condition)
        Statement;
```

The condition is logical operator which are used in the condition statement. The condition part should not end with a semicolon, since the condition and statement should be put together as a single statement. If condition is true the block will be execute.

For example:

```
Void main()
{
        int age=90;
        if(age>18)
        {
                Printf("\n u are eligible for watting..");
        }
}
```

2) If...Else statement:

If the result of the condition is true, then program statement 1 is executed, otherwise program statement 2 will be executed. If any case either program statement 1 is executed or program statement 2 is executed. Both statements will not execute at a time.

For example:

```
Void main()
{
        int age=15;
        If (age>18) {
                Printf("\n u are eligible for watting..");}
        Else{
                Printf("\n u are not eligible for watting.."); }
}
```

In above example if the age is greater tha 18 then student is eligible.here age is 15 so else part will be executed. An example program showing the relational operator.

```
#include <stdio.h>
```

```
Main ()
{
        int num1, num2;
        printf("\nEnter first number ");
        scanf("%d",&num1);
        printf("\nEnter second number ");
        scanf("%d",&num2);
        if (num2 ==0)
                printf("\n\nCannot devide by zero\n\n");
        else
                printf("\n\nAnswer is %d\n\n",num1/num2);
}
```

3) Nested if statement:

The if statement may itself contain another if statement is known as nested if statement.

Syntax:

```
If (condition1)
  If (condition2)
    Statement-1;
  Else
    Statement-2;
Else
   Statement-3;
```

Here if one condition is true then inner condition will be executed. So one block of code will only be executed if two conditions are true. Condition 1 is tested first and then condition 2 is tested. The second if condition is nested in the first. The second if condition is tested only when the first condition is true else the program flow will skip to the corresponding else statement.

For example:

```
main()
{
   int a,b,c;
   printf ("Enter three numbers")
   scanf ("%d %d %d", &a, &b, &c)
   if (a > b)
   {      if (a > c)  printf("\n a is greater.");
           Else     printf("\n c is greater.");
   }
   else
       printf("\n b is greater.");
}
```

In above if a is greater b then it check to c.otherwise b is greater message will be display.

4) Else if ladder:

When many conditions have to be checked we may use the ladder else if statement which takes the following general form.

```
If (condition1)
  Statement – 1;
else if (condition2)
  statement2;
else if (condition3)
  statement3;
else if (condition)
  Statement n;
else
  Default statement;
Statement-x;
```

The conditions are check from the top to down. As soon on the true condition is found, the statement associated with it is executed and the control is transferred to the statement – x (skipping the rest of the ladder. When all the condition becomes false, the final else containing the default statement will be executed.)

For example:

```
#include<stdio.h>
#include<conio.h>
void main ()
{
  float per ;
  printf ("Enter marks\n") ;
  scanf ("%f", &per)  ;
 if (per>100)
       printf("Value Entered Greater than 100%. Please Enter Valid Value.");
 else if (per <= 100 && per >= 70)
       printf ("\n Distinction");
 else if (per >= 60)
       printf("\n First class") ;
 else if (per >= 50)
       printf ("\n second class");
 else if (per >= 35)
       printf ("\n pass class");
 else
       printf ("Fail");
getch();
}
```

In the first If condition statement it checks whether the input value is lesser than 100 and greater than 70. If both conditions are true it prints distinction .same as it is next if condition.

5) Switch statement.

The switch statement allows a program to select one statement for execution out of a set of alternatives. During the execution of the switch statement only one of the possible statements will be executed the remaining statements will be skipped. The usage of multiple If else statement increases the complexity of the program since when the number of If else statements increase it affects the readability of the program and makes it difficult to follow the program. The switch statement removes these disadvantages by using a simple and straight forward approach.we can't use float datatype in switch and also not use relational and logical operator.

The general format:

```
Switch (expression)
{
case label-1:
                statements;
                break;
case label-2:
                statements;
                break;
case label-n:
                statements;
                break;
………………
case default:
                break;
}
```

The switch statement is check the control expression is evaluated first and the value is compared with the case label values in the given order. If the label matches with the value of the expression then the control is transferred directly to the group of statements which follow the label. If none of the statements matches then the statement against the default is executed. The default statement is optional in switch statement.

For example:

```
void main ()
{
  char ch='i' ;
  switch (ch)
  {
            case 'a':
              printf("\nu press a");
              break;
            case 'i':
              printf("\nu press i");
              break;
            case 'o':
              printf("\nu press o ");
              break;
            case 'u':
              printf("\nu press u ");
```

```
                    break;
                  case 'e':
                    printf("\nu press e ");
                    break;
                  default:
                    printf ("\n unknown character.") ;
                    break;
     }
   }
```

In above example ch variable first check the case 'a' if case match execute that statement block and break means next statements are not execute. break is used when jump the statement or break the switch statement. here case i block will be execute.

Go to statement:

This statement is simple statement used to transfer the program control unconditionally from one statement to another statement. Although it might not be essential to use the goto statement in a highly structured language like C, there may be occasions when the use of goto is desirable.

The GOTO statement:

The goto requires a label in order to identify the place where the branch is to be made. A label is a valid variable name followed by a colon. The label is placed immediately before the statement where the control is to be transformed.There are main two types of goto statement.forward goto and backword goto. In forword goto the part of statement or program will be skip. Syntax of forword goto:

```
 goto label;

…………
Statement;
…………
Label:
Statement;
```

For example:

```
goto  stop;
printf("\n 1 statement..");
printf("\n 2 statement..");
stop:
printf("\n last statement..");
```

In above example 1st and 2nd statement will be skip.
Now in backword goto statement use, we can repeat the statement.
Syntax of backword goto:

```
Label:
…………
Statement;
…………
```

```
goto label;
```

For example:

```
n=2;
start:
if(n<5)
{
        Printf("\n value of n=%d",n);
        n++;
        goto start;
}
```

In above example the statement will execute 3 times.backword goto statement work like loop sometime, but we not use.

Control Loops

Decision making –Loop:

A loop means any statement repeat multiple times.A loop consists of two segments one known as body of the loop and other is the control statement. The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop. In looping process in general would include the following three steps

1) initialize counter
2) test or check that counter
3) update that counter

Types of loops:

1) While loop
2) For loop
3) Do while loop

1)The While loop:

The while loop is entry control loop means in this loop we first check the condition, if condition is true then execute statements. The general format of the while statement is:

```
Initialization;

While (test condition)
{
        //body of the loop
        Updation counter variable;
}
```

Here after the execution of the body, the test condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test condition finally becomes false and the control is transferred out of the loop.

For example sum of 1 to 10 number:

```
int i=1,n=10,sum=0;
```

```
            while (i<=n)
            {
                        Sum+=i;                 //sum=sum+i;
                        i++;                //increment of counter
            }
```

In above example the i is counter variable and test condition upto 10 times and execute the sum 10 times. At last when i=11 at that time the condition will be false so the loop will terminate.

2) For loop:

The for loop is also entry control loop means in this loop we first check the condition,if condition is true then execute statements.The general format of the for statement is:

```
for (Initialization;test condition;updation counter variable)
{
            //body of the loop
}
```

Here after the execution of the body, the test condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test condition finally becomes false and the control is transferred out of the loop.
For example sum of 1 to 10 number:

```
            int n=5, i;
            for (i =0;i<= n;i++)
            {
                        printf("\t%d",i);
            }
```

Here n and I are declared as integer variables and I is initialized to value zero. At last when i=6 at that time the condition will be false so the loop will terminate.

3)The do while loop:

The do while loop tests at the bottom of the loop after executing the body of the loop os its exit control loop. Since the body of the loop is executed first and then the loop condition is checked we can be assured that the body of the loop is executed at least once. The syntax of the do while loop is:

```
do
{
            statement;
            updation;
}
 while(condition);
```

Here the statement is executed, then expression is evaluated. When the expression becomes false. When the expression becomes false the loop terminates.
For example:

```
            int  i=10;
```

```
do
{
            printf(“\n value=%d”,i);
            i++;
} while (i<9);
```

In above example the i value is 10 but the statement will be execute at least once. then check condition.

Using break and continue within Loops

The break statement allows you to exit a loop from any point within its body, bypassing its normal termination expression. When the break statement is encountered inside a loop, the loop is immediately terminated, and program control resumes at the next statement following the loop. The break statement can be used with all three of C's loops. The continue statement is somewhat the opposite of the break statement. It forces the next iteration of the loop to take place, skipping any code in between itself and the test condition of the loop. In while and do-while loops, a continue statement will cause control to go directly to the test condition and then continue the looping process. In the case of the for loop, the increment part of the loop continues.For example:

```
#include <stdio.h>
main()
{
            int x ;
            For (x=0; x<=100; x++)
            {
                  if (x%2==0)
                        continue;
                   printf("%d\n" , x);
             }
}
```

Here we have used C's modulus operator: %. An expression:
a % b
Produces the remainder when a is divided by b; and zero when there is no remainder. Here's an example of a use for the break statement:

```
#include <stdio.h>
main()
{           int i=1;
            while (i<=6)
            {           if(i==4) break;
                        printf(“\n%d”,i); }}
```

In above example only 1 to 3 display.because when i=4 the if condition execute and stop execution.

Array:

There are times when we need to store a complete list of numbers or other data items. You could do this by creating as many individual variables as would be needed for the job, but this is a hard and tedious process.In C programming Array is solution of this. An array have a common datatype, common name with different values with continues memory allocation.So we can easily use each value with its name.
Array has 3 types:

1) One dimension array.
2) Two dimension array.
3) Multi dimension array.

1) One dimension array.

One dimension array is a continues memory allocation which have stored in memory like a row or like a column. Declaration of 1D array:

```
Datatype  array_name[array_size];
```

Here the type specifies the type of the elements that will be contained in the array, such as int ,float or char and the size indicates the maximum number of elements that can be stored. For example:

```
int a[5];
```

in above example the first element is a[0] and the last a[4]. C programmer's always start counting at zero. The first array element is array[0] and the last is array[size-1].

Initialize array at compile time:

We can initialize array element compile time and run time also. Here the example of compile time:

```
int a[5]={1,2,3,4,5};
```

In above example the compiler knows the array element so its called compile time array. The memory allocate like:

```
a[0] = 1;
a[1] = 2;
a[2] = 3;
a[3] = 4;
a[4] = 5;
```

Initialize array at run time

We can initialize array element at run time. Means when program run the array value will initialize.user can enter value at run time. Here the example of compile time:

```
main()
{
```

```
            int a[5];
            int i;
            for(i =0;i < 5;i++)
                        scanf("%d",&a[i]);
            for(i =4;i> =0;i--)
                      printf("%d",a[i]);
}
```

The for loop and the array data type were more or less made for each other. The for loop can be used to generate a sequence of values to pick out and process each element in an array in turn. An array of character variables is in no way different from an array of numeric variables, but programmers often like to think about them in a different way. For example, if you want to read in and reverse five characters you could use:

```
Main()
{
            char a[5];
            int i;
            for(i=0; i<5; ++i) scanf("%c",&a[i]);
            for(i=4;i>=0;--i) printf("%c",a[i]);
}
```

Notice that the only difference, is the declared type of the array and the %c used to specify that the data is to be interpreted as a character in scanf and printf. The trouble with character arrays is that to use them as if they were text strings you have to remember how many characters they hold. In other words, if you declare a character array 40 elements long and store H E L L O in it you need to remember that after element 4 the array is empty. This is such a nuisance that C uses the simple convention that the end of a string of characters is marked by a null character. A null character is, as you might expect, the character with ASCII code 0.

String input output using character array:

Gets and puts function:

The gets function can be used to read a character string from the standard input device. The scanf can also be used to achieve the function. The gets has the following form.

Character _arr = gets();
Array name is a valid aaray name, that has been declared already and that possess the type char.
For example:

```
char  arr[10];
arr= gets();
```

The puts function which in analogus to gets function can be used for writing character straing at a time to the output terminal. The general form is

puts(aray_name);
for example:
puts (arr);

Displays the value stored in array arr to the standard screen.

Two dimension array.

Two dimension array is a continues memory allocation which have stored in memory like tabular format.means multiple row and multiple column formate.

Declaration of 2D array:

```
Datatype  array_name[row_size][column_size];
```

Here the type specifies the type of the elements that will be contained in the array, such as int ,float or char and the size indicates the maximum number of elements that can be stored.first subscribe is used for row size and second subscribe is used for column size.

For example:

```
int b[2][3];
```

In above example array name is b and the total row size is 2 and column size is 3. the first element is b[0][0] and the last b[1][2].

Initialize array at compile time:

We can initialize array element compile time and run time also. here the example of compile time:

int b[2][3]={1,2,3,4,5,6};

in above example the compiler knows the array element so its called compile time array.the memory allocate like:

```
b[0][0] = 1;
b[0][1] = 2;
b[0][3] = 3;
b[1][0] = 4;
b[1][1] = 5;
```

Initialize array at run time:

We can initialize array element at run time.means when program run the arrray value will initialize.user can enter value at run time. Here the example of compile time:

```
main()
{
        int a[2][2],i,j;
        for(i =0;i < 2; ++i)
          for(j =0;j < 2; ++i)
                scanf("%d",&a[i][j]);
        for(i =0;i < 2; ++i)
          for(j =0;j < 2; ++i)
               printf("%d",a[i][j]);
}
```

In above example 2 number of rows and 2 number of column. At run time user can input the data and also display that data. Here 2 by 2 matrix create.2D array also called matrix.

Multidimensional arrays:

C allows arrays of three or more dimensions. The compiler determines the maximum number of dimension. The general form of a multidimensional array declaration is:

date_type array_name[s1][s2][s3].....[sn];

Some examples are:
int survey[3][5][12];
float table[5][4][5][3];
Survey is a 3 dimensional array declared to contain 180 integer elements. Similarly table is a four dimensional array containing 300 elements of floating point type.

Handling of Character String

In this tutorial you will learn about Initializing Strings, Reading Strings from the terminal, Writing strings to screen, Arithmetic operations on characters, String operations (string.h), Strlen() function, strcat() function, strcmp function, strcmpi() function, strcpy() function, strlwr () function, strrev() function and strupr() function.

In C language, strings are stored in an array of char type along with the null terminating character "\0" at the end. In other words to create a string in C you create an array of chars and set each element in the array to a char value that makes up the string. When sizing the string array you need to add plus one to the actual size of the string to make space for the null terminating character, "\0"
Syntax to declare a string in C:

char fname[4];

The above statement declares a string called fname that can take up to 3 characters. It can be I indexed just as a regular array as well.

fname[] = {'t','w','o'};

Character	t	W	O	\0
ASCII Code	116	119	41	0

The last character is the null character having ASCII value zero.

Initializing Strings

To initialize our fname string from above to store the name Brian,

char fname[31] = {"Purvi"};

You can observe from the above statement that initializing a string is same as with any array. However we also need to surround the string with quotes.

Writing Strings to the Screen

To write strings to the terminal, we use a file stream known as stdout. The most common function to use for writing to stdout in C is the printf function, defined as follows:

int printf(const char *format, ...);
To print out a prompt for the user you can:
printf("Please type a name: **\n**");

The above statement prints the prompt in the quotes and moves the cursor to the next line.If you wanted to print a string from a variable, such as our fname string above you can do this:
printf("First Name: %s", fname);
You can insert more than one variable; hence the "..." in the prototype for printf but this is sufficient. Use %s to insert a string and then list the variables that go to each %s in your string you are printing. It goes in order of first to last. Let's use a first and last name printing example to show this:

printf("Full Name: %s %s", fname, lname);

The first name would be displayed first and the last name would be after the space between the %s's.

Reading Strings from the Terminal

When we read a string from the terminal we read from a file stream known as stdin. *nix users are probably familiar with this, it's how you can type a program name into the terminal and pass it arguments also. Say we want to allow the user to enter a name from stdin. Aside from taking the name as a command line argument, we can use the scanf function which has the following prototype:

int scanf(const char *format, ...);

Note the similarity to printf.

Quick example to tie the last few sections together.

```
#include <stdio.h>
void main() {
char fname[30];
char lname[30];
printf("Type first name:\n");
scanf("%s", fname);
  printf("Type last name:\n");
scanf("%s", lname);
    printf("Your name is: %s %s\n", fname, lname);
}
```

OUTPUT:

```
Type First Name:
tops
Type Last Name:
ahmedabad
Your Name is: tops ahmedabad
```

We declare two strings fname and lname. Then we use the printf function to prompt the user for a first name. The scanf function takes the input from stdio and automatically exits once the user presses enter. Then we repeat the above sequence except using the last name this time. Finally we print the full name that was typed back to stdout. Should look something like this:

Arithmetic Operations on Strings

Characters in C can be used just like integers when used with arithmetic operators. This is nice, for example, in low memory applications because unsigned chars take up less memory than do regular integers as long as your value does not exceed the rather limited range of an unsigned char. Let us cut to our example,

```
#include <stdio.h>
void main() {
        unsigned char val1 = 20;
        unsigned char val2 = 30;
        int answer;
        printf("%d\n", val1);
        printf("%d\n", val2);
        answer = val1 + val2;
        printf("%d + %d = %d\n", val1, val2, answer);
        val1 = 'a';
        answer = val1 + val2;
        printf("%d + %d = %d\n", val1, val2, answer);
}
```

First we make two unsigned character variables and give them (rather low) number values. We then add them together and put the answer into an integer variable. We can do this without a cast because characters are an alphanumeric data type. Next we set var1 to an expected character value, the letter lowercase a. Now this next addition adds 97 to 30, why? Because the ASCII value of lowercase a is 97. So it adds 97 to 30, the current value in var2. Notice it did not require casting the characters to integers or having the compiler complain. This is because the compiler knows when to automatically change between characters and integers or other numeric types.

String Operations

Character arrays are a special type of array that uses a "\0" character at the end. As such it has it is own header library called string.h that contains built-in functions for performing operations on these specific array types. You must include the string header file in your programs to utilize this functionality.

#include <string.h>

We will cover the essential functions from this library over the next few sections.

Length of a String

Use the strlen function to get the length of a string minus the null terminating character. int strlen(string);
If we had a string, and called the strlen function on it we could get its length.

```
 char fname[30] = {"Bob"};
 int length = strlen(fname);
```

This would set length to 3.

Concatenation of Strings

The strcat function appends one string to another.

```
char  strcat(string1, string2);
```

The first string gets the second string appended to it. So for example to print a full name from a first and last name string we could do the following:

```
 char fname[30] = {"Bob"};
 char lname[30] = {"by"};
 printf("%s", strcat(fname, lname));
```

The output of this snippet would be "Bobby."

Compare Two Strings

Sometimes you want to determine if two strings are the same. For this we have the strcmp function.

```
int strcmp(string1, string2);
```

The return value indicates how the 2 strings relate to each other. if they are equal strcmp returns 0. The value will be negative if string1 is less than string2, or positive in the opposite case. For example if we add the following line to the end of our getname.c program:

```
printf("%d", strcmp(fname, lname));
```

When run on a Linux computer with the following first and last name combinations, the program will yield the following output.

First name: Bob, last name: bob, output: -1.
First name: bob, last name: Bob, output 1.
First name: Bob, last name: Bob, output 0.

Compare Two Strings (Not Case Sensitive)

If you do not care whether your strings are upper case or lower case then use this function instead of the strcmp function. Other than that, it's exactly the same.

