By Ganesh Kumar Butcha
Published: 2009-01-07 18:03

Learning C/C++ Step-By-Step 01. Step-by-Step C/C++ --- Introduction

Many people are really interested in learning and implementing C/C++ programs on their favorite platforms like DOS/Windows or Linux. If you are the one looking for a step-by-step guide to get started, this tutorial is for you. Let me know your comments on my tiny attempt to serve the community.

Contents

Introduction to Language & ExpressionsWhat is C?

C is a compiler based programming language supports both high level and low level statements to interact directly with the hardware.

Development of C Language

The C programming language evolved from a succession of programming languages developed at Bell Laboratories in early 1970s. It was not until the late 1970s that this programming language began to gain widespread popularity and support. This was because until that time C compilers were not readily available for commercial use outside of Bell Laboratories.

The C language was the outcome of Dennis Ritchie(TM)s work on a project in Bell Laboratories, to invent a suitable high level language for writing an Operating System which manages the input and output devices of a computer, allocates its storage and schedules the running of other programs.

UNIX operating system is written in the C language. Hence the Unix Operating system has C as its standard programming language. In fact over 90% of the operating system itself is written in the C language. So originally C language was designed and implemented on the Unix Operating System.

C as a general purpose Language

C is a high level, procedural/structured, and general purpose programming language and resembles few other high level languages such as Fortran, Pascal, and PL/1. However, we cannot call the C language as a œPurely High Level Language•.

C stands somewhere between the high-level languages meant for carrying on special activities and the low level languages such as assembly language of a machine because of some features like œSystem Independence•, œLimited Data Type•, œHigh Flexibility•, it is considered as a powerful language C has also become popular because of its portability across systems.

History of C

Year	Language	Developed by	Remarks
1960	ALGOL	International Committee	Too general, Too abstract
1963	CPL	Cambridge University	Hard to learn, Difficult to implement
1967	BCPL	Martin Richards	Could deal with only specific problems
1970	B	Ken Thompson AT & TBell Labs	Could deal with only specific problems
1972	C	Dennis Ritchie AT & TBell Labs	Lost generality of BCPL and B restored
Early 80(TM)s	C++	Bjarne Stroustrup AT & T	Introduces OOPs to C.

Features of C

- Simple, versatile, general purpose language

- Programs are fast and efficient
- Has got rich set of operators
- more general and has no restrictions
- can easily manipulates with bits, bytes and addresses
- Varieties of data types are available
- separate compilation of functions is possible and such functions can be called by any C program
- block-structured language
- Can be applied in System programming areas like operating systems, compilers & Interpreters, Assemblers etc.,

II. Programming BasicsComponents of a program

1. Constants
2. Variables
3. Operators
4. Statements

So, before writing serious programming we must be clear with all the above components of programs. According to above example every program is a set of statements, and statement is an instruction to the computer, which is a collection of constants, variables, operators and statements.

Constants

A constant is a fixed value, which never altered during the execution of a program.
Constants can be divided into two major categories:

1. Primary Constants
2. Secondary Constants

Data Types

The kind of data that the used variables can hold in a programming language is known as the data type.

Basic data types are as follows:

1. Numeric Data Type
2. Non-Numeric Data Type
3. Integer Data Type
4. Real Data Type
5. Logical Data Type
6. Enumerated Data Type

1. Numeric Data Type: Totally deals with the numbers. These numbers can be of integer (int) data type or real (float) data type.

2. Non-Numeric Data Type : Totally deals with characters. Any character or group of characters enclosed within quotes will be considered as non-numeric or character data type.

3. Integer Data Type : Deals with integers or whole numbers. All arithmetic operations can be achieved through this data type and the results are again integers.

4. Real Data Type : deals with real numbers or the numeric data, which includes fractions. All arithmetic operations can be achieved through this data type and the results can be real data type.

5. Logical or Boolean Data Type : can hold only either of the two values TRUE or FALSE at a time. In computer, a 1 (one) is stored for TRUE and a 0 (zero) is stored for FALSE.

6. Enumerated Data Type : Includes the unstructured data grouped together to lead to a new type. This data type is not standard and us usually defined by user.

Ex.

```
Week_days = { œsun•, œmon•, œtue•, œwed•, œthu•, œfri•, œsat• };
Directions = {•North•, œEast•, œWest•, œSouth• };
```

The following table shows the standard data types with their properties.

Keyword

Range: low

Range: high

Digits of precision

Bytes of memory

Format-ID

vhar

-128

127

n/a

1

%c

int

-32, 768

32, 767

N/a

2 (on 16 bit processor)

%d

long

-2,147, 483, 648

2, 147, 483, 647

N/a

4

%ld

float

3.4 x 10-38

3.4 x 1038

7

4

%f

double

1.7 x 10-308

1.7 x 10308

15

8

%lf

long double

3.4 x 10-4932

3.4 x 10-4932

19

10

%Lf

NOTE: ***The required ranges for signed and unsigned int***are identical to those for signed and unsigned short. Oncompilers for 8 and 16 bit processors (including Intel x86processors executing in 16 bit mode, such as under MS-DOS), anint is usually 16 bits and has exactly the same representation asa short. On compilers for 32 bit and larger processors (includingIntel x86 processors executing in 32 bit mode, such as Win32 orLinux) an int is usually 32 bits long and has exactly the samerepresentation as a long.

I want you to refer this page for more information on int type for different processors:

Ref: ***http://www.jk-technology.com/c/inttypes.html***

02. Step-by-Step C/C++ --- IDE and Compilers for C/C++

C / C++ is a compiler based programming languages. In order to run a program you need a compiler software (i.e., GNU GCC, Tiny C, MS Visual C++,

Cygwin C, Borland, Intel C etc..). Also you need an IDE to create/edit programs (eg: Dev-C++, Code::Blocks, Eclipse, TurboC, etc..)

I am giving you a couple of examples of my favorite compiler and IDEs, You may choose the best from the vast list.

1. Installing GNU GCC Compiler

1. Installing GNU GCC Compiler1.1. For Linux

- For Redhat, get a gcc-c++ RPM, e.g. using Rpmfind and then install (as root) using

```
rpm -ivh gcc-c++-version-release.arch.rpm
```

- For Fedora Core/ CentOS, install the GCC C++ compiler (as root) by using

```
yum install gcc-c++
```

- For Mandrake, install the GCC C++ compiler (as root) by using

```
urpmi gcc-c++
```

- For Debian, install the GCC C++ compiler (as root) by using

```
apt-get install g++
```

- For Ubuntu, install the GCC C++ compiler by using

```
sudo apt-get install g++
```

- If you cannot become root, get the tarball from ***ftp://ftp.gnu.org/*** and follow the instructions in it to compile and install in your home directory.

1.2. For Mac OS X

Xcode has GCC C++ compiler bundled.

1.3. For Windows (MinGW + DevCpp-IDE)

- Go to ***http://www.bloodshed.net/devcpp.html***, choose the version you want (eventually scrolling down), click on the appropriate download link! For the most current version, you will be redirected to ***http://www.bloodshed.net/dev/devcpp.html***
- Scroll down to read the license and then to the download links. Download a version with Mingw/GCC. It's much easier than to do this assembling yourself. With a very short delay (only some days) you will always get the most current version of mingw packaged with the devcpp IDE. It's absolutely the same as with manual download of the required modules.
- You get an executable that can be executed at user level under any WinNT version. If you want it to be setup for all users, however, you need admin rights. It will install devcpp and mingw in folders of your wish.
- Start the IDE and experience your first project!

You will find something mostly similar to MSVC, including menu and button placement. Of course, many things are somewhat different if you were familiar with the former, but it's as simple as a handfull of clicks to let your first program run.

1.4. How to Create, Compile and Execute Programs

If you are using Linux, create/edit a program:

```
vi hello.cpp
```

Compilation:

```
g++ -Wall -g -o hello.out hello.cpp
```

Running a program:

```
 ./hello.out
```

1.5. Example Programs:C Example Program:

```
\* 0001_hello.c *\

#include <stdio.h>

int main()

{

      printf("\nHello world");

      return 0;

}
```

C++ Example Program:

```
\* 0001_hello.cpp *\

#include <iostream>

using namespace std;

int main()

{

     cout << endl << "Hello, Happy programming";

     return 0;

}
```

03. Step-by-Step C/C++ --- C Programming - Basic IO StatementsContents

- Structure of a C program
- I/O Statements
- Printf
- Escape Characters
- Using Variables in programs
- Scanf

- More IO Statements
- gets
- puts
- getch
- putch
- getche
- getchar

As discussed, every program is a set of statements, and statement is an instruction to the computer, which is a collection of constants, variables, operators and statements.

Structure of a C program

```
<return type> main( arg-list )

{

     <declaration part>



     <Statement block>



     <Return Values    >

}
```

We are going to start with Input / Output Statements as they play important roles in our further programs.

I/O Statements

Printf

This statement displays the given literal / prompt / identifiers on the screen with the given format.

Syntax:

```
printf(<"prompt/literal/format id/esc char.     ">, id1,id2, .....);
```

E.g.:

```
       printf("Hello");
  printf("Student number   :  %d", sno);
  printf("Student name    :  %s", name);
printf("3Subjects Marks   :  %d, %d, %d", m1, m2, m3);
```

1. Program to print a message:

```
/* 02_print.c */

#include <stdio.h>

int main( )

{

     printf("Hello");

     return 0;

}
```

Escape Characters

Common Escape Sequences

Escape Sequence

Character

a

Bell(beep)

b

Backspace

f

Form feed

n

New line

r

Return

t

Tab

\

Backslash

(TM)

Single quotation mark

•

Double quotation marks

xdd

Hexadecimal representation

2. Program to print a message in a new line
- Compare with the last program.

```
/* 03_esc.c */

#include <stdio.h>

int main()

{

     printf("\nHello");

     return 0;
```

```
}
```

3. Program to display address of a person
 - Multiple statements in main

```
/* 04_multi.c */

#include <stdio.h>

int main()

{

     printf("\nName of the Person");

     printf("\nStreet, Apartment//House No. ");

     printf("\nzip, City");

     printf("\nCountry");

     return 0;

}
```