```
int strcmpi(string1, string2);
```

Imagine using this function in place of strcmp in the above example, all of the first and last combinations would output 0.

Copy Strings

To copy one string to another string variable, you use the strcpy function. This makes up for not being able to use the "=" operator to set the value of a string variable.

```
1.  strcpy(string1, string2);
```

To set the first name of our running example in code rather than terminal input we would use the following:

strcpy(fname, "purvi");

Converting Uppercase Strings to Lowercase Strings

This function is not part of the ANSI standard and therefore strongly recommended against if you want your code to be portable to platforms other than Windows.

strlwr(string);

This will convert uppercase characters in string to lowercase. So "TOPS" would become "tops".

Reversing the Order of a String

This function is not part of the ANSI standard and therefore strongly recommended against if you want your code to be portable to platforms other than Windows.
strrev(string);
Will reverse the order of string. So if string was "bobby", it would become "ybbob".

Converting Lowercase Strings to Uppercase Strings

This function is not part of the ANSI standard and therefore strongly recommended against if you want your code to be portable to platforms other than Windows.
strupr(string);
This will convert lowercase characters in string to uppercase. So "bobby" would become "BOBBY".

Function:

Functions - C's Building Blocks or we can say subroutines or procedures or method. C program does not execute the functions directly. It is required to invoke or call that functions. When a function is called in a program then program control goes to the function body. Then, it executes the statements which are involved in a function body. Therefore, it is possible to call function whenever we want to process that functions statements.

Types of Function

There are two types of function

1. Built-in function
2. Userdefine function.

Built In Functions:

These functions are also called “library function” .we need to include header file to use these functions.

For Example:
Printf();- this function is used to print any statement.
Scanf();- this function is used to read any value from the user.
Strcpy();- this function is used to copy one string to another string.
Strlen();- this function is used to find the length of the string.

User Define Functions: This type of functions are created by programmers.

Types of User Define function

1. With return type and with parameter
2. With return type and without parameter
3. Without return type and with parameter
4. Without return type and without parameter

Benefit of using function

- C programming language supports function to provide modularity to the software.
- One of major advantage of function is it can be executed as many time as necessary from different points in your program so it helps you avoid duplication of work.
- By using function, you can divide complex tasks into smaller manageable tasks and test them independently before using them together.
- In software development, function allows sharing responsibility between team members in software development team. The whole team can define a set of standard function interface and implement it effectively.

Function body

Function body is the place you write your own source code. All variables declare in function body and parameters in parameter list are local to function or in other word have function scope.

Return statement

Return statement returns a value to where it was called. A copy of return value being return is made automatically. A function can have multiple return statements. If the void keyword used, the function don’t need a return statement or using return; the syntax of return statement is return expression;

Structure of function

Function header

The general form of function header is:
return_type function_name(parameter_list)
Function header consists of three parts:

- Data type of return value of function; it can be any legal data type such as int, char, pointer… if the function does not return a value, the return type has to be specified by keyword void.
- Function name; function name is a sequence of letters and digits, the first of which is a letter, and where an underscore counts as a letter. The name of a function should meaningful and express what it does, for example bubble_sort,
- And a parameter list; parameter passed to function have to be separated by a semicolon, and each parameter has to indicate it data type and parameter name. Function does not require formal parameter so if you don't pass any parameter to function you can use keyword void.

Here are some examples of function header:

```
void sort(int a[], int size);
bool authenticate(char username,char password)
```

How to use the function

Before using a function we should declare the function so the compiler has information of function to check and use it correctly. The function implementation has to match the function declaration for all part return data type, function name and parameter list. When you pass the parameter to function, they have to match data type, the order of parameter.

Source code example

```
#include<stdio.h>
void swap(int *x, int *y);
void main()
{
   int x = 10;
   int y = 20;
   printf("x,y before swapping\n");
   printf("x = %d\n",x);
   printf("y = %d\n",y);
   // using function swap
   swap(&x,&y);
   printf("x,y after swapping\n");
   printf("x = %d\n",x);
   printf("y = %d\n",y);
}
  void swap(int *x, int *y){
            int temp = *x;
           *x = *y;
           *y = temp;
}
```

Output:

```
x,y before swapping
x = 10
y = 20
x,y after swapping
x = 20
y = 10
Press any key to continue
```

Passing Values to Function

Pass by value

With this mechanism, all arguments you pass to function are copied into copy versions and your function work in that copy versions. So parameters does not affects after the function finished.

Pass by pointer

In some programming contexts, you want to change arguments you pass to function. In this case, you can use pass by pointer mechanism. Remember that a pointer is a memory address of a variable. So when you pass a pointer to a function, the function makes a copy of it and changes the content of that memory address, after function finish the parameter is changed with the changes in function body.

Pass an array to function

C allows you pass an array to a function, in this case the array is not copied. The name of array is a pointer which points to the first entry of it. And this pointer is passed to the function when you pass an array to a function. Source code example of various way to pass arguments to function

```
#include <stdio.h>
#include<conio.h>
double getAverage(int arr[], int size);
 int main ()
{
                    int balance[5] = {1000, 2, 3, 17, 50};
                    double avg;
                    avg = getAverage( balance, 5 ) ;
                    printf( "Average value is: %f ", avg );
                    getch();
                    return 0;
  }
  double getAverage(int arr[], int size)
  {
                    int    i;
                    double avg;
                    double sum;
                    for (i = 0; i < size; ++i)
                    {
                             sum += arr[i];
                    }
                    avg = sum / size;
                    return avg;
  }
```

Why Recursive Function

Recursive function allows you to divide your complex problem into identical single simple cases which can handle easily. This is also a well-known computer programming technique: divide and conquer.

Note of Using Recursive Function

Recursive function must have at least one exit condition that can be satisfied. Otherwise, the recursive function will call itself repeatly until the runtime stack overflows.

Example of Using Recursive Function

Recursive function is closely related to definitions of functions in mathematics so we can solving factorial problems using recursive function.

All you know in mathematics the factorial of a positive integer N is defined as follows:

N! = N*(N-1)*(N-2)…2*1;

Or in a recursive way:

N! = 1 if N <=1 and N*(N-1)! if N > 1

Example:

```
# include<stdio.h>
int factorial( int number)
{
        if(number <= 1)
                return 1;
        return number * factorial(number - 1);
}
void main()
{
        int x = 5;
        printf("Factorial of %d is %d",x,factorial(x));
}
```

Output:

```
Factorial of 5 is 120
```

Structures and unions:

In this tutorial you will learn about C Programming - Structures and Unions, initializing structure, assigning values to members, functions and structures, passing structure to functions, passing entire function to functions, arrays of structure, structure within a structure and union.
Structures are slightly different from the variable types you have been using till now. Structures are data types by themselves. When you define a structure or union, you are creating a custom data type.

Structures

Structures in C are used to encapsulate, or group together different data into one object. You can define a Structure as shown below:

```
struct object {
         char id[20];
         int xpos;
         int ypos;
       };
```

Structures can group data of different types as you can see in the example of a game object for a video game. The variables you declare inside the structure are called data members.

Initializing a Structure

Structure members can be initialized when you declare a variable of your structure:
struct object player1 = {“player1”, 0, 0};

The above declaration will create a struct object called player1 with an id equal to “player1”, xpos equal to 0, and ypos equal to 0. To access the members of a structure, you use the “.” (Scope resolution) operator. Shown below is an example of how you can accomplish initialization by assigning values using the scope resolution operator?

```
struct object player1;
       player1.id = “player1”;
       player1.xpos = 0;
       player1.ypos = 0;
```

Arrays of Structure

Since structures are data types that are especially useful for creating collection items, why not make a collection of them using an array? Let us now modify our above example object1.c to use an array of structures rather than individual ones.

```
        #include <stdio.h>
       #include <stdlib.h>
                struct object {
                char id[20];
                int xpos;
                int ypos;
              };
 struct object createobj(char id[], int xpos, int ypos);
 void printobj(struct object obj);
 void main() {
   int i;
  struct object gameobjs[2];
  gameobjs[0] = createobj("player1", 0, 0);
  gameobjs[1] = createobj("enemy1", 2, 3);
 for (i = 0; i < 2; i++)
   printobj(gameobjs[i]);
 //update enemy1 position
  gameobjs[1].xpos = 1;
  gameobjs[1].ypos = 2;
 for (i = 0; i < 2; i++)
   printobj(gameobjs[i]);
 }
struct object createobj(char id[], int xpos, int ypos)
 { struct object newobj;
  strcpy(newobj.id, id);
 newobj.xpos = xpos;
  newobj.ypos = ypos;
  return newobj;
 }
void printobj(struct object obj) {
  printf("name: %s, ", obj.id);
  printf("x position: %d, ", obj.xpos);
  printf("y position: %d", obj.ypos);
  printf("\n");
}
```

We create an array of structures called gamobjs and use the createobj function to initilize it's elements. You can observer that there is not much difference between the two programs. We added an update for the enemy1's position to show how to access a structure's members when it is an element within an array.

Structure within a Structure

Structures may even have structures as members. Imagine our x, y coordinate pair is a structure called coordinates. We can redeclare our object structure as follows:

```
struct object
{
    char id[20];
    struct coordinates loc;
};
```

You can still initialize these by using nested braces, like so:
struct object player1 = {“player1”, {0, 0}};
To access or set members of the above internal structure you would do like this:

```
 {geshibot language="c"}      struct object player1;
        player1.id = “player1”;
        player1.loc.xpos = 0;
        player1.loc.ypos = 0;
```

You simply add one more level of scope resolution.

Unions

Unions and Structures are identical in all ways, except for one very important aspect. Only one element in the union may have a value set at any given time. Everything we have shown you for structures will work for unions, except for setting more than one of its members at a time. Unions are mainly used to conserve memory. While each member within a structure is assigned its own unique storage area, the members that compose a union share the common storage area within the memory. Unions are useful for application involving multiple members where values are not assigned to all the members at any one time.

```
 union deadoralive
{
        int alive;
        int dead;
}
```

DIFFERENCE BETWEEN STRUCTURE AND UNION

All the members of the structure can be accessed at once, where as in an union only one member can be used at a time. Another important difference is in the size allocated to a structure and a union.

For eg: struct example

```
        {
                int integer;
                float floating_numbers;
        };
```

The size allocated here is sizeof(int)+sizeof(float); where as in an union union example { int integer; float floating_numbers; } size allocated is the size of the highest member. so size is=sizeof(float); The compiler allocates a piece of storage that is large enough to access a union member we can use the same syntax that we use to access structure members. That is

code.m
code.p
code.c
Are all valid member variables. During accessing we should make sure that we are accessing the member whose value is currently stored. For example a statement such as

```
code.m=456;
code.p=456.78;
printf("%d",code.m);
```

Would produce erroneous result.

In effect a union creates a storage location that can be used by one of its members at a time. When a different number is assigned a new value the new value supercedes the previous members value. Unions may be used in all places where a structure is allowed. The notation for accessing a union member that is nested inside a structure remains the same as for the nested structure.

Functions and Structures

Since structures are of custom data types, functions can return structures and also take them as arguments. Keep in mind that when you do this, you are making a copy of the structure and all it's members so it can be quite memory intensive To return a structure from a function declare the function to be of the structure type you want to return. In our case a function to initialize our object structure might look like this:

```
struct object createobj(char id[], int xpos, int ypos) {
```

```
    struct object newobj;
    strcpy(newobj.id, name);
    newobj.xpos = xpos;
    newobj.ypos = ypos;
    return newobj;
  }
```

Pass Structure to a Function

Let us now learn to pass a structure to a function. As an example let us use a function that prints embers of the structure passed to it:

```
void printobj(struct object obj) {
    printf("name: %s, ", obj.id);
    printf("x position: %d, ", obj.xpos);
    printf("y position: %d", obj.ypos);
    printf("n");
  }
```

For completeness we shall include the full source of the above examples so you may see how it all fits together.

```
#include <stdio.h>
#include <stdlib.h>
struct object {
char id[20];
int xpos;
int ypos;
};
struct object createobj(char id[], int xpos, int ypos);
void printobj(struct object obj);
void main() {
        struct object player1 = createobj("player1", 0, 0);
        struct object enemy1 = createobj("enemy1", 2, 3);
```

Output:

```
Name: player1, x position: 0, y position: 0
Name: enimy1, x position: 2, y position: 3
```

```
		printobj(player1);
		printobj(enemy1);getch();
	}
	struct object createobj(char id[], int xpos, int ypos){
		struct object newobj;
		strcpy(newobj.id, id);
		newobj.xpos = xpos;
		newobj.ypos = ypos;
		return newobj;
	}
	void printobj(struct object obj) {
		printf("name: %s, ", obj.id);
		printf("x position: %d, ", obj.xpos);
		printf("y position: %d", obj.ypos);
		printf("n");
	}
```

Pointer

Pointers are used everywhere in C, so if you want to use the C language fully you have to have a very good understanding of pointers. They have to become *comfortable* for you. The goal of this section and the next several that follow is to help you build a complete understanding of pointers and how C uses them. For most people it takes a little time and some practice to become fully comfortable with pointers, but once you master them you are a full-fledged C programmer.

C uses pointers in three different ways:

- C uses pointers to create **dynamic data structures** -- data structures built up from blocks of memory allocated from the heap at run-time.
- C uses pointers to handle **variable parameters** passed to functions.
- Pointers in C provide an alternative way to **access information stored in arrays**. Pointer techniques are especially valuable when you work with strings. There is an intimate link between arrays and pointers in C.

In some cases, C programmers also use pointers because they make the code slightly more efficient. What you will find is that, once you are completely comfortable with pointers, you tend to use them all the time. We will start this discussion with a basic introduction to pointers and the concepts surrounding pointers, and then move on to the three techniques described above. Especially on this article, you will want to read things twice. The first time through you can learn all the concepts. The second time through you can work on binding the concepts together into an integrated whole in your mind. After you make your way through the material the second time, it will make a lot of sense. C pointer is a memory address. When you define a variable for example:

```
int x = 10;
```

You specify variable name (x), its data type (integer in this example) and its value is 10. The variable x resides in memory with a specified memory address. To get the memory address of variable x, you use operator & before it. This code snippet print memory address of x printf("memory address of x is %d\n",&x);

and in my PC the output is memory address of x is 1310588 Now you want to access memory address of variable x you have to use pointer. After that you can access and modify the content of memory address which pointer point to. In this case the memory address of x is 1310588 and its content is 10. To declare pointer you use asterisk notation (*) after pointer's data type and before pointer name as follows:

int *px;

Now if you want pointer px to points to memory address of variable x, you can use address-of operator (&) as follows:
int *px = &x;
After that you can change the content of variable x for example you can increase, decrease x value :

```
*px += 10;
        printf("value of x is %d\n",x);
 *px -= 5;
 printf("value of x is %d\n",x);
```

and the output indicates that x variable has been change via pointer px.

```
value of x is 20
value of x is 15
```

It is noted that the operator (*) is used to dereference and return content of memory address.
In some programming contexts, you need a pointer which you can only change memory address of it but value or change the value of it but memory address. In this cases, you can use *const* keyword to define a pointer points to a constant integer or a constant pointer points to an integer as follows:

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int c = 10,c2 = 20,y = 10,y2 = 20;
        int const *py = &y;
        const int *pc = &c;
        clrscr();
        printf("value of pc is %d\n",*pc);
        pc = &c2;
        printf("value of pc is %d\n",*pc);
        *py++;
        printf("value of y is %d\n",y);
        getch();
}
```

Output:

```
value of pc is 10
value of pc is 20
value of y is 10
```

Dynamic Memory Allocation

malloc, calloc, or realloc are the three functions used to manipulate memory. These commonly used functions are available through the stdlib library so you must include this library in order to use them.

#include <stdlib.h>

After including the stdlib library you can use the malloc, calloc, or realloc functions to manipulate chunks of memory for your variables.

Dynamic Memory Allocation Process

When a program executes, the operating system gives it a stack and a heap to work with. The stack is where global variables, static variables, and functions and their locally defined variables reside. The heap is a free section for the program to use for allocating memory at runtime.

Allocating a Block of Memory

Use the malloc function to allocate a block of memory for a variable. If there is not enough memory available, malloc will return NULL.

The prototype for malloc is:

```
void *malloc(size_t size);
```

Do not worry about the size of your variable, there is a nice and convenient function that will find it for you, called sizeof. Most calls to malloc will look like the following example:

ptr = (struct mystruct*)malloc(sizeof(struct mystruct));

This way you can get memory for your structure variable without having to know exacly how much to allocate for all its members as well. Allocating Multiple Blocks of Memory You can also ask for multiple blocks of memory with the calloc function:

void *calloc(size_t num, size_t size);

If you want to allocate a block for a 10 char array, you can do this:

```
char *ptr;

ptr = (char *)calloc(10, sizeof(char));
```

The above code will give you a chunk of memory the size of 10 chars, and the ptr variable would be pointing to the beginning of the memory chunk. If the call fails, ptr would be NULL.

Releasing the Used Space

All calls to the memory allocating functions discussed here, need to have the memory explicitly freed when no longer in use to prevent memory leaks. Just remember that for every call to an *alloc function you must have a corresponding call to free. The function call to explicitly free the memory is very simple and is written as shown here below:

free(ptr);

Just pass this function the pointer to the variable you want to free and you are done.To Alter the Size of Allocated Memory Lets get to that third memory allocation function, realloc.

void *realloc(void *ptr, size_t size);

Pass this function the pointer to the memory you want to resize and the new size you want to resize the allocated memory for the variable you want to resize. Here is a simple and trivial example to give you a quick idea of how you might see calloc and realloc in action. You will have many chances for malloc viewing as it is the most popular of the three by far.

```
#include <stdio.h>
#include <stdlib.h>
void main() {
        char *ptr, *retval;
        ptr = (char *)calloc(10, sizeof(char));
        if (ptr == NULL)
               printf("calloc failed\n");
        else
                printf("calloc successful\n");
        retval = realloc(ptr, 5);
        if (retval == NULL)
                printf("realloc failed\n");
        else
                printf("realloc successful\n");
        free(ptr);
        free(retval);
}
```

First we declared two pointers and allocated a block of memory the size of 10 chars for ptr using the calloc function. The second pointer retval is used for getting the return value from the call to realloc. Then we reallocate the size of ptr to 5 chars instead of 10. After we check whether all went well with that call, we free up both pointers. You can play around with the values of size passed to either of the memory allocation functions to see how big a chunk you can ask for before it fails on you. Do not worry, your operating system has the ability to keep your program in check, you will not hurt it this way.

Linked List:

Linked lists are a type of data structure for storing information as a list. They are a memory efficient alternative to arrays because the size of the list is only ever as large as the data. Plus they do not have to shift data and recopy when resizing as dynamic arrays do. They do have the extra overhead of 1 or 2 pointers per data node, so they make sense only with larger records. You would not want to store a list of integers in a linked list because it would actually double your memory overhead over the same list in an array. There are three different types of linked lists, but the other two are just variations of the basic singly linked list. If you understand this linked list then you will be able to handle other two types of lists.

Advantages of Linked Lists

A linked list is a dynamic data structure and therefore the size of the linked list can grow or shrink in size during execution of the program. A linked list does not require any extra space therefore it does not waste extra memory. It provides flexibility in rearranging the items efficiently. The limitation of linked list is that it consumes extra space when compared to a array since each node must also contain the address of the next item in the list to search for a single item in a linked list is cumbersome and time consuming process.

Linked lists Types

There are 4 different kinds of linked lists:

1. Linear singly linked list
2. Circular singly linked list
3. Two way or doubly linked list
4. Circular doubly linked list.