Using Variables in programs

Basic Variable Types

Keyword

Range: low

Range: high

Digits of precision

Bytes of memory

Format-ID

Char

-128

127

n/a

1

%c

Int

-32, 768

32, 767

N/a

2

%d

Long

-2,147, 483, 648

2, 147, 483, 647

N/a

4

%ld

Float

3.4 x 10-38

3.4 x 1038

7

4

%f

Double

1.7 x 10-308

1.7 x 10308

15

8

%lf

long double

3.4 x 10-4932

3.4 x 10-4932

19

10

%Lf

4. Program to find the sum of two values

- Variables are introduced in this program

```
/* 05_var.c */

#include <stdio.h>

int main()

{
```

```
    int a , b , c;

    a = 5;

    b = 10;

    c = a + b;

    printf("%d", c);

    return 0;

}
```

5. Program to find the sum of two values with message
- Compare with the last program

```
#include <stdio.h>

int main()

{

    int a, b, c;

    a = 5;

    b = 10;

    c = a + b;
```

```
    printf("\nSum is   %d", c);

    /* We have inserted extra text before printing the value*/

    return 0;

}
```

Scanf

Using this statement we can accept and values to variables during the execution of the program.

Syntax:

```
scanf(<format id/esc char">, id1,id2, .....);
```

Eg.

```
scanf("%d", &sno);
scanf("%s", name);
scanf("%d%d%d", &m1, &m2, &m3);
```

6. Program to find the sum of two value using scanf

- When you run the program it shows you the cursor and waits for your input, enter a numeric value and press "Return", do this twice and you will get the output.

```
/* 07_scanf.c */

#include <stdio.h>
```

```
int main()

{

     int a , b, c;      scanf("%d", &a);

     scanf("%d", &b);

     c = a + b;

     printf("\nSum is   %d", c);

     return 0;

}
```

More Excercises:

7. Program to find the sum of two values with message display
 - Messages are optional but introduces user-friendly interaction
- Compare with the last program

```
/* 08_sum.c */

#include <stdio.h>

int main()

{

     int a , b, c;      printf("Enter A value ");  scanf("%d", &a);
```

```
    printf("Enter B value ");  scanf("%d", &b);

    c = a + b;

    printf("\nSum is   %d", c);

    return 0;

}
```

8. Program to find the result of (a+ b)2

- Similar to sum of two values program but the formulae is different

```
/* 09_formula.c */

#include <stdio.h>

int main()

{

    int a, b, c;

    printf("Enter A value "); scanf("%d", &a);

    printf("Enter B value "); scanf("%d", &b);

    c = a * a + b * b + 2 * a * b;
```

```
    printf("Result is       %d", c);

    return 0;

}
```

9. Program to find the annual salary of an employee
- input : eno, name, sal
- Process : Asal = sal * 12
- Output : Eno, name, sal, asal
- This program introduces the different types of variable

```
/* 10_emp.c */

#include <stdio.h>

int main()

{

    int eno;

    char name[10];                  /*  name with 10 characters width */

    float sal, asal;                      /* sal & asal as real values */

    printf("Enter Employee number  "); scanf("%d", &eno);

    printf("Enter Employee name     "); scanf("%s", name);

    printf("Enter Employee salary    "); scanf("%f", &sal);
```

```
    asal = sal * 12;

    printf("\nEmployee number      %d", eno);

    printf("\nEmployee name         %s", name);

    printf("\nEmployee salary        %f", sal);

    printf("\nAnnual Salary            %f", asal);

    return 0;

}
```

10. Write a program to find the total and average marks of a student
- Input : Sno, name, sub1, sub2, sub3
- process : total = sub1 + sub2 + sub3; avg = total / 3
- output : sno, name, total, avg
- Similar to the above program just accept, process, and print the values

```
/* 11_stud.c */

#include <stdio.h>

int main()

{

    int sno, sub1, sub2, sub3, total;
```

```
char name[10];

float avg;



clrscr();               /* clear the screen before its output  */

printf("Enter Student number      "); scanf("%d", &sno);

printf("Enter Student name    "); scanf("%s", name);

printf("Enter Subject1 marks    "); scanf("%d", &sub1);

printf("Enter Subject2 marks    "); scanf("%d", &sub2);

printf("Enter Subject3 marks    "); scanf("%d", &sub3);



total = sub1 + sub2 + sub3;

avtg = total / 3;



printf("\nStudent number    %d", sno);

printf("\nStudent name    %s", name);

printf("\nTotal marks        %d", total);
```

```
    printf("\nAverage marks    %f" , avg);

    return 0;

}
```

More IO Statements

Gets:

To accept a string from the key board. It accepts string value up to the carriage return.

Syntax:

```
gets( <id.> );
```

E.g.:

```
gets(name);
gets(street);
```

puts

It displays the given string value on the screen.

Syntax:

```
puts( <id.> / <œprompt•>);
```

E.g.:

```
puts(name);
puts(street);
```

getch - Read char without echo

getche - read char with echo

getchar - read char and accept carriage return

putch

It can print a character on the screen.

Syntax:

```
putch(<char>).
```

E.g.:

```
putch(˜a(TM));
putch(65);
```

getch

It accepts a character from console.

Syntax:

```
char = getch().
```

E.g.:

```
  ch = getch();
option = getch();
```

04. Step-by-Step C/C++ --- C Programming - Conditional Statements

- Introduction to Conditional Statements:
- if..else
- switch

1. Introduction to Conditional Statements:

A computer is an electronic device which can perform arithmetic operations as well logical decisions.

At this point, computer is far away from an ordinary calculator which able to perform only arithmetic operations.

We can ask the biggest value from the given two values using conditional statements like if-else, switch.

2. if..else

It is a conditional statement to find the variance between two expressions.

Syntax:

```
if ( <condition> )

     { <St.block>; }
```

```
else

     { <St block>;  }
```

Every if has a condition and two statement blocks. If the condition is true it executes the first st.block and vice versa.
Eg.

```
If( a>b )

     printf(â€œA is bigâ€•);

else

     printf(â€œB is bigâ€•);
```

Note: No need of block for ***Single*** statements.

1. Program to find the biggest of 2 values

```
/* 12_if.c */

#include <stdio.h>

int main()

{            /* Begin */

     int a, b;             /* Declaration of Variables */
```

```
     printf("\nEnter A value : "); scanf("%d", &a);      /* Read value A */

     printf("\nEnter B value : "); scanf("%d", &b);       /* Read value B */

     if( a>b )           /* Compare both */

          printf("A is big");

     else

          printf("B is big");        /* Print the result */

      return 0;

}      /* End */
```

This is a list of ***operators*** in the ***C++*** and ***C programming languages***. All the operators listed exist in C++

Ref: ***http://en.wikipedia.org/wiki/Operators_in_C_and_C++***

Arithmetic Operators

Operator Purpose

+ Addition
- Subtraction
* Multiplication
/ Division
% Remainder after integer division (modulus)

Unary Operators

Operator Purpose

- Minus (negative number)
++ Increment (increase by 1)

-- Decrement (decrease by 1)
sizeof Size, in bytes
(***type***) Cast

Relational Operators

Operator Purpose
< Less Than
<= Less Than Or Equal To
> Greater Than
>= Greater Than Or Equal To

Equality Operators

Operator Purpose
== Equal To
!= Not Equal To

Logical Operators

Operator Purpose
&& AND
|| OR
! NOT

Bit-Manipulating Operators

Operator Purpose
& AND
| OR
~ NOT
^ XOR
<< Shift Left
>> Shift Right

Operator Precedence Groups

Operator Category Operators Associativity

unary operators - ++ -- ! sizeof (***type***) R to L
arithmetic multiply, divide and remainder * / % L to R
arithmetic add and subtract + - L to R
relational operators < <= > >= L to R
equality operators == != L to R
logical operators && || L to R
conditional operators ? : R to L
assignment operators = += -= *= /= %= R to L

More Excercises

The reason behind more exercises is to get acquainted with the learned statements, if you are confident you don(TM)t have to run the following programs.

/* 01. Program to find the age of a person from the following details */
/* age <= 12 Child Age
age >= 13 and age <= 19 Teen Age
age >= 20 and age <= 35 Young Age
age >= 36 and age < 50 Middle Age
age >= 50 Old Age
*/

```
/* 13_age.c */

#include <stdio.h>

int main()

{

     char name[20];

     int age;
```

```
     clrscr();

     print "Enter U'r name  ";  input name;

     print "Enter U'r age   ";  input age;

      printf("\n%s U are in     ");

      if ( age <= 12 )                printf("Child Age");

       if ( age >= 13 and age <= 19 )  printf("Teen Age");

      if ( age >= 20 and age <= 35 ) printf("Young Age");

       if ( age >= 36 and age <  50 ) printf("Middle Age");

      if ( age >= 50 )   printf("Old Age");

      return 0;

}
```

/* 02. Program to find the biggest of 3 Values */

```
/* 14_big3.c */

#include <stdio.h>

#include <conio.h>

int main()
```

```
{

     int a, b, c;

     clrscr();

     printf("Enter A value   "); scanf("%d", &a);

     printf("Enter B value   "); scanf("%d", &b);

     printf("Enter C value   "); scanf("%d", &c);

     if( a > b && a > c ) printf( "A is big " );

     if( b > a && b > c ) printf( "B is big " );

     if( c > a && c > b ) printf( "C is big " );

     return 0;

}
```

/* 03. Program to find the biggest of 3 Values using if..else */

```
/* 15_big3.c */

#include <stdio.h>

#include <conio.h>

int main()
```

```
{

     int a, b, c;

     clrscr();

     printf("Enter A value   "); scanf("%d", &a);

     printf("Enter B value   "); scanf("%d", &b);

     printf("Enter C value   "); scanf("%d", &c);

     if( a > b && a > c )

          printf( "A is big " );

     else

          if ( b > c )

               printf( "B is big " );

          else

               printf( "C is big " );

     return 0;

}
```