Linked List Nodes

A linked list gets their name from the fact that each list node is "linked" by a pointer. A linked list node is comparable to an array element. A node contains a data portion and a pointer portion, and is declared as structs in C. As an example, we can implement a list of high scores for a game as a linked list. Let us now declare a node: We have two variables that make up the data portion of the node, and then the pointer to the next node in the list.

Creating Linked Lists

A linked list always has a base node that is most commonly called a head, but some call it as a root. An empty linked list will have this node and will be set to NULL. This goes for all three types of linked lists. The last node in a linked list always points to NULL. Let us declare our linked list for implementing high scores list.

```
struct llnode *head = NULL;
```

Here we just declared a regular pointer variable of the node struct we declared, and then set the head to NULL to indicate the list is empty.

Traversing a Linked List

Moving through a linked list and visiting all the nodes is called traversing the linked list. There is more than one way to encounter segmentation faults when traversing a linked list, but if you are careful and follow 2 basic checks you will be able to handle segmentation faults. To traverse a singly linked list you create a pointer and set it to head. Often called the current node as a reminder that it is keeping track of the current node. Always make sure to check that head is not NULL before trying to traverse the list or you will get a segmentation fault. You may also need to check that the next pointer of the current node is not NULL, if not, you will go past the end of the list and create a segmentation fault. Here is a failsafe way of traversing a singly linked list:

```
 if (head != NULL) {

          while (currentnode->next != NULL) {

            currentnode = currentnode->next;

          }        }
```

We first check that the head is not NULL and if so, as long as the current pointer's next is not NULL, we set current equal to the next node effectively moving through the entire list until we do encounter a next pointer that is NULL.
If you follow that formula, you will have no problems with segmentation faults during traversal.

File Management In C

In this tutorial you will learn about C Programming - File management in C, File operation functions in C, Defining and opening a file, Closing a file, The getw and putw functions, The fprintf & fscanf functions, Random access to files and fseek function.

C supports a number of functions that have the ability to perform basic file operations, which include:
1. Naming a file
2. Opening a file
3. Reading from a file
4. Writing data into a file
5. Closing a file

- Real life situations involve large volume of data and in such cases, the console oriented I/O operations pose two major problems
- It becomes cumbersome and time consuming to handle large volumes of data through terminals.
- The entire data is lost when either the program is terminated or computer is turned off therefore it is necessary to have more flexible approach where data can be stored on the disks and read whenever necessary, without destroying the data. This method employs the concept of files to store data.

File Operation function in C:

Function Name	**Operation**
fopen()	Creates a new file for use Opens a new existing file for use
fclose	Closes a file which has been opened for use

getc()	Reads a character from a file
putc()	Writes a character to a file
fprintf()	Writes a set of data values to a file
fscanf()	Reads a set of data values from a file
getw()	Reads a integer from a file
putw()	Writes an integer to the file
fseek()	Sets the position to a desired point in the file
ftell()	Gives the current position in the file
rewind()	Sets the position to the begining of the file

Defining and opening a file:

If we want to store data in a file into the secondary memory, we must specify certain things about the file to the operating system. They include the filename, data structure, purpose. The general format of the function used for opening a file is

```
FILE *fp;
fp=fopen("filename","mode");
```

The first statement declares the variable fp as a pointer to the data type FILE. As stated earlier, File is a structure that is defined in the I/O Library. The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp. This pointer, which contains all the information about the file, is subsequently used as a communication link between the system and the program.
The second statement also specifies the purpose of opening the file. The mode does this job.
R open the file for read only.
W open the file for writing only.
A open the file for appending data to it.
Consider the following statements:

```
FILE *p1, *p2;
p1=fopen("data","r");
p2=fopen("results","w");
```

In these statements the p1 and p2 are created and assigned to open the files data and results respectively the file data is opened for reading and result is opened for writing. In case the results file already exists, its contents are deleted and the files are opened as a new file. If data file does not exist error will occur.

Closing a file:

The input output library supports the function to close a file; it is in the following format.
fclose(file_pointer);
A file must be closed as soon as all operations on it have been completed. This would close the file associated with the file pointer. Observe the following program. ….

```
FILE *p1 *p2;
p1=fopen ("Input","w");
```

p2=fopen (“Output”,”r”);
….
…
fclose(p1);
fclose(p2)

The above program opens two files and closes them after all operations on them are completed, once a file is closed its file pointer can be reversed on other file. The getc and putc functions are analogous to getchar and putchar functions and handle one character at a time. The putc function writes the character contained in character variable c to the file associated with the pointer fp1. ex putc(c,fp1); similarly getc function is used to read a character from a file that has been open in read mode. c=getc(fp2).

```
#include<stdio.h>
void main()
{
        FILE *f1;
        char c;
        clrscr();
        printf("Data input output");
        f1=fopen("Input","w");
        while ((c=getchar())!='0')
        {       putc(c,f1);     }
        fclose(f1);
        printf("\nData output\n");
        f1=fopen ("INPUT","r");
        while ((c=getc(f1))!=EOF)
        {
                printf("%c",c);
        }
         fclose(f1);
}
```

The getw and putw functions:

These are integer-oriented functions. They are similar to get c and putc functions and are used to read and write integer values. These functions would be usefull when we deal with only integer data. The general forms of getw and putw are:

```
putw(integer,fp);
getw(fp);

        #include<stdio.h>
        void main()
        {
                FILE *f1,*f2,*f3;
                int number,i;
                printf("\n MYDATA file contents:");
                f1=fopen("MYDATA","w");
                for (i=1;i<10;i++)    {
                        scanf("%d",&number);
                        if(number==-1)
                        break;
               putw(number,f1);   }
               fclose(f1);
              f1=fopen("MYDATA","r");
                f2=fopen("ODD","w");
                f3=fopen("EVEN","w");
                while ((number=getw(f1))!=EOF)
                {
                        if (number%2==0)
                                putw(number,f3);
                        else
                                putw(number,f2);
                }
                fclose(f1);
                fclose(f2);
                fclose(f3);
                f2=fopen("ODD","r");
                f3=fopen("EVEN","r");
                printf("\n contents of ODD file:  ");
                while ((number=getw(f2))!=EOF)
                       printf("%d  ",number);
                       printf("\n Contents of File EVEN:  ");
                while ((number=getw(f3))!=EOF)
                      printf("%d  ",number);
                fclose(f2);
                fclose(f3);
                getch();
        }
```

The fprintf & fscanf functions:

The fprintf and fscanf functions are identical to printf and scanf functions except that they work on files. The first argument of these functions is a file pointer which specifies the file to be used. The general form of fprintf is

fprintf(fp,"control string", list);

Where fp id a file pointer associated with a file that has been opened for writing. The control string is file output specifications list may include variable, constant and string.

fprintf(f1,%s%d%f",name,age,7.5);

Here name is an array variable of type char and age is an int variable the general format of fscanf is

fscanf(fp,"controlstring",list);

This statement would cause the reading of items in the control string.

Example:

fscanf(f2,"%s%d",item,&quantity");

```
        #include <stdio.h>
```

```
#include <conio.h>
#include <stdlib.h>
int main(void)
{
        FILE *fp;
        char s[80];
        int t;
        if((fp=fopen("test", "w")) == NULL) {
                printf("Cannot open file.\n");
                exit(1);
        }
        printf("Enter a string and a number: ");
        fscanf(stdin, "%s%d", s, &t); /* read from keyboard */
        fprintf(fp, "%s %d", s, t); /* write to file */
        fclose(fp);
        if((fp=fopen("test","r")) == NULL) {
                printf("Cannot open file.\n");
                exit(1);
        }
        fscanf(fp, "%s%d", s, &t); /* read from file */
        fprintf(stdout, "%s %d", s, t); /* print on screen */
        getch();
        return 0;
}
```

Random access to files:

Sometimes it is required to access only a particular part of the and not the complete file. This can be accomplished by using the following function:

1 > fseek

fseek function:
The general format of fseek function is a s follows:
fseek(file pointer,offset, position);

This function is used to move the file position to a desired location within the file. Fileptr is a pointer to the file concerned. Offset is a number or variable of type long, and position in an integer number. Offset specifies the number of positions (bytes) to be moved from the location specified bt the position. The position can take the 3 values.

Value	Meaning
0	beginning of the file
1	Current position
2	End of the file.

C Language - The Preprocessor

In this tutorial you will learn about C Language - The Preprocessor, Preprocessor directives, Macros, #define identifier string, Simple macro substitution, Macros as arguments, Nesting of macros, Undefining a macro and File inclusion.

The Preprocessor

A unique feature of c language is the preprocessor. A program can use the tools provided by preprocessor to make his program easy to read, modify, portable and more efficient. Preprocessor is a program that processes the code before it passes through the compiler. It operates under the control of preprocessor command lines and directives. Preprocessor directives are placed in the source program before the main line before the source code passes through the compiler it is examined by the preprocessor for any preprocessor directives. If there is any appropriate actions are taken then the source program is handed over to the compiler. Preprocessor directives follow the special syntax rules and begin with the symbol #bin column1 and do not require any semicolon at the end. A set of commonly used preprocessor directives.

Preprocessor directives:

Directive	**Function**
#define	Defines a macro substitution
#undef	Undefined a macro
#include	Specifies a file to be included
#ifdef	Tests for macro definition
#endif	Specifies the end of #if
#ifndef	Tests whether the macro is not def
#if	Tests a compile time condition
#else	Specifies alternatives when # if test fails

The preprocessor directives can be divided into three categories
1. Macro substitution division
2. File inclusion division
3. Compiler control division

Macros:

Macro substitution is a process where an identifier in a program is replaced by a pre defined string composed of one or more tokens we can use the #define statement for the task. It has the following form

#define identifier string

The preprocessor replaces every occurrence of the identifier int the source code by a string. The definition should start with the keyword #define and should follow on identifier and a string with at least one blank space between them. The string may be any text and identifier must be a valid c name. There are different forms of macro substitution. The most common form is

1. Simple macro substitution
2. Argument macro substitution
3. Nested macro substitution

Simple macro substitution:

Simple string replacement is commonly used to define constants example: #define pi 3.1415926 Writing macro definition in capitals is a convention not a rule a macro definition can include more than a simple constant value it can include expressions as well. Following are valid examples:

#define AREA 12.36

Macros as arguments:

The preprocessor permits us to define more complex and more useful form of replacements it takes the following form.
# define identifier (f1, f2, f3…..fn) string.
Notice that there is no space between identifier and left parentheses and the identifier f1,f2,f3 …. Fn is analogous to formal arguments in a function definition. There is a basic difference between simple replacement discussed above and replacement of macro arguments is known as a macro call A simple example of a macro with arguments is
# define CUBE (x) (x*x*x) If the following statements appears later in the program,
volume=CUBE(side); The preprocessor would expand the statement to
volume =(side*side*side)
Nesting of macros:

We can also use one macro in the definition of another macro. That is macro definitions may be nested. Consider the following macro definitions
# define SQUARE(x)((x)*(x))

Undefined a macro:

A defined macro can be undefined using the statement # undef identifier. This is useful when we want to restrict the definition only to a particular part of the program.

File inclusion:

The preprocessor directive "#include file name” can be used to include any file in to your program if the function s or macro definitions are present in an external file they can be included in your file In the directive the filename is the name of the file containing the required definitions or functions alternatively the this directive can take the form
#include< filename >
Without double quotation marks. In this format the file will be searched in only standard directories.
The c preprocessor also supports a more general form of test condition #if directive. #if constant expression
{
statement-1;
statemet2’

… }
#endif the constant expression can be a logical expression such as test < = 3 etc
If the result of the constant expression is true then all the statements between the #if and #endif are included for processing otherwise they are skipped. The names TEST LEVEL etc., may be defined as macros.

Introduction to Graphics In C

What Is C Graphics?

- This C Graphics is for those who want to learn fundamentals of Graphics programming, without any prior knowledge of graphics.
- With the help of the C language, programs which create computer graphics can be made.
- This contains lots of fundamental graphics program like drawing of various geometrical shapes(rectangle, circle eclipse etc), use of mathematical function in drawing curves, coloring an object with different colors and patterns and simple animation programs like jumping ball and moving cars.
- It's not like traditional C programming in which you have to apply complex logic in your program and then you end up with a lot of errors and warnings in your program.

What is Computer Graphics??

- Computer graphics is an art of drawing pictures, lines, charts, etc using computers with the help of programming.
- Computer graphics is made up of number of pixels. Pixel is the smallest graphical picture or unit represented on the computer screen.
- Basically there are two types of computer graphics namely:.
- **Interactive Computer Graphics** involves a two way communication between computer and user. Here the observer is given some control over the image by providing him with an input device for example the video game controller of the ping pong game. This helps him to signal his request to the computer.
- **Non Interactive Computer Graphics** otherwise known as passive computer graphics. it is the computer graphics in which user does not have any kind of control over the image.
- Image is merely the product of static stored program and will work according to the instructions given in the program linearly.The image is totally under the control of program instructions not under the user. Example: screen savers.

Working With C Graphics

- C graphics using graphics.h functions or WinBGIM (Windows 7) can be used to draw different shapes, display text in different fonts, change colors and many more. Using functions of graphics.h in turbo c compiler you can make graphics programs, animations, projects and games.

Sample C Graphics Program

```
#include<graphics.h>
```

```
#include<conio.h>
void main() {
        int gd = DETECT, gm;
        initgraph(&gd, &gm, "c:\\tc\\bgi");
        circle(300, 300, 50);
        getch(); closegraph(); }
```

Requirement to Run This Program

- Graphics.h Header File
- Graphics.lib library file
- Graphics driver (BGI file)

Header File : graphics.h

- All Graphical Functions are Included in Graphics.h
- After Including graphics.h Header File [You can get access graphical functions]

Graphics mode Initialization

- First of all we have to call the initgraph function that will initialize the graphics mode on the computer. initigraph has the following prototype.

```
void initgraph(int far *graphdriver, int far *graphmode, char far *pathtodriver);
```

- Initgraph initializes the graphics system by loading the graphics driver from disk (or validating a registered driver) then putting the system into graphics mode.
- Initgraph also resets all graphics settings (color, palette, current position, viewport, etc.) to their defaults, then resets graphresult to 0.

***graphdriver**

- Integer that specifies the graphics driver to be used.
- We can give graphdriver a value using a constant of the graphics_drivers enumeration type whcih is listed in graphics.h.
- Normally we use value as &#8220;0&#8221; (requests auto-detect).
- Other values are 1 to 10 and description of each enumeration type is listed here

graphics_drivers constant	**Numeric value**
DETECT	0 (requests autodetect)
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7

ATT400	8
VGA	9
PC3270	10

***graphmode**

- Integer that specifies the initial graphics mode (unless *graphdriver = DETECT).
- If *graphdriver = DETECT, initgraph sets *graphmode to the highest resolution available for the detected driver.
- We can give *graphmode a value using a constant of the graphics_modes enumeration type and description of each enumeration type is listed here

Graphics	Columns				
Driver	graphics_mode	Value	x Rows	Palette	Pages
CGA	CGAC0	0	320 x 200	C0	1
	CGAC1	1	320 x 200	C1	1
	CGAC2	2	320 x 200	C2	1
	CGAC3	3	320 x 200	C3	1
	CGAHI	4	640 x 200	2 color	1
MCGA	MCGAC0	0	320 x 200	C0	1
	MCGAC1	1	320 x 200	C1	1
	MCGAC2	2	320 x 200	C2	1
	MCGAC3	3	320 x 200	C3	1
	MCGAMED	4	640 x 200	2 color	1
	MCGAHI	5	640 x 480	2 color	1
EGA	EGALO	0	640 x 200	16 color	4
	EGAHI	1	640 x 350	16 color	2
EGA64	EGA64LO	0	640 x 200	16 color	1
	EGA64HI	1	640 x 350	4 color	1
EGA-MONO	EGAMONOHI	3	640 x 350	2 color	1 w/64K
	EGAMONOHI	3	640 x 350	2 color	2 w/256K
HERC	HERCMONOHI	0	720 x 348	2 color	2
ATT400	ATT400C0	0	320 x 200	C0	1
	ATT400C1	1	320 x 200	C1	1
	ATT400C2	2	320 x 200	C2	1
	ATT400C3	3	320 x 200	C3	1
	ATT400MED	4	640 x 200	2 color	1
	ATT400HI	5	640 x 400	2 color	1
VGA	VGALO	0	640 x 200	16 color	2
	VGAMED	1	640 x 350	16 color	2

	VGAHI	2	640 x 480	16 color	1
PC3270	PC3270HI	0	720 x 350	2 color	1
IBM8514	IBM8514HI	0	640 x 480	256 color	?
	IBM8514LO	0	1024 x 768	256 color	?

***pathtodriver**

- Specifies the directory path where initgraph looks for graphics drivers (*.BGI) first.
- If they&#8217;re not there, initgraph looks in the current directory. If pathtodriver is null, the driver files must be in the current directory.
- *graphdriver and *graphmode must be set to valid graphics drivers and graphics mode values or you&#8217;ll get unpredictable results. (The exception is graphdriver = DETECT.)
- After a call to initgraph, *graphdriver is set to the current graphics driver, and *graphmode is set to the current graphics mode.
- We can tell initgraph to use a particular graphics driver and mode, or to auto detect the attached video adapter at run time and pick the corresponding driver.
- If we tell initgraph to auto detect, it calls detectgraph to select a graphics driver and mode.
- Normally, initgraph loads a graphics driver by allocating memory for the driver (through _graphgetmem), then loading the appropriate .BGI file from disk. As an alternative to this dynamic loading scheme, you can link a graphics driver file (or several of them) directly into your executable program file.

closegraph() :

- This function switches back the screen from graphcs mode to text mode. It clears the screen also.
- A graphics program should have a closegraph function at the end of graphics. Otherwise DOS screen will not go to text mode after running the program.
- Here, closegraph() is called after getch() since screen should not clear until user hits a key.

C Graphics Functions:

arc: arc function is used to draw an arc with center (x,y) and stangle specifies starting angle, endangle specifies the end angle and last parameter specifies the radius of the arc. arc function can also be used to draw a circle but for that starting angle and end angle should be 0 and 360 respectively.

```
Declaration :- void arc(int x, int y, int stangle, int endangle, int radius);
```

Bar: Bar function is used to draw a 2-dimensional, rectangular filled in bar . Coordinates of left top and right bottom corner are required to draw the bar. Left specifies the X-coordinate of top left corner, top specifies the Y-coordinate of top left corner, right specifies the X-coordinate of right bottom corner, bottom specifies the Y-coordinate of right bottom corner. Current fill pattern and fill color is used to fill the bar. To change fill pattern and fill color use

```
Declaration :- void bar(int left, int top, int right, int bottom);
```

CircleCircle function is used to draw a circle with center (x,y) and third parameter specifies the radius of the circle. The code given below draws a circle.

```
Declaration :- void circle(int x, int y, int radius);
```

Cleardevice: cleardevice function clears the screen in graphics mode and sets the current position to (0,0). Clearing the screen consists of filling the screen with current background color.

```
Declaration :- void cleardevice();
```

Drawpoly: Drawpoly function is used to draw polygons i.e. triangle,

```
Declaration :- void drawpoly( int num, int *polypoints );
```

num indicates (n+1) number of points where n is the number of vertices in a polygon, polypoints points to a sequence of (n*2) integers . Each pair of integers gives x and y coordinates of a point on the polygon. We specify (n+1) points as first point coordinates should be equal to (n+1)

Ellipse: Ellipse is used to draw an ellipse (x,y) are coordinates of center of the ellipse, stangle is the starting angle, end angle is the ending angle, and fifth and sixth parameters specifies the X and Y radius of the ellipse. To draw a complete ellipse strangles and end angle should be 0 and 360 respectively

```
Declaration:void ellipse(int x, int y, int stangle, int endangle, int xradius, int yradius);
```

FloodFill: floodfill function is used to fill an enclosed area. Current fill pattern and fill color is used to fill the area.(x, y) is any point on the screen if (x,y) lies inside the area then inside will be filled otherwise outside will be filled,border specifies the color of boundary of area. To change fill pattern and fill color use setfillstyle.

```
Declaration :- void floodfill(int x, int y, int border);
```

getarccoords : getarccoords function is used to get coordinates of arc which is drawn most recently.

```
Declaration :- void getarccoords(struct arccoordstype *var);
```

getx : getx function returns the X coordinate of current position.

```
Declaration :- int getx();
```

Getpixel: getpixel function returns the color of pixel present at location(x, y)

```
Declaration :- int getpixel(int x, int y)
```

SetColor: Setcolor is used to set the color of the particular shapes(border color)

```
Declaration :- void setcolor(int color);
```

Settextstyle: Settextstyle function is used to change the way in which text appears, using it we can modify the size of text, change direction of text and change the font of text.font argument specifies the font of text, Direction can be HORIZ_DIR (Left to right) or VERT_DIR (Bottom to top).

```
Declaration :- void settextstyle( int font, int direction, int charsize);
```

Sample Programs:

Drawing Shapes

```
#include<graphics.h>
#include<conio.h>
main()
{
   int gd = DETECT,gm,left=100,top=100,right=200,bottom=200,x= 300,y=150,radius=50;
   initgraph(&gd, &gm, "C:\\TC\\BGI");
   rectangle(left, top, right, bottom);
   circle(x, y, radius);
   bar(left + 300, top, right + 300, bottom);
   line(left - 10, top + 150, left + 410, top + 150);
   ellipse(x, y + 200, 0, 360, 100, 50);
   outtextxy(left + 100, top + 325, "My First C Graphics Program");
   getch();
   closegraph();
   return 0;
}
```

Drawing pie chart

```
#include<graphics.h>
#include<conio.h>
 int main(){
   int gd = DETECT, gm, midx, midy;
   initgraph(&gd, &gm, "C:\\TC\\BGI");
   setcolor(MAGENTA);
   rectangle(0,40,639,450);
   settextstyle(SANS_SERIF_FONT,HORIZ_DIR,2);
   setcolor(WHITE);
   outtextxy(275,10,"Pie Chart");
   midx = getmaxx()/2;
   midy = getmaxy()/2;
```

```
  setfillstyle(LINE_FILL,BLUE);
  pieslice(midx, midy, 0, 75, 100);
  outtextxy(midx+100, midy - 75, "20.83%");
  setfillstyle(XHATCH_FILL,RED);
  pieslice(midx, midy, 75, 225, 100);
  outtextxy(midx-175, midy - 75, "41.67%");
  setfillstyle(WIDE_DOT_FILL,GREEN);
  pieslice(midx, midy, 225, 360, 100);
  outtextxy(midx+75, midy + 75, "37.50%");
  getch();
  return 0; }
```

Drawing concentric circles:

```
#include <graphics.h>
int main(){
  int gd = DETECT, gm;
  int x = 320, y = 240, radius;
  initgraph(&gd, &gm, "C:\\TC\\BGI");
  for ( radius = 25; radius <= 125 ; radius = radius + 20)
    circle(x, y, radius);
  getch();
  closegraph();
  return 0;
}
```

C graphics program moving car

```
#include <graphics.h>
#include <dos.h>
int main(){
  int i, j = 0, gd = DETECT, gm;
  initgraph(&gd,&gm,"C:\\TC\\BGI");
  settextstyle(DEFAULT_FONT,HORIZ_DIR,2);
  outtextxy(25,240,"Press any key to view the moving car");
  getch();
  for( i = 0 ; i <= 420 ; i = i + 10, j++ )
  {
    rectangle(50+i,275,150+i,400);
    rectangle(150+i,350,200+i,400);
    circle(75+i,410,10);
    circle(175+i,410,10);
    setcolor(j);
    delay(100);
    if( i == 420 )
      break;
    if ( j == 15 )
      j = 2;
    cleardevice(); // clear screen
  }
  getch();
  closegraph();
  return 0;
}
```

Introduction to C++

Created by Bjarne Stroustrup of AT&T Bell Labs as an extension of C, C++ is an object-oriented computer language used in the development of enterprise and commercial applications. Microsoft's Visual C++ became the premier language of choice among developers and programmers.

As a procedural programming language, C++ uses program structures such as i/o (input/output), assignment statement, iterative statements, conditional statements and subprograms. Data structures of C++ include integer, real, char, arrays, structs and pointers.

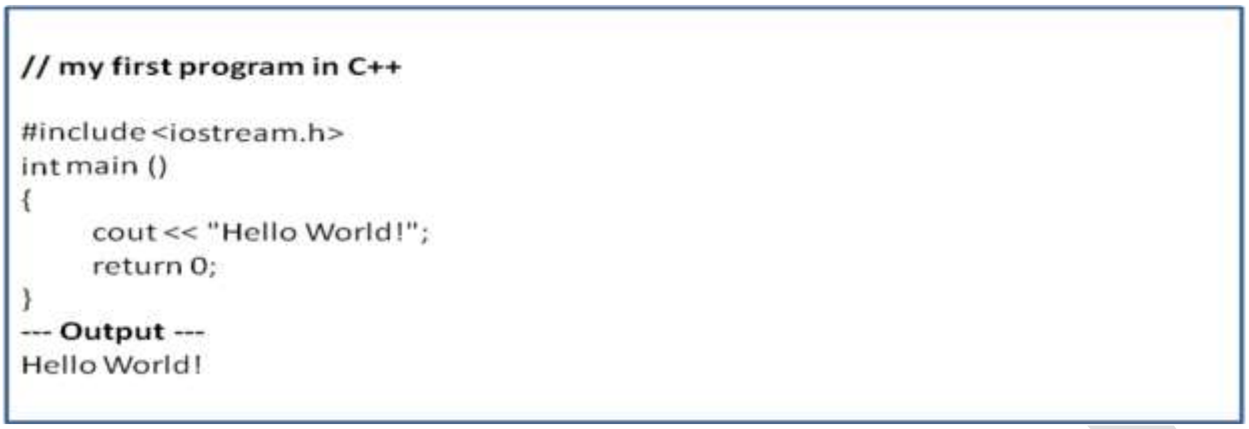
Employment opportunities are numerous and well paid for C++ programmers and developers looking to work in the field of Software Engineering or as an IT Professional. Oftentimes, C++ Professionals will also be familiar with C, Linux, Unix, Java, .NET and VB (Visual Basic). Developers working with C++ can expect to participate in a variety of programming opportunities: developing systems for trading applications for an Investment Bank, developing cutting edge software applications for groundbreaking new technologies (Smartphone, PDA, etc.) to creating applications for 3-D Imaging Software or spectroscopic systems.

C++ Tutorials available in this section include explanations for simple to more advanced concepts of C++ in detail with sample coding information. A new programmer or developer interested in learning about C++ programming language and finding out why C++ is one of the most widely used programming languages for creating large-scale applications can utilize the tutorials and articles on C++ made available in this section.

Basic Concept of OOP and Structure of C++ Program:

Probably the best way to start learning a programming language is by writing a program. Therefore, here is our

first program:



```
// my first program in C++

#include <iostream.h>
int main ()
{
        cout << "Hello World!";
        return 0;
}
--- Output ---
Hello World!
```

We are going to look line by line at the code we have just written:

// my first program in C++

This is a comment line. All lines beginning with two slash signs (//) are considered comments and do not have any effect on the behavior of the program. The programmer can use them to include short explanations or observations within the source code itself. In this case, the line is a brief description of what our program is.

#include <iostream.h>

Lines beginning with a hash sign (#) are directives for the preprocessor. They are not regular code lines with expressions but indications for the compiler's preprocessor. In this case the directive #include <iostream> tells the preprocessor to include the iostream standard file. This specific file (iostream) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

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, and in fact it will be included in most of the source codes included in these tutorials.

int main ()

This line corresponds to the beginning of the definition of the main function. The main function is the point by where all C++ programs start their execution, independently of its location within the source code.

- The // in first line is used for representing comment in the program.
- The second line of the program has a # symbol which represents the preprocessor directive followed by header file to be included placed between < >.
- The next structure present in the program is the class definition. This starts with the keyword class followed by class name employee. Within the class are data and functions.

The data defined in the class are generally private and functions are public. These explanations we will be detailed in later sections. The class declaration ends with a semicolon.

- main() function is present in all C++ programs.
- An object e1 is created in employee class. Using this e1 the functions present in the employee class are accessed and there by data are accessed.
- The input named ename and eno is received using the input statement named cin and the values are outputted using the output statement named cout.

Characteristics of c++

- Emphasis is on data rather than procedure
- Programs are divided into what are known as objects
- Data and Functions are tied together in the data structure
- Data is hidden and can not be accessed by external functions
- Objects may communicate with each other through functions
- New deta and functions can be easily added whenever necessary
- Bottom-up approach

Object Oriented Programming:

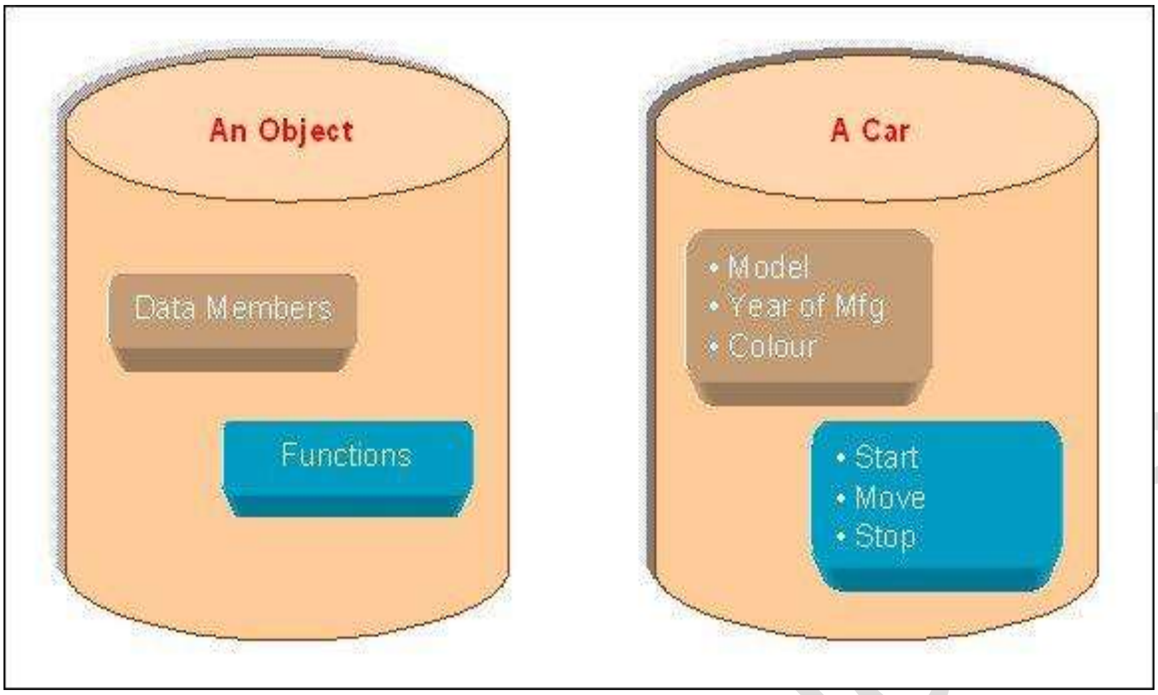
Before starting to learn C++ it is essential to have a basic knowledge of the concepts of Object oriented programming. Some of the important object oriented features are namely:

- Objects
- Classes
- Inheritance
- Data Abstraction
- Data Encapsulation
- Polymorphism
- Overloading
- Reusability

In order to understand the basic concepts in C++, a programmer must have a good knowledge of the basic terminology in object-oriented programming. Below is a brief outline of the concepts of object-oriented programming languages :

Objects:

Object is the basic unit of object-oriented programming. Objects are identified by its unique name. An object represents a particular instance of a class. There can be more than one instance of a class. Each instance of a class can hold its own relevant data.



An Object is a collection of data members and associated member functions also known as methods.

Classes:

Classes are data types based on which objects are created. Objects with similar properties and methods are grouped together to form a Class. Thus a Class represents a set of individual objects. Characteristics of an object are represented in a class as Properties. The actions that can be performed by objects become functions of the class and are referred to as Methods.

For example consider we have a Class of Cars under which Santro Xing, Alto and WaganR represents individual Objects. In this context each Car Object will have its own, Model, Year of Manufacture, Color, Top Speed, Engine Power etc., which form Properties of the Car class and the associated actions i.e., object functions like Start, Move, and Stop form the Methods of Car Class.

No memory is allocated when a class is created. Memory is allocated only when an object is created, i.e., when an instance of a class is created.

Inheritance:

Inheritance is the process of forming a new class from an existing class or base class. The base class is also known as parent class or super class. The new class that is formed is called derived class. Derived class is also known as a child class or sub class. Inheritance helps in reducing the overall code size of the program, which is an important concept in object-oriented programming.

Data Abstraction:

Data Abstraction increases the power of programming language by creating user defined data types. Data Abstraction also represents the needed information in the program without presenting the details.

Data Encapsulation:

Data Encapsulation combines data and functions into a single unit called Class. When using Data Encapsulation, data is not accessed directly; it is only accessible through the functions present inside the class. Data Encapsulation enables the important concept of data hiding possible.

Polymorphism:

Polymorphism allows routines to use variables of different types at different times. An operator or function can be given different meanings or functions. Polymorphism refers to a single function or multi-functioning operator performing in different ways.

Overloading:

Overloading is one type of Polymorphism. It allows an object to have different meanings, depending on its context. When an existing operator or function begins to operate on new data type, or class, it is understood to be overloaded.

Reusability:

This term refers to the ability for multiple programmers to use the same written and debugged existing class of data. This is a time saving device and adds code efficiency to the language. Additionally, the programmer can incorporate new features to the existing class, further developing the application and allowing users to achieve increased performance. This time saving feature optimizes code, helps in gaining secured applications and facilitates easier maintenance on the application.

The implementation of each of the above object-oriented programming features for C++ will be highlighted in later sections.

```
#include < iostream.h >              // Preprocessor directive
class employee                  // Class Declaration
{
private:
        char empname[50];
        int empno;

  public:
        void getvalue()
        {
            cout<<"INPUT Employee Name:";
            cin>>empname;                        // waiting input from the Keyboard for the name
            cout<<"INPUT Employee Number:";
            cin>>empno;                            // waiting input from the Keyboard for the number
        }
        void displayvalue(){
        cout<<"Employee Name:"<< empname << endl; // displays the employee name
        cout<<"Employee Number:"<< empno << endl; // displays the emloyee number

        }
};
void main()
{
        employee e1;                // Creation of Object
        e1.getvalue();    // the getvalue method is being called
        e1.displayvalue(); // the displayvalue method is being called
}
```

Output:

```
Enter Employee Name: Jigar
Enter Employee Number: 35

Employee Name: Jigar
Employee Number:35

Press any key to continue…..
```

Comments

Comments are parts of the source code disregarded by the compiler. They simply do nothing. Their purpose is only to allow the programmer to insert notes or descriptions embedded within the source code.

C++ supports two ways to insert comments:

```
// line comment   single line

/* block comment */    multi line
```

Declaration of Variable

In order to use a variable in C++, we must first declare it specifying which data type we want it to be. The syntax to declare a new variable is to write the specifier of the desired data type (like int, bool, float...) followed by a valid variable identifier.

For example:

```
int a;
float mynumber;
```

These are two valid declarations of variables. The first one declares a variable of type int with the identifier a. The second one declares a variable of type float with the identifier mynumber. Once declared, the variables a and
mynumber can be used within the rest of their scope in the program.

If you are going to declare more than one variable of the same type, you can declare all of them in a single statement by separating their identifiers with commas.

For example:

```
int a, b, c;
```

```
// operating with variables
#include <iostream.h>
int main ()
{
// declaring variables:
int a, b;
int result;
// process:
a = 5;
b = 2;
a = a + 1;
result = a - b;
// print out the result:
cout << result;
// terminate the program:
return 0;
}
```

Output 4

Data Type

Every variable in C++ can store a value. However, the type of value which the variable can store has to be specified by the programmer.

C++ supports the following inbuilt data types:- int (to store integer values), float (to store decimal values) and char (to store characters), bool (to store Boolean value either 0 or 1) and void (signifies absence of information).

Integer data type

Integer (int) variables are used to store integer values like 34, 68704 etc. To declare a variable of type integer, type keyword int followed by the name of the variable. You can give any name to a variable but there are certain constraints, they are specified in Identifiers section.

For example, the statement

int sum;

Declares that sum is a variable of type int. You can assign a value to it now or later. In order to assign values along with declaration use the assignment operator (=).

int sum = 25;

assigns value 25 to variable sum.

There are three types of integer variables in C++, short, int and long int. All of them store values of type integer but the size of values they can store increases from short to long int. This is because of the size occupied by them in memory of the computer. The size which they can take is dependent on type of computer and varies. More is the size, more the value they can hold. For example, int variable has 2 or 4 bytes reserved in memory so it can hold 2^{32}= 65536 values. Variables can be signed or unsigned depending they store only positive values or both positive and negative values. And short, variable has 2 bytes. Long int variable has 4 bytes.

Float Data Type

To store decimal values, you use float data type. Floating point data types comes in three sizes, namely float, double and long double. The difference is in the length of value and amount of precision which they can take and it increases from float to long double.For example, statement

float average = 2.34;

declares a variable average which is of type float and has the initial value 2.34

Character Data Type

A variable of type char stores one character. It size of a variable of type char is typically 1 byte. The statement

char name = 'c';

declares a variable of type char (can hold characters) and has the initial values as character c. Note that value has to be under single quotes.

The C-style casting takes the synatax as….

(type) expression
C++-style casting is as below namely:

type (expression)
Let us see the concept of **type casting in C++** with a small example:

```
#include<conio.h>
#include <iostream.h>
void main()
{
        int a;
        float b,c;

        cout<< "Enter the value of a:";
        cin>>a;

        cout<< "\n Enter the value of b:";
        cin>>b;

        c = float(a)+b;

        cout<<"\n The value of c is:"<<c;
}
```

The output of the above program is…

```
Enter the value of a: 10
Enter the value of b: 12.5
The value of c is: 22.5
```

Key words

Keywords are the reserved words in any language which have a special pre defined meaning and cannot be used for any other purpose. You cannot use keywords for naming variables or some other purpose. We saw the use of keywords main, include, return, int in our first C++ program.

Identifiers

Identifiers are the name of functions, variables, classes, arrays etc. which you create while writing your programs. Identifiers are just like names in any language. There are certain rules to name identifiers in C++.

They are:-

- Identifiers must contain combinations of digits, characters and underscore (_).
- Identifier names cannot start with digit.
- Keyword cannot be used as an identifier name and upper case and lower case are distinct.
- Identifier names can contain any combination of characters as opposed to the restriction of 31 letters in C.