/* 04. Program to find the biggest of 3 Values using nested if */

```
/* 16_big3.c */

#include <stdio.h>

#include <conio.h>

int main()

{

     int a, b, c;

     clrscr();

     printf("Enter A value   "); scanf("%d", &a);

     printf("Enter B value   "); scanf("%d", &b);

     printf("Enter C value   "); scanf("%d", &c);

     if( a > b )

          if( a > c )

               printf(" A is big ");

          else

               printf(" C is big ");

     else

          if( b > c )
```

```
            printf(" B is big ");

        else

            printf(" C is big ");

    return 0;

}
```

/* 05. To find the week day of the given number */

```
/* 17_week.c */

#include <stdio.h>

int main()

{

    int week;

    printf("Enter week number  "); scanf("%d", &week);

    if (week == 1  )  printf ("Sunday");

    if (week == 2  )  printf ("Monday");

    if (week == 3  ) printf ("Tuesday");

    if (week == 4  ) printf ("Wednesday");
```

```
    if (week == 5  )  printf ("Thursday");

    if (week == 6  ) printf ("Friday");

    if (week == 7  ) printf ("Saturday");

    if ( week < 1 || week > 7 ) printf("Bad Day");

    return 0;

}
```

3. Switch

A multi-conditional st. has the ability to check the variance of more than one expression.

Syntax:

```
switch(<id>)

{

    case <expr.> : <st. block>; break;

    case <expr.> : <st. block>; break;

    ......

    Default : <st. block>;
```

```
}
```

Eg.

```
switch(week)

{

     case 1 : printf( â€œSun Dayâ€•); break;

     case 2 : printf(â€œMon Dayâ€•); break;

     .

     .

     case 7: printf(â€œSatur Dayâ€•); break;

     default : printf(â€œWrong Entryâ€•);

}
```

/* 06. To find the week day of the given number using switch statement */

```
/* 18_switch.c */

#include <stdio.h>

int main()
```

```
{

     int week;

     printf("Enter week number  "); scanf("%d", &week);

     switch(week)

     {

          case 1 : printf ("Sunday"); break;

          case 2 : printf ("Monday"); break;

          case 3 : printf ("Tuesday"); break;

          case 4 : printf ("Wednesday"); break;

          case 5 : printf ("Thursday"); break;

          case 6 : printf ("Friday"); break;

          case 7 : printf ("Saturday"); break;

          default : printf("Wrong Entry");

     }

     return 0;

}
```

05. Step-by-Step C/C++ --- C Programming - Looking Statements

- Branching Statement - goto
- Looping Statements
for
while
do..while

1. Branching Statement

goto

It transfers the control pointer from one place to another in the current program.

Syntax:

```
goto <label>;
```

Note: Label name must be defined with colon(:) and it should not exceed more than 32 characters in length.

Eg.

```
abc:

     printf(â€œHelloâ€•);

     goto abc;
```

/* 01. A demonstration program to illustrate goto statement */

```
/* 19_goto.c */

#include <stdio.h>
```

```
int main()

{

     abc:        /* Label name */



          printf("\nHello");

     goto abc;     /* branching statement */

     return 0;

}
```

/* 07. Continuous execution will be stopped with a carry varaible and a conditional statement */
/* Find the difference between the last program and this, note all the differences in this program*/

```
/* 20_goto.c */

#include <stdio.h>

int main()

{

     int i = 1;

     abc:
```

```
          printf("\nHello");

          i ++;

          if ( i<= 10 )   /* Take care of this statement */

     goto abc;

     return 0;

}
```

2. Looping Statements

`for`

An iterative statement to execute a statement block for a number of times.

Syntax:

```
for(<initialization> ; <condition> ; <step value>)

{

     <st. block>

}
```

Eg.

```
for(I=1;I<=10; I++)
```

```
     printf(â€œ\n%dâ€•,i);
```

Eg.

```
for(I=1, j = 0; I<10; I+=2, j+=2)

     printf(â€œ%d      %d\nâ€•, i, j);
```

/* 08. To print a message 5 times */

```
/* 21_for.c */

#include <stdio.h>

int main()

{

     int i;

     for(i = 1; i <= 5; i++ )

          printf("\nHello");

     return 0;

}
```

/* 09. To print a message with it's count upto 5 times */

```
/* 22_hello.c */

#include <stdio.h>

int main()

{

     int i;

     for(i = 1; i <= 5; i++ )

          printf("\nHello - %d", i);

     return 0;

}
```

/* 10. To print 1 to 10 natural numbers */

```
/* 23_nat.c */

#include <stdio.h>

int main()

{

     int i;

     for(i = 1; i <= 10; i++ )
```

```
          printf("\n%d", i);  /* Eleminating message */

     return 0;

}
```

/* 11. To print second multiplication table */
/* Note : Compare it, with the last program */

```
/* 24_table.c */

#include <stdio.h>

int main()

{

     int i;

     for(i = 1; i <= 20; i++ )

          printf("\n%d * 2 = %d", i, i * 2);

     return 0;

}
```

/* 12. To print a multiplication table for the given number */

```
/* 25_tablen.c */
```

```
#include <stdio.h>

#include <conio.h>

int main()

{

     int i, t;                  /*  a new variable 't'  */

     clrscr();

     printf("Which table to print   :"); scanf("%d", &t);

     for(i = 1; i <= 20; i++ )

          printf("\n%d * %d = %d", i, t, i * t);

     return 0;

}
```