Constants

Constants are fixed values which cannot change. For example 123, 12.23, 'a' are constants. Now it's time to move on to our next tutorial Input values using cin operator.

Variable Scope in Functions:

The scope of the variables can be broadly be classified as

- Local Variables
- Global Variables

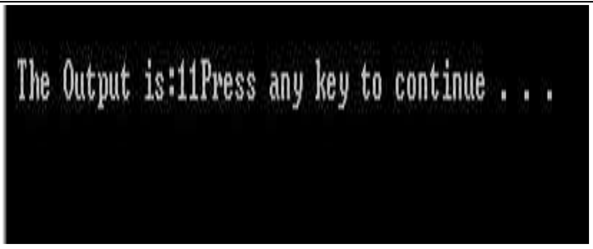
Local Variables:

The variables defined local to the block of the function would be accessible only within the block of the function and not outside the function. Such variables are called local variables. That is in other words the scope of the local variables is limited to the function in which these variables are declared.

Let us see this with a small example:

```
#include <iostream.h>
int exforsys(int,int);
void main( )
{
       int b;
       int s=5,u=6;
       b=exforsys(s,u);
       cout << "\n The Output is:" << b;
}

int exforsys(int x,int y)
{
       int z;
       z=x+y;
       return(z);
}
```

The output of the above example is:



```
The Output is:11Press any key to continue . . .
```

In the above program the variables x, y, z are accessible only inside the function exforsys( ) and their scope is limited only to the function exforsys( ) and not outside the function. Thus the variables x, y, z are local to the function exforsys. Similarly one would not be able to access variable b inside the function exforsys as such. This is because variable b is local to function main.

Global Variables:

Global variables are one which are visible in any part of the program code and can be used within all functions and outside all functions used in the program. The method of declaring global variables is to declare the variable outside the function or block.

For instance:

```
Int x,y,z;
void main()
{

}
```

In the above the integer variables x, y and z and the float variables a, b and c which are declared outside the block are global variables and the integer variables s and u and the float variables w and q which are declared inside the function block are called local variables.

Thus the scope of global variables is between the point of declaration and the end of compilation unit whereas scope of local variables is between the point of declaration and the end of innermost enclosing compound statement.

Let us see an example which has number of local and global variable declarations with number of inner blocks to understand the concept of local and global variables scope in detail.

```
int g;
void main( )
{
        ...
        int a;
        {
                int b;
                b=25;
               a=45;
                g=65;   // end of scope for variable b
        }
        a=50;
        exforsys( );
       ...
}// end of scope for variable a

void exforsys( )
{
        g = 30; //Scope of g is throughout the program and so is used between function calls
}
```

In the context of scope of variables in functions comes the important concept named as storage class which is discussed in detail in next section.

Storage Class

What is storage Class?

Storage class defined for a variable determines the accessibility and longevity of the variable. The accessibility of the variable relates to the portion of the program that has access to the variable. The longevity of the variable refers to the length of time the variable exists within the program.

Automatic Storage Class

Variables defined within the function body are called automatic variables. Auto is the keyword used to declare automatic variables. By default and without the use of a Keyword, the variables defined inside a function are automatic variables.

External Storage Class

External variables are also called global variables. External variables are defined outside any function, memory is set aside once it has been declared and remains until the end of the program. These variables are accessible by any function. This is mainly utilized when a programmer wants to make use of a variable and access the variable among different function calls.

Static Storage Class

The static automatic variables, as with local variables, are accessible only within the function in which it is defined. Static automatic variables exist until the program ends in the same manner as external variables. In order to maintain value between function calls, the static variable takes its presence.

C++ Character Functions

The C++ char functions are extremely useful for testing and transforming characters. These functions are widely used and accepted. In order to use character functions header file <cctype> is included into the program. Some of the commonly used character functions are:
isalnum()- The function isalnum() returns nonzero value if its argument is either an alphabet or integer. If the character is not an integer or alphabet then it returns zero.
isalpha() - The function isalpha() returns nonzero if the character is an uppercase or lower case letter otherwise it returns zero.
iscntrl() - The function iscntrl() returns nonzero if the character is a control character otherwise it returns zero.
isdigit()- The function isdigit() returns nonzero if the character is a digit, i.e. through 0 to 9. It returns zero for non digit character.
isgraph()- The function isgraph() returns nonzero if the character is any printable character other than space otherwise it returns zero.
islower()- The function islower() returns nonzero for a lowercase letter otherwise it returns zero.

isprint()- The function isprint() returns nonzero for printable character including space otherwise it returns zero.
isspace()- The function isspace() returns nonzero for space, horizontal tab, newline character, vertical tab, formfeed, carriage return; otherwise it returns zero.
ispunct()- The function ispunct() returns nonzero for punctuation characters otherwise it returns zero. The punctuation character excludes alphabets, digits and space.

isupper()- The function isupper() returns nonzero for an uppercase letter otherwise it returns zero.

tolower()- The function tolower() changes the upper case letter to its equivalent lower case letter. The character other than upper case letter remains unchanged.

toupper()- The function toupper() changes the lower case letter to its equivalent upper case letter. The character other than lower case letter remains unchanged.

isxdigit()- The function isxdigit() returns nonzero for hexadecimal digit i.e. digit from 0 to 9, alphabet 'a' to 'f' or 'A' to 'F' otherwise it returns zero.

Function

Using functions we can structure our programs in a more modular way, accessing all the potential that structured programming can offer to us in C++.

A function is a group of statements that is executed when it is called from some point of the program. The following is its format:

ret_type name (parameter1, parameter2, ...) { statements }
where:

- type is the data type specifier of the data returned by the function.
- name is the identifier by which it will be possible to call the function.
- parameters (as many as needed): Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: int x) and which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas.
- statements is the function's body. It is a block of statements surrounded by braces { }.

Here you have the first function example:

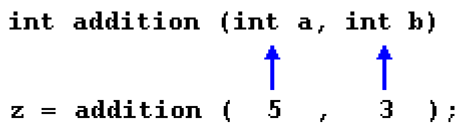
```
#include <iostream.h>
int addition (int a, int b)
{
  int r;
  r=a+b;
  return (r);
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
  return 0;
}
```

Output:

```
The result is 8
```

In order to examine this code, first of all remember something said at the beginning of this tutorial: a C++ program always begins its execution by the main function. So we will begin there.

We can see how the main function begins by declaring the variable z of type int. Right after that, we see a call to a function called addition. Paying attention we will be able to see the similarity between the structure of the call to the function and the declaration of the function itself some code lines above:



```
int addition (int a, int b)

z = addition ( 5  ,  3  );
```

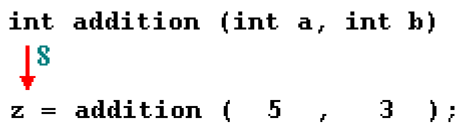
The parameters and arguments have a clear correspondence. Within the main function we called to addition passing two values: 5 and 3, that correspond to the int a and int b parameters declared for function addition.

At the point at which the function is called from within main, the control is lost by main and passed to function addition. The value of both arguments passed in the call (5 and 3) are copied to the local variables int a and int b within the function.
Function addition declares another local variable (int r), and by means of the expression r=a+b, it assigns to r the result of a plus b. Because the actual parameters passed for a and b are 5 and 3 respectively, the result is 8.

The following line of code:

return (r);

finalizes function addition, and returns the control back to the function that called it in the first place (in this case, main). At this moment the program follows its regular course from the same point at which it was interrupted by the call to addition. But additionally, because the return statement in function addition specified a value: the content of variable r (return (r);), which at that moment had a value of 8. This value becomes the value of evaluating the function call.



```
int addition (int a, int b)
  8
z = addition (  5  ,   3  );
```

So being the value returned by a function the value given to the function call itself when it is evaluated, the variable z will be set to the value returned by addition (5, 3), that is 8. To explain it another way, you can imagine that the call to a function (addition (5,3)) is literally replaced by the value it returns (8).
The following line of code in main is:

```
cout << "The result is " << z;
```

That, as you may already expect, produces the printing of the result on the screen.

Function With No type use of Void:

If you remember the syntax of a function declaration:
type name (argument1, argument2 ...) statement
you will see that the declaration begins with a type, that is the type of the function itself (i.e., the type of the datum that will be returned by the function with the return statement). But what if we want to return no value?

Imagine that we want to make a function just to show a message on the screen. We do not need it to return any value. In this case we should use the void type specifier for the function. This is a special specifier that indicates absence of type.

```
#include <iostream.h>
void printmessage ()
{
  cout << "I'm a function!";
}

int main ()
{
  printmessage ();
  return 0;
}
```

Output:

```
I'm a function!
```

void can also be used in the function's parameter list to explicitly specify that we want the function to take no actual parameters when it is called. For example, function printmessage could have been declared as:

```
void printmessage (void)
{
  cout << "I'm a function!";
}
```

Although it is optional to specify void in the parameter list. In C++, a parameter list can simply be left blank if we want a function with no parameters.

What you must always remember is that the format for calling a function includes specifying its name and enclosing its parameters between parentheses. The non-existence of parameters does not exempt us from the obligation to write the parentheses. For that reason the call to printmessage is:

```
printmessage ();
```

The parentheses clearly indicate that this is a call to a function and not the name of a variable or some other C++ statement. The following call would have been incorrect:

```
printmessage;
```

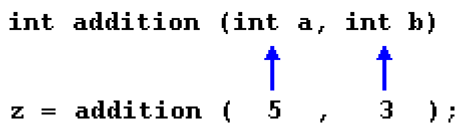
Argument Passed By Value and Passed By Reference:

Until now, in all the functions we have seen, the arguments passed to the functions have been passed *by value*. This means that when calling a function with parameters, what we have passed to the function were copies of their values but never the variables themselves. For example, suppose that we called our first function addition using the following code:

```
int x=5, y=3, z;
```

```
z = addition ( x , y );
```

What we did in this case was to call to function addition passing the values of x and y, i.e. 5 and 3 respectively, but not the variables x and y themselves.



```
int addition (int a, int b)

z = addition (  5  ,  3  );
```

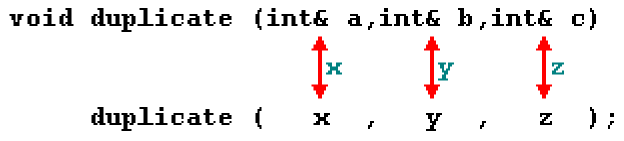
This way, when the function addition is called, the value of its local variables a and b become 5 and 3 respectively, but any modification to either a or b within the function addition will not have any effect in the values of x and y outside it, because variables x and y were not themselves passed to the function, but only copies of their values at the moment the function was called.
But there might be some cases where you need to manipulate from inside a function the value of an external variable. For that purpose we can use arguments passed by reference, as in the function duplicate of the following example:

```
#include <iostream.h>
void duplicate (int *a, int *b, int *c)
{
  a*=2;
  b*=2;
  c*=2;
}
int main ()
{
  int x=1, y=3, z=7;
  duplicate (&x, &y, &z);
  cout << "x=" << x << ", y=" << y << ", z=" << z;
  return 0;
}
```

Output:

```
x=2, y=6, z=14
```

The first thing that should call your attention is that in the declaration of duplicate the type of each parameter was followed by an ampersand sign (&). This ampersand is what specifies that their corresponding arguments are to be passed *by reference* instead of *by value*.
When a variable is passed by reference we are not passing a copy of its value, but we are somehow passing the variable itself to the function and any modification that we do to the local variables will have an effect in their counterpart variables passed as arguments in the call to the function.



```
void duplicate (int& a,int& b,int& c)

      duplicate (  x  ,  y  ,  z  );
```

To explain it in another way, we associate a, b and c with the arguments passed on the function call (x, y and z) and any change that we do on a within the function will affect the value of x outside it. Any change that we do on b will affect y, and the same with c and z.

That is why our program's output, that shows the values stored in x, y and z after the call to duplicate, shows the values of all the three variables of main doubled.
If when declaring the following function:

void duplicate (*int*& a, *int*& b, *int*& c)

we had declared it this way:

void duplicate (*int* a, *int* b, *int* c)

i.e., without the ampersand signs (&), we would have not passed the variables by reference, but a copy of their values instead, and therefore, the output on screen of our program would have been the values of x, y and z without having been modified.
Passing by reference is also an effective way to allow a function to return more than one value. For example, here is a function that returns the previous and next numbers of the first parameter passed.

```
#include <iostream.h>
void prevnext (int x, int& prev, int& next)
{
  prev = x-1;
  next = x+1;
}
int main ()
{
  int x=100, y, z;
  prevnext (x, y, z);
  cout << "Previous=" << y << ", Next=" << z;
  return 0;
}
```

Output:
```
Previous=99, Next=101
```

Default Values in Parameters:

When declaring a function we can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead. For example:

```
#include <iostream.h>
int divide (int a, int b=2)
{
  int r;
  r=a/b;
  return (r);
}
int main ()
{
  cout << divide (12);
  cout << endl;
  cout << divide (20,4);
  return 0;
}
```

Output:

```
6
5
```

As we can see in the body of the program there are two calls to function divide. In the first one:

divide (12)

we have only specified one argument, but the function divide allows up to two. So the function divide has assumed that the second parameter is 2 since that is what we have specified to happen if this parameter was not passed (notice the function declaration, which finishes with int b=2, not just int b). Therefore the result of this function call is 6 (12/2).

In the second call:

divide (20,4)

there are two parameters, so the default value for b (int b=2) is ignored and b takes the value passed as argument, that is 4, making the result returned equal to 5 (20/4).

Overloaded Functions:

In C++ two different functions can have the same name if their parameter types or number are different. That means that you can give the same name to more than one function if they have either a different number of parameters or different types in their parameters. For example:

```
#include <iostream.h>
int operate (int a, int b)
{
  return (a*b);
}
float operate (float a, float b)
{
  return (a/b);
}
int main ()
{
  int x=5,y=2;
  float n=5.0,m=2.0;
  cout << operate (x,y);
  cout << "\n";
  cout << operate (n,m);
  cout << "\n";
  return 0;
}
```

Output:

```
10
2.5
```

In this case we have defined two functions with the same name, operate, but one of them accepts two parameters of type int and the other one accepts them of type float. 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 ints as its arguments it calls to the function that has two int parameters in its prototype and if it is called with two floats it will call to the one which has two float parameters in its prototype.

In the first call to operate the two arguments passed are of type int, therefore, the function with the first prototype is called; This function returns the result of multiplying both parameters. While the second call passes two arguments of type float, so the function with the second prototype is called. This one has a different behavior: it divides one parameter by the other. So the behavior of a call to operate depends on the type of the arguments passed because the function has been *overloaded*.

Notice that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

Inline functions.

The inline specifier indicates the compiler that inline substitution is preferred to the usual function call mechanism for a specific function. This does not change the behavior of a function itself, but is used to suggest to the compiler that the code generated by the function body is inserted at each point the function is called, instead of being inserted only once and perform a regular call to it, which generally involves some additional overhead in running time.

The format for its declaration is:

inline type name (arguments ...) { instructions ... }

and the call is just like the call to any other function. You do not have to include the inline keyword when calling the function, only in its declaration.

Most compilers already optimize code to generate inline functions when it is more convenient. This specifier only indicates the compiler that inline is preferred for this function.

Example

```
#include <iostream.h>
inline int add(int a,int b)

{

        return a+b;

}
void main( )
{
int x,y;
        cout << “\n Enter the Input Value: ”;
        cin>>x>>y;
        cout<<”\n The Output is: “ << add(x,y);
}
```

```
Output:
Enter the Input Value: 10 20
The Output is: 30
```

Recursive Function

Recursivity is the property that functions have to be called by themselves. It is useful for many tasks, like sorting or calculate the factorial of numbers. For example, to obtain the factorial of a number (n!) the mathematical formula would be:

n! = n * (n-1) * (n-2) * (n-3) ... * 1

more concretely, 5! (factorial of 5) would be:

5! = 5 * 4 * 3 * 2 * 1 = 120

and a recursive function to calculate this in C++ could be:

```
#include <iostream.h>
long factorial (long a)
{
  if (a > 1)
   return (a * factorial (a-1));
  else
```

```
    return (1);
}
int main ()
{
   long number;
   cout << "Please type a number: ";
   cin >> number;
   cout << number << "=" << factorial (number);
   return 0;
}
```

Output:

```
Please type a number: 9
9 = 362880
```

Notice how in function factorial we included a call to itself, but only if the argument passed was greater than 1, since otherwise the function would perform an infinite recursive loop in which once it arrived to 0 it would continue multiplying by all the negative numbers (probably provoking a stack overflow error on runtime).

This function has a limitation because of the data type we used in its design (long) for more simplicity. The results given will not be valid for values much greater than 10! or 15!, depending on the system you compile it.

Declaring functions.

Until now, we have defined all of the functions before the first appearance of calls to them in the source code. These calls were generally in function main which we have always left at the end of the source code. If you try to repeat some of the examples of functions described so far, but placing the function main before any of the other functions that were called from within it, you will most likely obtain compiling errors. The reason is that to be able to call a function it must have been declared in some earlier point of the code, like we have done in all our examples.

But there is an alternative way to avoid writing the whole code of a function before it can be used in main or in some other function. This can be achieved by declaring just a prototype of the function before it is used, instead of the entire definition. This declaration is shorter than the entire definition, but significant enough for the compiler to determine its return type and the types of its parameters.

Its form is:

type name (argument_type1, argument_type2, ...);

It is identical to a function definition, except that it does not include the body of the function itself (i.e., the function statements that in normal definitions are enclosed in braces { }) and instead of that we end the prototype declaration with a mandatory semicolon (;).

The parameter enumeration does not need to include the identifiers, but only the type specifiers. The inclusion of a name for each parameter as in the function definition is optional in the prototype declaration. For example, we can declare a function called protofunction with two int parameters with any of the following declarations:

1 *int* protofunction (*int* first, *int* second);

2 *int* protofunction (*int*, *int*);

Anyway, including a name for each variable makes the prototype more legible.

```
#include <iostream>
using namespace std;

void odd (int a);
void even (int a);

int main ()
{
  int i;
  do {
    cout << "Type a number (0 to exit): ";
    cin >> i;
    odd (i);
  } while (i!=0);
  return 0;
}

void odd (int a)
{
  if ((a%2)!=0) cout << "Number is odd.\n";
  else even (a);
}

void even (int a)
{
  if ((a%2)==0) cout << "Number is even.\n";
  else odd (a);}
```

Output:

```
Type a number (0 to exit): 9
Number is odd.
Type a number (0 to exit): 6
```

```
Number is even.
Type a number (0 to exit): 1030
Number is even.
Type a number (0 to exit): 0
Number is even.
```

This example is indeed not an example of efficiency. I am sure that at this point you can already make a program with the same result, but using only half of the code lines that have been used in this example. Anyway this example illustrates how prototyping works. Moreover, in this concrete example the prototyping of at least one of the two functions is necessary in order to compile the code without errors.

The first things that we see are the declaration of functions odd and even:

```
void odd (int a);
void even (int a);
```

This allows these functions to be used before they are defined, for example, in main, which now is located where some people find it to be a more logical place for the start of a program: the beginning of the source code.

Anyway, the reason why this program needs at least one of the functions to be declared before it is defined is because in odd there is a call to even and in even there is a call to odd. If none of the two functions had been previously declared, a compilation error would happen, since either odd would not be visible from even (because it has still not been declared), or even would not be visible from odd (for the same reason).