/* 13. To print a multiplication table for the given number */
/* Note : Compare it, with the last program */

```
/* 26_tablen.c */

#include <stdio.h>

int main()

{
```

```
    int i, t;

    clrscr();

    for(t = 1; t <= 20; t++)    /* One more for loop  */

         for(i = 1; i <= 20; i++ )

              printf("\n%d * %d = %d", i, t, i * t);

    return 0;

}
```

/* 14. To print numbers in triangle form */
/* Note : Compare it, with the last program */

```
/* 27_tri.c */

#include <stdio.h>

int main()

{

    int i, j;

    clrscr();

    for( i = 1 ; i<= 5; i++ )
```

```
    {

         for( j = 1; j <= 5; j++ )

              printf("%4d", j);

         printf("\n");

    }

    return 0;

}
```

Few more examples of for loops:

```
 /* Infinite Loop */
for ( ; ; )
{
    printf("nHello");
}/* Print 1-5 numbers */
  for ( i=1; i<=5; )
  {
    printf("n%d", i++);
}
 /* Explicit Loop break*/
   for ( i=1; ;  )
   {
    printf("n%d", i++);
    if ( i> 5 ) break;
} int i = 1;
  for ( ;i<=5 ;  )
  {
```

```
    printf("n%d", i++);
}
```

while

An iterative statement to execute a statement block until the given condition is satisfied.

do.. while

This iterative statement executes statement block at the begin and then it checks the condition validity. If the condition is true it executes the statement block again and vice versa.

Syntax:

```
while( < condition > )

{

      <st. block>;

}
```

Syntax:

```
do

{

      <st. block>

} while(<condition>);
```

Eg.

The following example displays natural numbers from 1 to 10.

```
int main()

{

      int i=1;

      while( i<=10)

      {

            printf(â€œ\n%dâ€•,i);

            i++;

      }

      return 0;

}
```

```
int main()

{

      int i=1;

      do

      {

            printf(â€œ\n%dâ€•,i);
```

```
            i++;

      }while(i<=10);

      return 0;

}
```

It checks the condition first and executes the block next , So you should have an initial value for the condition ***It executes the block first and checks the condition next*** , You can determine the initial value in the st.block. ***More Examples***

/* 15. To print 1 to 5 numbers */
/* Note : It's a reference program */

```
/* 28_while.c */

#include <stdio.h>

int main()

{

     int i;

     i = 1;   /*  Initial value is 1 */

     while( i<= 10 )
      /* True i is less than or equal to 10 at first */
```

```
     {

          printf("\n%d", i);

          i ++;

     }

     return 0;

}
```

/* 16. To print 1 to 5 numbers */

```
/* 29_dowhile.c */

#include <stdio.h>

int main()

{

     int i;

     i = 1;   /*  Initial value is 1 */

     do

     {

          printf("\n%d", i);

          i ++;
```

```
    }while( i<=10 );
    /* True, i is less than or equal to 10 at Second */

    return 0;

}
```

/* 17. Demonstration of while */
/* Note : If the initial value is 100 what was the output?, Check it. */

```
/* 30_demow.c */

#include <stdio.h>

int main()

{

    int i;

    i = 1;

    while( i<= 10 )

    {

        printf("\n%d", i);

        i++;
```

```
    }

    return 0;

}
```

/* 18. Demonstration of do */
/* Note : If the initial value is 100 what was the output?, Check it. */

```
/* 31_demod.c */

#include <stdio.h>

int main()

{

    int i;

    i = 1;

    do

    {

        printf("\n%d", i);

        i++;

    }while( i<= 10 );

    return 0;
```

```
}
```

*06. Step-by-Step C/C++ --- C Programming - Functions*Functions

I. Introduction

Here is a program to print the address of a person twice, which is written in both methods using functions and without using functions. It will demonstrate the advantage of functions.

```
#include <stdio.h>

int main()

{

     printf("\nName of the Person");

     printf("\nStreet, Apartment//House No. ");

     printf("\nzip, City");

     printf("\nCountry");


     printf("\nName of the Person");

     printf("\nStreet, Apartment//House No. ");

     printf("\nzip, City");

     printf("\nCountry");



     return 0;

}
```

```
#include <stdio.h>

void address()
```

```
{

     printf("\nName of the Person");

     printf("\nStreet, Apartment//House No. ");

     printf("\nzip, City");

     printf("\nCountry");

}


int main()

{

     address();

     address();


     return 0;

}
```

II. Function Definition

A statement block, which has ability to accept values as arguments and return results to the calling program. So, A function is a self-contained block of statements that perform a specific task.

III. Types of functions

- Built-in functions/ Library Functions/ Pre-Defined functions
- User defined functions

IV. Library Functions

Library functions are designed by the manufacturer of the software, They were loaded in to the disk whenever the software is loaded.

The following functions are the example of the library functions.