Having the prototype of all functions together in the same place within the source code is found practical by some programmers, and this can be easily achieved by declaring all functions prototypes at the beginning of a program.

Variable Scope in Functions:

The scope of the variables can be broadly be classified as

- Local Variables
- Global Variables

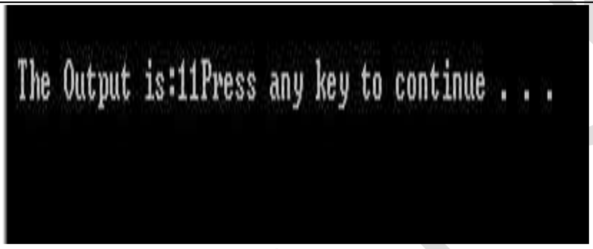
Local Variables:

The variables defined local to the block of the function would be accessible only within the block of the function and not outside the function. Such variables are called local variables. That is in other words the scope of the local variables is limited to the function in which these variables are declared.

Let us see this with a small example:

```
#include <iostream.h>
int exforsys(int,int);
void main( )
{
      int b;
      int s=5,u=6;
      b=exforsys(s,u);
      cout << "\n The Output is:" << b;
}

int exforsys(int x,int y)
{
      int z;
      z=x+y;
      return(z);
}
```

The output of the above example is:



```
The Output is:11Press any key to continue . . .
```

In the above program the variables x, y, z are accessible only inside the function exforsys( ) and their scope is limited only to the function exforsys( ) and not outside the function. Thus the variables x, y, z are local to the function exforsys. Similarly one would not be able to access variable b inside the function exforsys as such. This is because variable b is local to function main.

Global Variables:

Global variables are one which are visible in any part of the program code and can be used within all functions and outside all functions used in the program. The method of declaring global variables is to declare the variable outside the function or block.

For instance:

```
Int x,y,z;
void main()
{
}
```

In the above the integer variables x, y and z and the float variables a, b and c which are declared outside the block are global variables and the integer variables s and u and the float variables w and q which are declared inside the function block are called local variables.

Thus the scope of global variables is between the point of declaration and the end of compilation unit whereas scope of local variables is between the point of declaration and the end of innermost enclosing compound statement.

Let us see an example which has number of local and global variable declarations with number of inner blocks to understand the concept of local and global variables scope in detail.

```
int g;
void main( )
{
        ...
        int a;
        {
               int b;
               b=25;
              a=45;
               g=65;   // end of scope for variable b
        }
        a=50;
        exforsys( );
       ...
}// end of scope for variable a

void exforsys( )
{
        g = 30; //Scope of g is throughout the program and so is used between function calls
}
```

In the context of scope of variables in functions comes the important concept named as storage class which is discussed in detail in next section.

Inline Function:

What is Inline Function?

Inline functions are functions where the call is made to inline functions. The actual code then gets placed in the calling program.

Reason for the need of Inline Function:

Normally, a function call transfers the control from the calling program to the function and after the execution of the program returns the control back to the calling program after the function call. These concepts of function save program space and memory space and are used because the function is stored only in one place and is only executed when it is called. This execution may be time consuming since the registers and other processes must be saved before the function gets called.

The extra time needed and the process of saving is valid for larger functions. If the function is short, the programmer may wish to place the code of the function in the calling program in order for it to be executed. This type of function is best handled by the inline function. In this situation, the programmer may be wondering "why not write the short code repeatedly inside the program wherever needed instead of going for inline function?". Although this could accomplish the task, the problem lies in the loss of clarity of the program. If the programmer repeats the same code many times, there will be a loss of clarity in the program. The alternative approach is to allow inline functions to achieve the same purpose, with the concept of functions.

What happens when an inline function is written?

The inline function takes the format as a normal function but when it is compiled it is compiled as inline code. The function is placed separately as inline function, thus adding readability to the source program. When the program is compiled, the code present in function body is replaced in the place of function call.

General Format of inline Function:

The general format of inline function is as follows:

inline datatype function_name(arguments)

The keyword inline specified in the above example, designates the function as inline function. For example, if a programmer wishes to have a function named exforsys with return value as integer and with no arguments as inline it is written as follows:

inline int exforsys( )

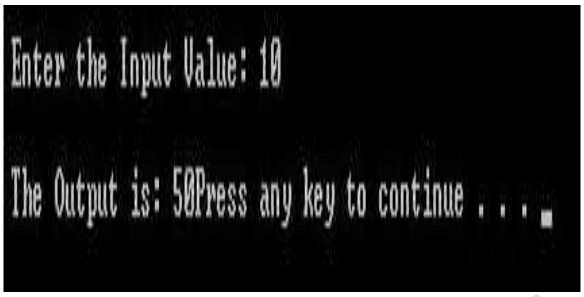
Example:

The concept of inline functions:

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

```
        cout << "n Enter the Input Value: ";
        cin>>x;
        cout << "n The Output is: " << exforsys(x);
}

inline int exforsys(int x1)
{
        return 5*x1;
}
```

The output of the above program is:



```
Enter the Input Value: 10

The Output is: 50Press any key to continue . . .
```

The output would be the same even when the inline function is written solely as a function. The concept, however, is different. When the program is compiled, the code present in the inline function exforsys( ) is replaced in the place of function call in the calling program. The concept of inline function is used in this example because the function is a small line of code.

The above example, when compiled, would have the structure as follows:

```
#include <iostream.h>
int exforsys(int);
void main( )
{
        int x;
        cout << "n Enter the Input Value: ";
        cin>>x;
        cout << "n The Output is: " << exforsys(x);
}

inline int exforsys(int x1)
{
        return 5*x1;
}
```

When the above program is written as normal function the compiled code would look like below:

```
#include <iostream.h>

int exforsys(int);
void main( )
{
        int x;
        cout << "n Enter the Input Value: ";
        cin>>x;
        cout << "n The Output is: " << exforsys(x); //Call is made to the function exforsys
}

inline int exforsys(int x1)
{
        return 5*x1;
}
```

A programmer must make wise choices when to use inline functions. Inline functions will save time and are useful if the function is very small. If the function is large, use of inline functions must be avoided.

Friend Functions

The Need for Friend Function:

As discussed in the earlier sections on access specifiers, when a data is declared as private inside a class, then it is not accessible from outside the class. A function that is not a member or an external class will not be able to access the private data. A programmer may have a situation where he or she would need to access private data from non-member functions and external classes. For handling such cases, the concept of Friend functions is a useful tool.

What is a Friend Function?

A friend function is used for accessing the non-public members of a class. A class can allow non-member functions and other classes to access its own private data, by making them friends. Thus, a friend function is an ordinary function or a member of another class.

How to define and use Friend Function in C++:

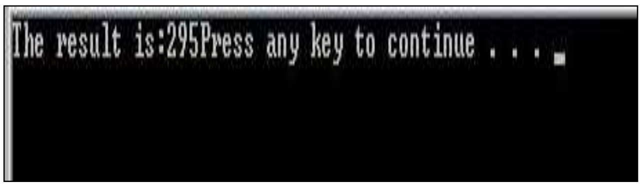
The friend function is written as any other normal function, except the function declaration of these functions is preceded with the keyword friend. The friend function must have the class to which it is declared as friend passed to it in argument.

Some important points to note while using friend functions in C++:

- The keyword friend is placed only in the function declaration of the friend function and not in the function definition.
- It is possible to declare a function as friend in any number of classes.
- When a class is declared as a friend, the friend class has access to the private data of the class that made this a friend.
- A friend function, even though it is not a member function, would have the rights to access the private members of the class.
- It is possible to declare the friend function as either private or public.
- The function can be invoked without the use of an object. The friend function has its argument as objects, seen in example below.

```
 #include <iostream.h>

 class exforsys
 {
 private:
       int a,b;
 public:
       void test()
       {
             a=100;
             b=200;
       }
      friend int compute(exforsys e1);
//Friend Function Declaration with keyword friend and with the //object of class exforsys to which it
is friend passed to it
 };

 int compute(exforsys e1)
 {
 //Friend Function Definition which has access to private data
       return int(e1.a+e1.b)-5;
 }

 void main()
 {
       exforsys e;
       e.test();
       cout << "The result is:" << compute(e);
//Calling of Friend Function with object as argument.
}
```

The output of the above program is



```
The result is:295Press any key to continue . . .
```

The function compute() is a non-member function of the class exforsys. In order to make this function have access to the private data a and b of class exforsys , it is created as a friend function for the class exforsys. As a first step, the function compute() is declared as friend in the class exforsys as:

friend int compute (exforsys e1)

The keyword friend is placed before the function. The function definition is written as a normal function and thus, the function has access to the private data a and b of the class exforsys. It is declared as friend inside the class, the private data values a and b are added, 5 is subtracted from the result, giving 295 as the result. This is returned by the function and thus the output is displayed as shown above.

Scope Resolution Operator

Member functions can be defined within the class definition or separately using **scope resolution operator, ::**. Defining a member function within the class definition declares the function **inline**, even if you do not use the inline specifier. So either you can define **Volume()** function as below:

```
class Box
{
  public:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box

    double getVolume(void)
    {
      return length * breadth * height;
    }
};
```

Arrays

What is an array?

An array is a group of elements of the same type that are placed in contiguous memory locations.
That means that, for example, we can store 5 values of type `int` in an array without having to declare 5 different variables, each one with a different identifier. Instead of that, using an array we can store 5 different values of the same type, `int` for example, with a unique identifier.

Initializing arrays

When declaring a regular array of local scope (within a function, for example), if we do not specify otherwise, its elements will not be initialized to any value by default, so their content will be undetermined until we store some value in them. The elements of global and static arrays, on the other hand, are automatically initialized with their default values, which for all fundamental types this means they are filled with zeros.

How to access an array element?

You can access an element of an array by adding an index to a unique identifier.

```
 #include <iostream>
int billy [] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ()
{
      for ( n=0 ; n<5 ; n++ )

     {
           result += billy[n];
     }
     cout << result;
     return 0;

}
```

Output:

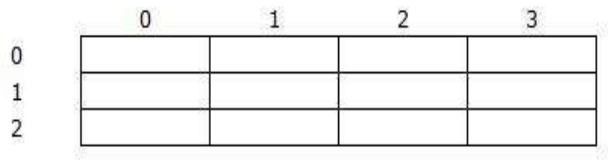
```
12206
Press any key to continue
```

What is a Multidimensional Array?

int Exforsys[3][4];

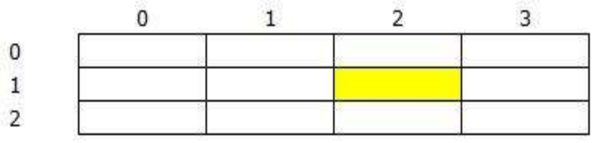
It is represented internally as:

Exforsys Data Type: int



How to access the elements in the Multidimensional Array

Exforsys Data Type: int



Based on the above two-dimensional arrays, it is possible to handle multidimensional arrays of any number of rows and columns in C++ programming language. This is all occupied in memory. Better utilization of memory must also be made.

Multidimensional Array Example:

```
#include <iostream.h>
const int ROW=4;
const int COLUMN =3;
void main()
{
    int i,j;
    int Exforsys[ROW][COLUMN];
    for(i=0;i < ROW;i++)  //goes through the ROW elements
        for(j=0;j < COLUMN;j++)   //goes through the COLUMN elements
        {
            cout << "Enter value of Row " << i+1;
            cout << ",Column " << j+1 << ":";
            cin>>Exforsys[i][j];
        }
        cout << "\n\n\n";
        cout << " COLUMN\n";
        cout << " 1 2 3";
        for(i=0;i < ROW;i++)
        {
            cout << "\nROW " << i+1;
            for(j=0;j < COLUMN;j++)
                cout << Exforsys[i][j];
```

```
          }
 }
```

Static in C++ Class

We can define class members static using **static** keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator **::** to identify which class it belongs to.

```
class Box
{
  public:
    static int objectCount;
    // Constructor definition
    Box(double l=2.0, double b=2.0, double h=2.0)
    {
      cout <<"Constructor called." << endl;
      length = l;
      breadth = b;
      height = h;
      // Increase every time object is created
      objectCount++;
    }
    double Volume()
    {
      return length * breadth * height;
    }
  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};
// Initialize static member of class Box
int Box::objectCount = 0;
int main(void)
{
  Box Box1(3.3, 1.2, 1.5);    // Declare box1
```

```
   Box Box2(8.5, 6.0, 2.0);    // Declare box2
   // Print total number of objects.
   cout << "Total objects: " << Box::objectCount << endl;
}
```

Output:

```
Constructor called.
Constructor called.
Total objects: 2
```

Static Function Members

By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the static functions are accessed using only the class name and the scope resolution operator ::.

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to the this pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

```
class Box
{
   public:
     static int objectCount;
     // Constructor definition
     Box(double l=2.0, double b=2.0, double h=2.0)
     {
        cout <<"Constructor called." << endl;
        length = l;
        breadth = b;
        height = h;
        objectCount++;
     }
     double Volume()
     {
        return length * breadth * height;
     }
     static int getCount()
     {
        return objectCount;
      }
   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};
// Initialize static member of class Box
int Box::objectCount = 0;
int main(void)
{

   // Print total number of objects before creating object.
   cout << "Inital Stage Count: " << Box::getCount() << endl;
   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2
   // Print total number of objects after creating object.
   cout << "Final Stage Count: " << Box::getCount() << endl;
   return 0;
}
```

Output:

```
Inital Stage Count: 0
Constructor called.
Constructor called.
Final Stage Count: 2
```

Class, Constructor and Destructor

Class Constructors and destructors in C++

In this C++ tutorial you will learn about Class Constructors and destructors in C++ viz., Constructors, What is the use of Constructor, General Syntax of Constructor, Destructors, What is the use of Destructors and General Syntax of Destructors.

Constructors:

What is the use of Constructor

The main use of constructors is to initialize objects. The function of initialization is automatically carried out by the use of a special member function called a constructor.

General Syntax of Constructor

A constructor is a special member function that takes the same name as the class name. The syntax generally is as given below:

{ arguments};

The default constructor for a class X has the form

X::X()

In the above example, the arguments are optional.

The constructor is automatically named when an object is created. A constructor is named whenever an object is defined or dynamically allocated using the "new" operator.

There are several forms in which a constructor can take its shape namely:

Default Constructor:

This constructor has no arguments in it. The default Constructor is also called as the no argument constructor.

```
class Exforsys
{
   private:
      int a,b;
   public:
      Exforsys();
      ...
};

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

Copy constructor:

This constructor takes one argument, also called one argument constructor. The main use of copy constructor is to initialize the objects while in creation, also used to copy an object. The copy constructor allows the programmer to create a new object from an existing one by initialization.

For example to invoke a copy constructor the programmer writes:

Exforsys e3(e2);

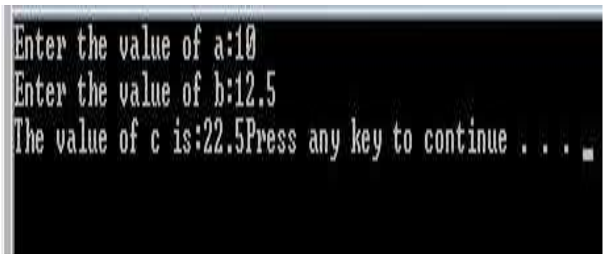
or

Exforsys e3=e2;

Both the above formats can be sued to invoke a copy constructor.

```
#include <iostream.h>
class Exforsys
{
   private:
      int a;
   public:
      Exforsys()
      { }
      Exforsys(int w)
   {
      a=w;
   }
   Exforsys(Exforsys e)
   {
      a=e.a;
      cout << " Example of Copy Constructor";
   }
   void result()
   {
      cout<< a;
   }
};
void main()
{
   Exforsys e1(50);
   Exforsys e3(e1);
   cout<< "ne3=";e3.result();
}
```

In the above the copy constructor takes one argument an object of type Exforsys which is passed by reference. The output of the above program is.



```
Enter the value of a:10
Enter the value of b:12.5
The value of c is:22.5Press any key to continue . . .
```

Some important points about constructors:

- A constructor takes the same name as the class name.
- The programmer cannot declare a constructor as virtual or static, nor can the programmer declare a constructor as const, volatile, or const volatile.
- No return type is specified for a constructor.
- The constructor must be defined in the public. The constructor must be a public member.
- Overloading of constructors is possible. This will be explained in later sections of this tutorial.

Parameterized Constructor

C++ permits us to achieve this objects bt passing argument to the constructor function when the object are created . The constructor that can take arguments are called parametrized constructors

```
class example

{

             public:

            int health;

            example(int h)

            {

                        health = h;

             }

 };

example character(99);

int main()

{
            cout <<character.health;
}
class Line
{
            public:
            int getLength( void );
            Line( int len ); // Parameterize constructor
            Line( const Line &obj); // copy constructor
            ~Line(); // destructor
};
```

Dynamic Constructor

The constructor can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount for each object when the objects are not of the same size, thus resulting in the saving of memory.

Allocation of memory to objects at the time of their construction is known as dynamic constructor of objects. The memory is allocated with the help of the new operator.

```
#include <iostream.h>
#include <conio.h>
class Account
{
        private:
                int account_no;
                int balance;
        public :
                Account(int a,int b)
                {
account_no=a;
balance=b;
        }
        void display()
        {
cout<< "\nAccount number is : "<< account_no;
cout<< "\nBalance is : " << balance;
}
};
void main()
{
        clrscr();
        int an,bal;
        cout<< "Enter account no : ";
        cin >> an;
        cout<< "\nEnter balance : ";
        cin >> bal;
        Account *acc=new Account(an,bal); //dynamic constructor
        acc->display(); //'->' operator is used to access the method
        getch();
}
```

Virtaual Constructor

C++ being static typed (the purpose of RTTI is different) language, it is meaningless to the C++ compiler to create an object polymorphically. The compiler must be aware of the class type to create the object. In other words, what type of object to be created is a compile time decision from C++ compiler perspective. If we make constructor virtual, compiler flags an error. In fact except inline, no other keyword is allowed in the declaration of constructor.

In practical scenarios we would need to create a derived class object in a class hierarchy based on some input. Putting in other words, object creation and object type are tightly coupled which forces modifications to extended. The objective of virtual constructor is to decouple object creation from it’s type.

```
#include <iostream.h>
#include<conio.h>

class Base
{
public:

   Base() { }

   virtual // Ensures to invoke actual object destructor
   ~Base() { }

   // An interface
   virtual void DisplayAction() = 0;
};
class Derived1 : public Base
{
public:
   Derived1()
   {
          cout << "\nDerived1 created" << endl;
   }

   ~Derived1()
   {
      cout << "Derived1 destroyed" << endl;
   }

   void DisplayAction()
   {
      cout << "Action from Derived1" << endl;
   }
};
class Derived2 : public Base
{
public:
   Derived2()
   {
      cout << "Derived2 created" << endl;
   }

   ~Derived2()
```

```
  {
     cout << "Derived2 destroyed" << endl;
  }

  void DisplayAction()
  {
     cout << "Action from Derived2" << endl;
  }
};
class User
{
public:
   // Creates Drived1
  User() : pBase(0)
  {
     // What if Derived2 is required? - Add an if-else ladder (see next sample)
     pBase = new Derived1();
  }
   ~User()
  {
     if( pBase )
     {
        delete pBase;
        pBase = 0;
     }
  }
   // Delegates to actual object
  void Action()
  {
     pBase->DisplayAction();
  }
 private:
  Base *pBase;
};

int main()
{
  clrscr();
  User *user = new User();

  // Need Derived1 functionality only
  user->Action();

  delete user;
  getch();
```

```
   return 0;
}
```

Out Put

```
Derived1 created
Action from Derived1
Derived1 destroyed
```

Destructors

What is the use of Destructors

Destructors are also special member functions used in C++ programming language. Destructors have the opposite function of a constructor. The main use of destructors is to release dynamic allocated memory. Destructors are used to free memory, release resources and to perform other clean up. Destructors are automatically named when an object is destroyed. Like constructors, destructors also take the same name as that of the class name.

General Syntax of Destructors

~ classname();

The above is the general syntax of a destructor. In the above, the symbol tilda ~ represents a destructor which precedes the name of the class.

Some important points about destructors:

- Destructors take the same name as the class name.
- Like the constructor, the destructor must also be defined in the public. The destructor must be a public member.
- The Destructor does not take any argument which means that destructors cannot be overloaded.
- No return type is specified for destructors.