1. Numeric Functions

Function	***Syntax***	***Eg.***	***Result***
Abs	Abs(n)	abs(-35)	35
ceil	ceil(n)	ceil(45.232)	46
floor	floor(n)	floor(45.232)	45
fmod	fmod(n,m)	fmod(5,2)	1
cos	cos(n)	cos(60)	0.5
sin	sin(n)	sin(60)	0.866
tan	tan(n)	tan(60)	1.732
sqrt	sqrt(n)	sqrt(25)	5
pow	pow(n,m)	pow(2,3)	8

2. String Functions

Functions	***Syntax***	***Eg.***
strlen	strlen(str)	strlen(œComputer•)
strcpy	strcpy(target,source)	strcpy(res,•Pass•)
strcat	strcat(target,source)	strcat(œmag•,•gic•)
strcmp	strcmp(str1,str2)	strcmp(œabc•,•Abc•)
strrev	strrev(target,scr)	fstrrev(res,•LIRIL•)

3. Character Test Functions

Function	***Description***
isalnum	is a letter or digit
isalpha	is a letter
isdigit	is a digit
iscntrl	is an ordinary control character
isascii	is a valid ASCII character
islower	is a lower character
isupper	is a upper character
isspace	is a space character
isxdigit	is hexa decimal character

There is a huge library of functions available, I have given you a tiny portion of it. For more Library Fuctions refer the Help Manual.

V. User-Defined Functions

The programs you have already seen perform divisions of labor. When you call ***gets, puts***, or ***strcmp***, you don(TM)t have to worry about how the innards of these functions work.

These and about 400 other functions are already defined and compiled for you in the Turbo C library. To use them, you need only include the appropriate header file in your program, œThe run-time library,• in the ***library reference*** to make sure you understand how to call the functions, and what value (if any) it returns.

But you(TM)ll need to write your own functions. To do so, you need to break your code into discrete sections (functions) that each perform a single, understandable task for your functions, you can call them throughout your program in the same way that you call ***C*** library functions.

Steps to implement a function

```
1. Declaration

2. Function Call
```

```
3. Definition
```

¢ Every function must be declared at the beginning of the program.
¢ Function definition contains the actual code of execution task.
¢ If a function is defined at the beginning of the program, there is no need of function declaration.
An example function to demonstrate the implementation

```
/* 32_egfun.c */

#include <stdio.h>



void address();

/* Declaration */
```

```
int main()
   {
       address();      /* Function Call */
       address();     /* Function Call */

     return 0;
   }

  void address()      /* Definition */
 {
     printf("nName of the Person");
```

```
    printf("nStreet, Apartment//House No. ");
    printf("nzip, City");
    printf("nCountry");
}
```

User defined functions can be divided in to 4 types based on how we are calling them.

1. Simple Functions
2. Function with Arguments
3. Function with Returns
4. Function with Recursion

1. Simple Functions

Performs a specific task only, no need of arguments as well as return values

Example of Simple Function

```
/* 33_line.c */

#include <stdio.h>

void line();

/* Declaration */
```

```
int main()
{
  line();       /* Function call */
  return 0;
}
```

```
void line()       /* Definition */
{
    int i;
    for(i =1;i<80; i++)
        putch(~*(TM));
}
```

2. Function with Arguments

A function, which accepts arguments, is known as function with arguments.

Eg.

```
/* 34_argu.c */

void line(char ch, int n)

int main()

{

     line("-", 50);

     line("*", 8);

     return 0;

}

void line(char ch, int n)

{
```

```
    int i;

    for( i = 1; i<=n; i++ )

         putch(ch);

}
```

3. Function with Return values

A function which can return values to the calling program is known as function with return values.

Eg.

```
/* 35_retu.c */

int abs(int n);

int main()

{

    int res;

    printf(â€œ%dâ€•, abs(-35))


    res = abs(-34);
```

```
/* Function Call*/
```

```
    printf(œ%d•, res);
    return 0;
}
void abs(int n)
{
    if( n < 0 )
    n = n * -1;
    return n;
}
```

4. Function with Recursion

If a statement within the body of a function call the same function is called ˜recursion(TM) . Sometimes called ˜circular definition(TM), recursion is thus the process of defining something in terms of itself.***Examples of Recursive of functions***
/* The following program demonstrates function call of itself */

```
int main( )

{

     printf(â€œ\nHelloâ€•);

     main( );   /* A function, which can call it self */

     return 0;

}
```

Don(TM)t run this program, it is still an explanation thus program is not valid logically.

The same output can be reached using another function:

```
void disp( );

int main( )
{
     disp( );
     return 0;
}

void disp( )
{
     printf(â€œ\nHelloâ€•);
     disp( );
}
```

The program must end at a certain point so the key of the recursion lies on soft interrupt, which can be defined using a conditional statement. Check the following example:

```
/* 36_recursion.c  */
```

```
int i = 1;       /* Declaring a global variable */

void disp( );

int main( )

{

     disp( );

     return 0;

}


void disp( )

{

     printf(â€œ\nHello  %d â€•, i);

     i ++;

     if( i < 10 )  /* if i value is less than 10 then call the function again */

          disp( );

}
```

Program to find the factorial of the given number:

```
/* 37_fact.c */

int factorial(int x);

void main

{

     int a, fact;

     printf("\nEnter any number   ");   scanf("%d", &a);

     fact = factorial(a);

     printf("\nFactorial is    = %d", fact);

}

int factorial(int x)

{

     int f = 1, i;

     for( i = x; i>=1; i--)

     f = f * i;

     return f;

}
```