```
class Exforsys
{
    private:
            ...
    public:
    Exforsys()
    { }
     ~ Exforsys()
    { }
}
```

Inheritance

What is Inheritance?

Inheritance is the process by which new classes called derived classes are created from existing classes called base classes. The derived classes have all the features of the base class and the programmer can choose to add new features specific to the newly created derived class.

For example, a programmer can create a base class named fruit and define derived classes as mango, orange, banana, etc. Each of these derived classes, (mango, orange, banana, etc.) has all the features of the base class (fruit) with additional attributes or features specific to these newly created derived classes. Mango would have its own defined features, orange would have its own defined features, banana would have its own defined features, etc.

This concept of Inheritance leads to the concept of polymorphism.

Types of inheritance

- ❑ Single Inheritance
- ❑ Multiple Inheritance
- ❑ Hierarchical Inheritance
- ❑ Multilevel Inheritance
- ❑ Hybrid Inheritance

Features or Advantages of Inheritance:

Reusability:

Inheritance helps the code to be reused in many situations. The base class is defined and once it is compiled, it need not be reworked. Using the concept of inheritance, the programmer can create as many derived classes from the base class as needed while adding specific features to each derived class as needed.

Saves Time and Effort:

The above concept of reusability achieved by inheritance saves the programmer time and effort. Since the main code written can be reused in various situations as needed.

Increases Program Structure which results in greater reliability.

Polymorphism (to be discussed in detail in later sections)

General Format for implementing the concept of Inheritance:

class derived_classname: access specifier baseclassname

For example, if the base class is exforsys and the derived class is sample it is specified as:

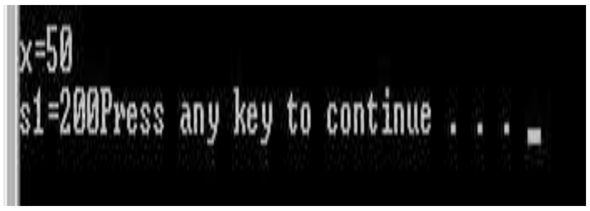
class sample: public exforsys

The above makes sample have access to both public and protected variables of base class exforsys. Reminder about public, private and protected access specifiers:

- If a member or variables defined in a class is private, then they are accessible by members of the same class only and cannot be accessed from outside the class.
- Public members and variables are accessible from outside the class.
- Protected access specifier is a stage between private and public. If a member functions or variables defined in a class are protected, then they cannot be accessed from outside the class but can be accessed from the derived class.

```
#include <iostream.h>
class exforsys
{
    public:
    exforsys(void) { x=0; }
    void f(int n1)
    {
        x= n1*5;
    }
    void output(void) { cout << "\n" << "x=" << x; }
    private:
    int x;
};

class sample: public exforsys
{
    public:
    sample(void) { s1=0; }
    void f1(int n1)
    {
        s1=n1*10;
    }
    void output(void)
    {
        exforsys::output();
        cout << "\n" << "s1=" << s1;
    }
    private:
    int s1;
};
int main(void)
{
    sample s;
    s.f(10);
    s.f1(20);
    s.output();
    return 0;
}
```

The output of the above program is



```
x=50
s1=200Press any key to continue . . .
```

In the above example, the derived class is sample and the base class is exforsys. The derived class defined above has access to all public and private variables. Derived classes cannot have access to base class constructors and destructors. The derived class would be able to add new member functions, or variables, or new constructors or new destructors. In the above example, the derived class sample has new member function f1( ) added in it. The line:

    sample s;

creates a derived class object named as s. When this is created, space is allocated for the data members inherited from the base class exforsys and space is additionally allocated for the data members defined in the derived class sample.

The base class constructor exforsys is used to initialize the base class data members and the derived class constructor sample is used to initialize the data members defined in derived class.

The access specifier specified in the line:

    class sample: public exforsys

Public indicates that the public data members which are inherited from the base class by the derived class sample remains public in the derived class.

Multiple Inheritance

```
#include<iostream.h>

#include<conio.h>



class student

{

   protected:

            int rno,m1,m2;

   public:

            void get()

            {

                     cout<<"Enter the Roll no :";

                     cin>>rno;

                     cout<<"Enter the two marks   :";

                     cin>>m1>>m2;

            }

};

class sports

{

   protected:

            int sm;           // sm = Sports mark

   public:

                     void getsm()

            {

                           cout<<"\nEnter the sports mark :";

                     cin>>sm;
            }
};


class statement:public student,public sports
{
    int tot,avg;
    public:
            void display()
            {
                        tot=(m1+m2+sm);
                        avg=tot/3;
                        cout<<"\n\n\tRoll No: "<<rno<<"\n\tTotal: "<<tot;
                cout<<"\n\tAverage    : "<<avg;
            }
};
void main()
{
   clrscr();
   statement obj;
   obj.get();
   obj.getsm();
   obj.display();
   getch();
}
```

Output:

```
            Enter the Roll no: 100
            Enter two marks
```

```
90

80

Enter the Sports Mark: 90

Roll No: 100

Total    : 260

Average: 86.66
```

Multilevel Inheritance

```
#include <iostream.h>
#include<conio.h>
class mm
 {
   protected:
     int rollno;
   public:
     void get_num(int a)
          { rollno = a; }
     void put_num()
          { cout << "Roll Number Is:\n"<< rollno << "\n"; }
 };
class marks : public mm
 {
   protected:
     int sub1;
     int sub2;
   public:
     void get_marks(int x,int y)
          {
            sub1 = x;
            sub2 = y;
          }
     void put_marks(void)
          {
           cout << "Subject 1:" << sub1 << "\n";
           cout << "Subject 2:" << sub2 << "\n";
          }
  };
```

```
class res : public marks
  {
    protected:
      float tot;
    public:
      void disp(void)
           {
             tot = sub1+sub2;
             put_num();
             put_marks();
             cout << "Total:"<< tot;
           }
  };
int main()
  {
    res std1;
    std1.get_num(5);
    std1.get_marks(10,20);
    std1.disp();
    getch();
    return 0;
  }
```

Result:

```
Roll Number Is: 5
Subject 1: 10
Subject 2: 20
Total: 30
```

Hierarchical Inheritance

```
#include <iostream.h>
#include<conio.h>
class Side
 {
   protected:
     int l;
   public:
     void set_values (int x)
          { l=x;}
 };
class Square: public Side
 {
   public:
     int sq()
     { return (l *l); }
  };
class Cube:public Side
  {
   public:
     int cub()
     { return (l *l*l); }
  };
int main ()
  {
    Square s;
    s.set_values (10);
    cout << "The square value is::" << s.sq() << endl;
    Cube c;
    c.set_values (20);
    cout << "The cube value is::" << c.cub() << endl;
    getch();
    return 0;
  }
```

Result:

```
The square value is:: 100
The cube value is::8000
```

Hybrid Inheritance

```
#include <iostream.h>
class mm
  {
    protected:
      int rollno;
    public:
      void get_num(int a)
       { rollno = a; }
      void put_num()
       { cout << "Roll Number Is:"<< rollno << "\n"; }
  };
class marks : public mm
  {
   protected:
    int sub1;
    int sub2;
  public:
    void get_marks(int x,int y)
     {
           sub1 = x;
           sub2 = y;
     }
    void put_marks(void)
     {
            cout << "Subject 1:" << sub1 << "\n";
            cout << "Subject 2:" << sub2 << "\n";
     }
 };
class extra
{
   protected:
     float e;
   public:
   void get_extra(float s)
     {e=s;}
   void put_extra(void)
     { cout << "Extra Score::" << e << "\n";}
 };
class res : public marks, public extra{
  protected:
    float tot;
  public:
    void disp(void)
     {
          tot = sub1+sub2+e;
          put_num();
          put_marks();
          put_extra();
          cout << "Total:"<< tot;
     }
 };
int main()
 {
  res std1;
  std1.get_num(10);
  std1.get_marks(10,20);
  std1.get_extra(33.12);
  std1.disp();
return 0;
```

```
}
```

Result:

```
Roll Number Is: 10
Subject 1: 10
Subject 2: 20
Extra score:33.12
Total: 63.12
```

Polymorphism and Overloading

The this pointer (C++ Only)

The keyword this identifies a special type of pointer. Suppose that you create an object named x of class A, and class A has a nonstatic member function f(). If you call the function x.f(), the keyword this in the body of f() stores the address of x. You cannot declare the this pointer or make assignments to it.

A static member function does not have a this pointer.

The type of the this pointer for a member function of a class type X, is X* const. If the member function is declared with the const qualifier, the type of the this pointer for that member function for class X, is const X* const.

The this pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions.

```
#include <iostream>
struct X {
private:
  int a;
public:
  void Set_a(int a) {
    // The 'this' pointer is used to retrieve 'xobj.a'
    // hidden by the automatic variable 'a'
    this->a = a;
  }
   void Print_a() { cout << "a = " << a << endl; }
};
int main() {
  X xobj;
  int a = 5;
  xobj.Set_a(a);
  xobj.Print_a();
}
```

Poly refers many. So Polymorphism as the name suggests is a certain item appearing in different forms or ways. That is making a function or operator to act in different forms depending on the place they are present is called Polymorphism. Overloading is a kind of polymorphism.

In other words say for instance we know that +, - operate on integer data type and is used to perform arithmetic additions and subtractions. But operator overloading is one in which we define new operations to these operators and make them operate on different data types in other words overloading the existing functionality with new one. This is a very important feature of object oriented programming methodology which extended the handling of data type and operations.

Operator Overloading

In this C++ tutorial you will learn about Operator Overloading in two Parts, In Part I of Operator Overloading you will learn about Unary Operators, Binary Operators and Operator Overloading - Unary operators.

Operator overloading is a very important feature of Object Oriented Programming. Curious to know why!!? It is because by using this facility programmer would be able to create new definitions to existing operators. In other words a single operator can take up several functions as desired by programmers depending on the argument taken by the operator by using the operator overloading facility.

After knowing about the feature of operator overloading now let us see how to define and use this concept of operator overloading in C++ programming language.

We have seen in previous sections the different types of operators. Broadly classifying operators are of two types namely:

- Unary Operators
- Binary Operators

Unary Operators:

As the name implies, it operates on only one operand. Some unary operators are named ++ also called the Increment operator, -- also called the Decrement Operator, ! , ~ are called unary minus.

Binary Operators:

The arithmetic operators, comparison operators, and arithmetic assignment operators all this which we have seen in previous section of operators come under this category.

Both the above classification of operators can be overloaded. So let us see in detail each of this.

Operator Overloading - Unary operators

As said before operator overloading helps the programmer to define a new functionality for the existing operator. This is done by using the keyword operator.

The general syntax for defining an operator overloading is as follows:

```
 return_type classname :: operator operator_symbol(argument)

 {

 ...

 statements;

}
```

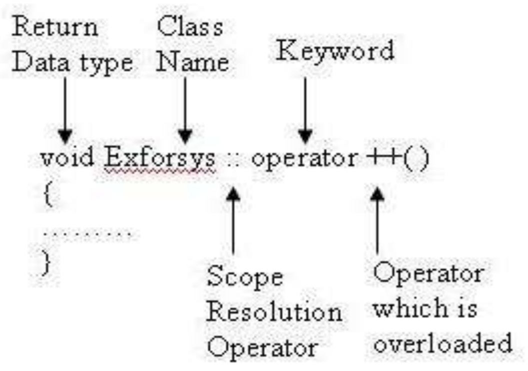
Thus the above clearly specifies that operator overloading is defined as a member function by making use of the keyword operator.

In the above:

- return_type - is the data type returned by the function
- class name - is the name of the class
- operator - is the keyword
- operator symbol - is the symbol of the operator which is being overloaded or
- defined for new functionality
- :: - is the scope resolution operator which is used to use the function definition outside the class. The usage of this is clearly defined in our earlier section of How to define class members.

For example

Suppose we have a class say Exforsys and if the programmer wants to define a operator overloading for unary operator say ++, the function is defined as


Return Data type
Class Name
Keyword
void Exforsys :: operator ++()
{
....
}
Scope Resolution Operator
Operator which is overloaded

Inside the class Exforsys the data type that is returned by the overloaded operator is defined as

```
class Exforsys

{

   private:

                  ------------

   public:

                  ----------

   void operator ++( );

};
```

So the important steps involved in defining an operator overloading in case of unary operators are:

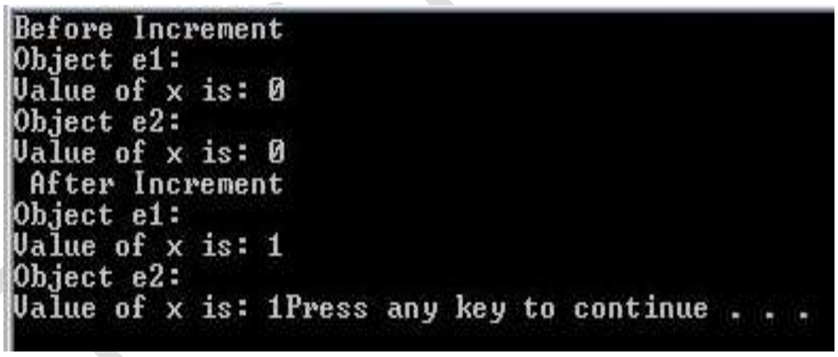
Inside the class the operator overloaded member function is defined with the return data type as member function or a friend function. The concept of friend function we will define in later sections. If in this case of unary operator overloading if the function is a member function then the number of arguments taken by the operator member function is none as seen in the below example. In case if the function defined for the operator overloading is a friend function which we will discuss in later section then it takes one argument.

The operator overloading is defined as member function outside the class using the scope resolution operator with the keyword operator as explained above

Now let us see how to use this overloaded operator member function in the program

```
#include <iostream.h>
class Exforsys
{
private:
      int x;
public:
        Exforsys( ) { x=0; }      //Constructor
        void display();
        void operator ++( );
  };
void Exforsys :: display()
  {
        cout << "nValue of x is: " << x;
  }
  void Exforsys :: operator ++( )  //Operator Overloading for operator ++ defined
  {
        ++x;
  }
  void main( )
  {
        Exforsys e1,e2;        //Object e1 and e2 created
        cout << "Before Increment";
        cout << "nObject e1: "; e1.display();
        cout << "nObject e2: "; e2.display();
        ++e1;  //Operator overloading applied
        ++e2;
        cout << "n After Increment";
        cout << "nObject e1: "; e1.display();
        cout << "nObject e2: "; e2.display();
}
```

The output of the above program is:



```
Before Increment
Object e1:
Value of x is: 0
Object e2:
Value of x is: 0
 After Increment
Object e1:
Value of x is: 1
Object e2:
Value of x is: 1Press any key to continue . . .
```

In the above example we have created 2 objects e1 and e2 f class Exforsys. The operator ++ is overloaded and the function is defined outside the class Exforsys.

When the program starts the constructor Exforsys of the class Exforsys initialize the values as zero and so when the values are displayed for the objects e1 and e2 it is displayed as zero. When the object ++e1 and ++e2 is called the operator overloading function gets applied and thus value of x gets incremented for each object separately. So now when the values are displayed for objects e1 and e2 it is incremented once each and gets printed as one for each object e1 and e2.

Operator overloading is a very important aspect of object-oriented programming. Binary operators can be overloaded in a similar manner as unary operators. In this C++ tutorial, you will learn about Binary Operators Overloading, explained along with syntax and example.

Operator Overloading - Binary Operators

Binary operators, when overloaded, are given new functionality. The function defined for binary operator overloading, as with unary operator overloading, can be member function or friend function.

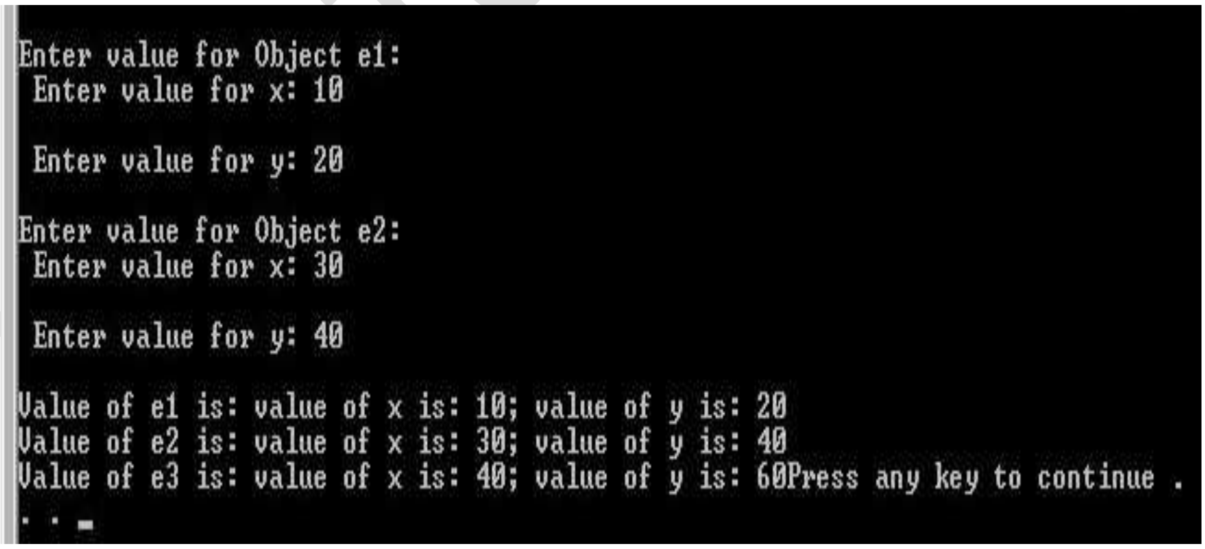
The difference is in the number of arguments used by the function. In the case of binary operator overloading, when the function is a member function then the number of arguments used by the operator member function is one (see below example). When the function defined for the binary operator overloading is a friend function, then it uses two arguments.

Binary operator overloading, as in unary operator overloading, is performed using a keyword operator.

Binary operator overloading example:

```
#include <iostream.h>
  class Exforsys
  {
  private:
        int x;
        int y;

  public:
        Exforsys()              //Constructor
        { x=0; y=0; }
void getvalue( )     //Member Function for Inputting Values
   {
              cout << "n Enter value for x: ";
              cin >> x;
              cout << "n Enter value for y: ";
              cin>> y;
   }
   void displayvalue( )   //Member Function for Outputting Values
    {
    cout << "value of x is: " << x << "; value of y is: " << y;
    }
        Exforsys operator +(Exforsys);
  };
  Exforsys Exforsys :: operator + (Exforsys e2)
      //Binary operator overloading for + operator defined
  {
        Exforsys rez; //declaring an Exforsys object to retain the final values
        int x1 = x+ e2.x;
        int y1 = y+e2.y;
        rez.x=x1;
        rez.y=y1;
        return rez;
  }
void main( )
{
        Exforsys e1,e2,e3;      //Objects e1, e2, e3 created
        cout << "\nEnter value for Object e1:";
        e1.getvalue( );
        cout << "nEnter value for Object e2:";
        e2.getvalue( );
        e3= e1+ e2;        //Binary Overloaded operator used
        cout << "\nValue of e1 is: "; e1.displayvalue();
        cout << "\nValue of e2 is: " ; e2.displayvalue();
        cout << "\nValue of e3 is: "; e3.displayvalue();
}
```

The output of the above program is:



```
Enter value for Object e1:
 Enter value for x: 10

 Enter value for y: 20

Enter value for Object e2:
 Enter value for x: 30

 Enter value for y: 40

Value of e1 is: value of x is: 10; value of y is: 20
Value of e2 is: value of x is: 30; value of y is: 40
Value of e3 is: value of x is: 40; value of y is: 60Press any key to continue .
. . _
```

In the above example, the class Exforsys has created three objects e1, e2, e3. The values are entered for objects e1 and e2. The binary operator overloading for the operator '+' is declared as a member function inside the class Exforsys. The definition is performed outside the class Exforsys by using the scope resolution operator and the keyword operator.

The important aspect is the statement:

e3= e1 + e2;

The binary overloaded operator '+' is used. In this statement, the argument on the left side of the operator '+', e1, is the object of the class Exforsys in which the binary overloaded operator '+' is a member function. The right side of the operator '+' is e2. This is passed as an argument to the operator '+' . Since the object e2 is passed as argument to the operator '+' inside the function defined for binary operator overloading, the values are accessed as e2.x and e2.y. This is added with e1.x and e1.y, which are accessed directly as x and y. The return value is of type class Exforsys as defined by the above example.

There are important things to consider in operator overloading with C++ programming language. Operator overloading adds new functionality to its existing operators. The programmer must add proper comments concerning the new functionality of the overloaded operator. The program will be efficient and readable only if operator overloading is used only when necessary.

- Some operators cannot be overloaded:
- scope resolution operator denoted by ::
- member access operator or the dot operator denoted by .
- the conditional operator denoted by ?:
- and pointer to member operator denoted by .*

Abstract Class

- An abstract class is a class that is designed to be specifically used as a base class.
- An abstract class contains at least one pure virtual function.
- You declare a pure virtual function by using a pure specifier (= 0) in the declaration of a virtual member function in the class declaration.

- You cannot use an abstract class as a parameter type, a function return type, or the type of an explicit conversion, nor can you declare an object of an abstract class.

```
#include <iostream.h>
#include<conio.h>
class MyInterface {
        public:
                virtual void Display() = 0;
};
class MyClass1 : public MyInterface {
        public:
                void Display() {
                cout << "MyClass1" << endl;
        }
};
```

```
class MyClass2 : public MyInterface {
public:
        void Display() {
        cout << "MyClass2" << endl;
    }
};
void main()
{
        clrscr();
        MyClass1 obj1;
        obj1.Display();
        MyClass2 obj2;
        obj2.Display();
        getch();
}
```

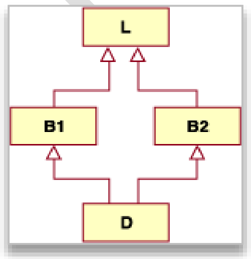
OUTPUT:

```
MyClass1
MyClass2
```

Virtual base Classes

When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited. Such a base class is known as virtual base class. This can be achieved by preceding the base class' name with the word virtual.

In the following example, an object of class D has two distinct subobjects of class L, one through class B1 and another through class B2. You can use the keyword virtual in front of the base class specifiers in the *base lists* of classes B1 and B2 to indicate that only one subobject of type L, shared by class B1 and class B2, exists.



Using the keyword virtual in this example ensures that an object of class D inherits only one subobject of class L.

```
#include<iostream.h>
#include<conio.h>
class A
{
    public:
    int i;
};
class B : virtual public A
{
    public:
    int j;
};
class C: virtual public A
{
    public:
    int k;
};
class D: public B, public C
{
    public:
    int sum;
};
int main()
{
clrscr();
D ob;
ob.i = 10; //unambiguous since only one copy of i is inherited.
ob.j = 20;
ob.k = 30;
ob.sum = ob.i + ob.j + ob.k;
cout <<"Value of i is : "<< ob.i<<"\n";
cout << "Value of j is : "<< ob.j<<"\n";
cout << "Value of k is :"<< ob.k<<"\n";
cout << "Sum is : "<< ob.sum <<"\n";
getch();
return 0;
}
```

Out Put:

```
Value of i is: 10
Value of j is: 20
Value of k is: 30
Sum is: 60
```

What are Virtual Functions?

Virtual, as the name implies, is something that exists in effect but not in reality. The concept of virtual function is the same as a function, but it does not really exist although it appears in needed places in a program. The object-oriented programming language C++ implements the concept of virtual function as a simple member function, like all member functions of the class.

The functionality of virtual functions can be overridden in its derived classes. The programmer must pay attention not to confuse this concept with function overloading. Function overloading is a different concept and will be explained in later sections of this tutorial. Virtual function is a mechanism to implement the concept of polymorphism (the ability to give different meanings to one function).

Need for Virtual Function:

The vital reason for having a virtual function is to implement a different functionality in the derived class.

For example: a Make function in a class Vehicle may have to make a Vehicle with red color. A class called FourWheeler, derived or inherited from Vehicle, may have to use a blue background and 4 tires as wheels. For this scenario, the Make function for FourWheeler should now have a different functionality from the one at the class called Vehicle. This concept is called Virtual Function.

Properties of Virtual Functions:

- Dynamic Binding Property:

Virtual Functions are resolved during run-time or dynamic binding. Virtual functions are also simple member functions. The main difference between a non-virtual C++ member function and a virtual member function is in the way they are both resolved. A non-virtual C++ member function is resolved during compile time or static binding. Virtual Functions are resolved during run-time or dynamic binding

- Virtual functions are member functions of a class.
- Virtual functions are declared with the keyword virtual, detailed in an example below.
- Virtual function takes a different functionality in the derived class.

Virtual Function:

Virtual functions are member functions declared with the keyword virtual.

```
class Vehicle //This denotes the base class of C++ virtual //function
{
public:
virtual void Make() //This denotes the C++ virtual function
{
cout << "Member function of Base Class Vehicle Accessed" ;
}
};
```

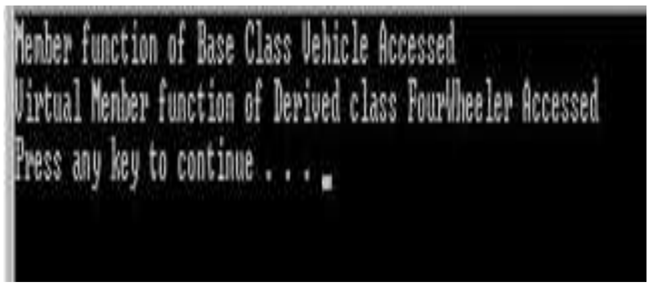
After the virtual function is declared, the derived class is defined. In this derived class, the new definition of the virtual function takes place.

When the class FourWheeler is derived or inherited from Vehicle and defined by the virtual function in the class FourWheeler, it is written as:

```
#include <iostream.h>
class Vehicle   //This denotes the base class of C++ virtual function
{
public:
virtual void Make()   //This denotes the C++ virtual function
{
   cout << "Member function of Base Class Vehicle Accessed" ;        }
 };

 class FourWheeler : public Vehicle
 {
 public:
     void Make()
     {
          cout << "Virtual Member function of Derived class FourWheeler Accessed" << endl;
     }
 };
void main()
 {
     Vehicle *a, *b;
     a = new Vehicle();
     a->Make();
     b = new FourWheeler();
     b->Make();
}
```

In the above example, it is evidenced that after declaring the member functions Make() as virtual inside the base class Vehicle, class FourWheeler is derived from the base class Vehicle. In this derived class, the new implementation for virtual function Make() is placed.

Output:



```
Member function of Base Class Vehicle Accessed
Virtual Member function of Derived class FourWheeler Accessed
Press any key to continue . . .
```

The programmer might be surprised to see the function call differs and the output is then printed as above. If the member function has not been declared as virtual, the base class member function is always called because linking takes place during compile time and is therefore static.

In this example, the member function is declared virtual and the address is bounded only during run time, making it dynamic binding and thus the derived class member function is called.

To achieve the concept of dynamic binding in C++, the compiler creates a v-table each time a virtual function is declared. This v-table contains classes and pointers to the functions from each of the objects of the derived class. This is used by the compiler whenever a virtual function is needed.

File in C++

The C++ standard libraries provide an extensive set of input/output capabilities which we will see in subsequent chapters. This chapter will discuss very basic and most common I/O operations required for C++ programming.

C++ I/O occurs in streams, which are sequences of bytes. If bytes flow from a device like a keyboard, a disk drive, or a network connection etc. to main memory, this is called **input operation** and if bytes flow from main memory to a device like a display screen, a printer, a disk drive, or a network connection, etc, this is called **output operation**.

So far we have been using the **iostream** standard library, which provides **cin** and **cout** methods for reading from standard input and writing to standard output respectively.

This tutorial will teach you how to read and write from a file. This requires another standard C++ library called **fstream** which defines three new data types:

Data Type	Description
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.

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.

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.

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

Mode Flag	Description
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::trunc	If the file already exists, its contents will be truncated before opening the file.

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.

Read & Write Example:

```
#include <fstream>
#include <iostream>
int main ()
{
   char data[100];
   // open a file in write mode.
   ofstream outfile;
   outfile.open("afile.dat");
   cout << "Writing to the file" << endl;
   cout << "Enter your name: ";
   cin.getline(data, 100);
   // write inputted data into the file.
           outfile << data << endl;
   cout << "Enter your age: ";
   cin >> data;
   cin.ignore();
   // again write inputted data into the file.
   outfile << data << endl;
   // close the opened file.
   outfile.close();
  // open a file in read mode.
   ifstream infile;
   infile.open("afile.dat");
   cout << "Reading from the file" << endl;
   infile >> data;
   // write the data at the screen.
            cout << data << endl;
   // again read the data from the file and display it.
   infile >> data;
   cout << data << endl;
   // close the opened file.
  infile.close();
}
```

Oputput:

```
$./a.out
Writing to the file
Enter your name: Zara
Enter your age: 9
Reading from the file
Zara
9
```

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 ostream.

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.

The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are:

```
// position to the nth byte of fileObject (assumes ios::beg)
        fileObject.seekg( n );
// position n bytes forward in fileObject
        fileObject.seekg( n, ios::cur );
// position n bytes back from end of fileObject
        fileObject.seekg( n, ios::end );
// position at end of fileObject
        fileObject.seekg( 0, ios::end );
```

String in C++

C++ provides following two types of string representations:

1. The C-style character string.
2. The string class type introduced with Standard C++.

```
#include <iostream>
using namespace std;
int main ()
{
   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
   cout << "Greeting message: ";
   cout << greeting << endl;
   return 0;
}
```

Output : Greeting message: Hello

C++ supports a wide range of functions that manipulate null-terminated strings:

S.N.	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

```
#include <iostream>
#include <cstring>
int main ()
{
  char str1[10] = "Hello";
  char str2[10] = "World";
  char str3[10];
  int  len ;
  // copy str1 into str3
  strcpy( str3, str1);
  cout << "strcpy( str3, str1) : " << str3 << endl;
  // concatenates str1 and str2
  strcat( str1, str2);
  cout << "strcat( str1, str2): " << str1 << endl;
  // total lenghth of str1 after concatenation
  len = strlen(str1);
  cout << "strlen(str1) : " << len << endl;
}
```

Output:

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

The string Class in C++

The standard C++ library provides a string class type that supports all the operations mentioned above, additionally much more functionality. We will study this class in C++ Standard Library but for now let us check following example:

At this point you may not understand this example because so far we have not discussed Classes and Objects. So can have a look and proceed until you have understanding on Object Oriented Concepts.

```
#include <iostream>
#include <string>
int main ()
{
  string str1 = "Hello";
  string str2 = "World";
  string str3;
  int  len ;
  // copy str1 into str3
  str3 = str1;
  cout << "str3 : " << str3 << endl;
  // concatenates str1 and str2
  str3 = str1 + str2;
  cout << "str1 + str2 : " << str3 << endl;
  // total lenghth of str3 after concatenation
  len = str3.size();
  cout << "str3.size() :  " << len << endl;
}
```

Output: str3 : Hello
```
            str1 + str2 : HelloWorld
            str3.size() :  10
```

C++ Memory Management Operators

Need for Memory Management operators

The concept of arrays has a block of memory reserved. The disadvantage with the concept of arrays is that the programmer must know, while programming, the size of memory to be allocated in addition to the array size remaining constant.

In programming there may be scenarios where programmers may not know the memory needed until run time. In this case, the programmer can opt to reserve as much memory as possible, assigning the maximum memory space needed to tackle this situation. This would result in wastage of unused memory spaces. Memory management operators are used to handle this situation in C++ programming language.

What are memory management operators?

There are two types of memory management operators in C++:

- new
- delete

These two memory management operators are used for allocating and freeing memory blocks in efficient and convenient ways.

New operator:

The new operator in C++ is used for dynamic storage allocation. This operator can be used to create object of any type.

General syntax of new operator in C++:

The general syntax of new operator in C++ is as follows:

pointer variable = new datatype;

In the above statement, new is a keyword and the pointer variable is a variable of type datatype.

For example:

int *a=new int;

In the above example, the new operator allocates sufficient memory to hold the object of datatype int and returns a pointer to its starting point. The pointer variable a holds the address of memory space allocated.

Dynamic variables are never initialized by the compiler. Therefore, the programmer should make it a practice to first assign them a value.

The assignment can be made in either of the two ways:

```
 int *a = new int;
 *a = 20;
```

Or

int *a = new int(20);

Delete operator:

The delete operator in C++ is used for releasing memory space when the object is no longer needed. Once a new operator is used, it is efficient to use the corresponding delete operator for release of memory.

General syntax of delete operator in C++:

The general syntax of delete operator in C++ is as follows:

delete pointer variable;

In the above example, delete is a keyword and the pointer variable is the pointer that points to the objects already created in the new operator. Some of the important points the programmer must note while using memory management operators are described below:

- The programmer must take care not to free or delete a pointer variable that has already been deleted.
- Overloading of new and delete operator is possible (to be discussed in detail in later section on overloading).
- We know that sizeof operator is used for computing the size of the object. Using memory management operator, the size of the object is automatically computed.
- The programmer must take care not to free or delete pointer variables that have not been allocated using a new operator.

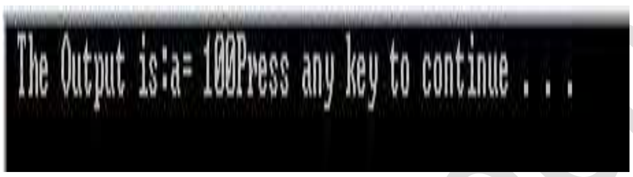
- Null pointer is returned by the new operator when there is insufficient memory available for allocation.

Example:

To understand the concept of new and delete memory management operator in C++:

```
#include <iostream.h>
void main()
{
        //Allocates using new operator memory space in memory
        //for storing a integer datatype
        int *a= new int;
        *a=100;
        cout << " The Output is:a= " << *a;
        //Memory Released using delete operator
        delete a;
}
```

The output of the above program is



```
The Output is:a= 100Press any key to continue . . .
```

In the above program, the statement:

int *a= new a;

Holds memory space in memory for storing a integer datatype. The statement:

*a=100

This denotes that the value present in address location pointed by the pointer variable a is 100 and this value of a is printed in the output statement giving the output shown in the example above. The memory allocated by the new operator for storing the integer variable pointed by a is released using the delete operator as:

delete a;

Templates

Independent concepts should be independently represented and should be combined only when needed. Where this principle is violated, you either bundle unrelated concepts together or create unnecessary dependencies. Either way, you get a less flexible set of components out of which to compose systems. Templates provide a simple way to represent a wide range of general concepts and simple ways to combine them. The resulting classes and functions can match hand-written, more-specialized code in run-time and space efficiency.

Templates provide direct support for generic programming, that is, programming using types as parameters. The C++ template mechanism allows a type to be a parameter in the definition of a class or a function. A template depends only on the properties that it actually uses from its parameter types and does not require different types used as arguments to be explicitly related. In particular, the argument types used for a template need not be from a single inheritance hierarchy.

The format for declaring function templates with type parameters is:
template<classidentifier>function_declaration;
template <typename identifier> function_declaration;

The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

For example, to create a template function that returns the greater one of two objects we could use:

```
template <class myType>
myType GetMax (myType a, myType b)
{
        return (a>b?a:b);
}
```

Here we have created a template function with myType as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. As you can see, the function template GetMax returns the greater of two parameters of this still-undefined type.

To use this function template we use the following format for the function call:

```
int x,y;
GetMax <int> (x,y);
```

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of myType by the type passed as the actual template parameter (int in this case) and then calls it.

This process is automatically performed by the compiler and is invisible to the programmer.

Example of Template Class

```
// class templates
#include <iostream.h>
template <class T>
class mypair {
          T a, b;
          public:
          mypair (T first, T second)
                    {a=first; b=second;}
          T getmax ();
};
template <class T>
T mypair<T>::getmax ()
{
          T retval;
          retval = a>b? a : b;
          return retval;
}
int main () {
          mypair <int> myobject (100, 75);
          cout << myobject.getmax();
          return 0;
}
```

Out Put:

```
100
```

Example of Function Templates with Multiple Parameters

```
#include <iostream.h>
template <class T, int N>
class mysequence
{
T memblock [N];
public:
void setmember (int x, T value);
T getmember (int x);
};
```

```
template <class T, int N>
void mysequence<T,N>::setmember (int x, T value)
{
memblock[x]=value;
        }
template <class T, int N>
T mysequence<T,N>::getmember (int x)
{
        return memblock[x];
}
int main ()
{
        clrscr();
        mysequence <int,5> myints;
        mysequence <double,5> myfloats;
        myints.setmember (0,100);
        myfloats.setmember (3,3.1416);
        cout << myints.getmember(0) << '\n';
        cout << myfloats.getmember(3) << '\n';
        getch();
        return 0;
}
```

Out Put:

```
100
3.1416
```

Command Line Argument in C/C++

Command-line arguments are given after the name of a program in command-line operating systems like DOS or Linux, and are passed in to the program from the operating system. To use command line arguments in your program, you must first understand the full declaration of the main function, which previously has accepted no arguments. In fact, main can actually accept two arguments: one argument is number of command line arguments, and the other argument is a full list of all of the command line arguments.

```
#include <iostream.h>
#include<conio.h>
int main(int argc, char **argv)
{
   clrscr();
```

```
    cout << "\n\nReceived " << argc << " arguments...\n";
    for (int i=0; i<argc; i++)
    cout << "argument " << i << ": " << argv[i] << endl;
    getch();
    return 0;
}
```