To find the factorial of a given number using recursion

```
/* 38_fact.c */

int rec_fact(int x);

int main( )

{

     int a, fact;

     printf("\nEnter any number  "); canf("%d", &a);

     fact = rec_fact(a);

     printf("\nFactorial value is = %d", fact);

     return 0;

}

int f = 1;

int rec_fact(int x)

{

     if( x > 1)

          f = x * rec_fact(x-1);

     return f;

}
```

VI. Pointers and Functions

Parameter Passing by Reference
Call by value
Call by Reference

1. Parameter Passing by Reference

The pointer ca n be used in function declaration and this makes a complex function to be easily represented as well as accessed. The function definition makes use of pointers in it, in two ways
- Call by value
- Call by reference

The call by reference mechanism is fast compared to call by value mechanism because in call by reference, the address is passed and the manipulation with the addresses is faster than the ordinary variables. More ever, only one memory location is created for each of the actual parameter.

When a portion of the program, the actual arguments, calls a function and the values altered within the function will be returned to the calling portion of the program in the altered form. This is termed as ***call by reference*** or ***call by address***. The use of pointer as a function argument in this mechanism enables the data objects to be altered globally ie within the function as well as within the calling portion of the program. When a pointer is passed to the function, the address of the argument is passed to the functions and the contents of this address are accessed globally. The changes made to the formal parameters (parameters used in function) affect the original value of the actual parameters (parameters used in function call in the calling program).

Eg.

```
/* 39_func.c */

void func_c( int *x );

int main()
```

```
{

     int i = 100;

     int *a;

     a = &i;

     printf("\nThe value is %d", i);

     func_c(a);

     printf("\nThe value is %d", i);

     return 0;

}


void func_c( int *x )

{

     (*x) ++;

     printf("\nThe value in function is %d ", *x);

}
```

In the above program, these are totally three ˜printf(TM) statements, tow in the main() function and one in the function subprogram. Due the effect of first printf statement the value of i is printed as 100. Later function call is made and inside function, the value is altered in and is 1001 due to increment. The altered value is again returned to main() and is printed as 1001.

Hence the output is:
The value is 100
The value in function is 101
The value is 101

More about Function Calls

Having had the first tryst with pointers let us now get back to what we had originally set out to learn - the two types of functions calls: call by value and call by reference. Arguments can generally be passed to function in one of the two ways:
a. Sending the values of the arguments
b. Sending the addresses of the arguments

2. Call by Value

In the first method the ˜value(TM) of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function. With this method the changes made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function. The following programming illustrated the ***Call by Value***.

```
/* 40_callbyvalue.c */

void swap( int x, int y )

int main( )

{

     int a = 10, b = 20;

     swap( a ,b );

     printf("\n a = %d, b = %d ", a, b);
```

```
    return 0;

}

void swap( int x, int y )

{

    int t;

    t = x;

    x = y;

    y = t;

    printf("\nx = %d, y = %d", x, y);

}
```

The output of the above program would be:
X = 20 y = 10
A = 10 b = 20

Note that value of ***a*** and ***b*** remain unchanged after exchanging the value of ***x*** and ***y***.

3. Call by Reference

This time the addresses of actual arguments in the calling function are copied into formal arguments of the called function. This means that using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them. The following program illustrates this fact.

```
* 41_callbyref.c */

void swap( int *x, int *y )

int main()
{
     int a = 10, b = 20;
     swap( &a, &b);
     printf("\na = %d, b = %d", a, b);
     return 0;
}

void swap( int *x, int *y )
{
     int t;
     t = *x;     *x = *y;   *y = t;
}
```

The output of the above program would be:

A = 20, b = 10Note that this program manages to exchange the values of ***a*** and ***b*** using their addresses stored in ***x*** and ***y***. Usually in C programming we make a call by value. I.e. in general you cannot alter the actual arguments. But if desired, it can always be achieved through a call by reference.

Using call by Reference intelligently ***we can make a function, which can return more than one value at a time***, which is not possible ordinarily. This is shown in the program given below.

```
/* 42_callbyref.c */


void areaperi(int r, float *a, float *p)


int main()

{

     int radius;

     float area, perimeter;

     printf("\nEnter radius of a circle :"); scanf("%d", &radius);


     areaperi(radius, &area, &perimeter);


     printf("\nArea = %f ", area);

     printf("\nPerimeter = %f", perimeter);

     return 0;
```

```
}


void areaperi(int r, float *a, float *p)

{

     *a = 3.14 * r * r;

     *p = 2 * 3.14 * r;

}
```

And here is the output:

Enter radius of a circle 5
Are = 78.500000
Perimeter = 31.400000

Here, we are making a mixed call, in the sense, we are passing the value of ***radius*** but, address of ***area*** and ***perimeter***. And since we are passing the addresses, any change that make in values stored at address contained in the variables ***a*** and ***p***, would make the change effective in main. That is why when the control returns from the function ***areaperi( )*** we are able to output the values of ***area*** and ***perimeter***.

Thus, we have been able to return two values from a called function, and hence, have overcome the limitation of the ***return*** statement, which can return only one value from a function at a time.

VII. Local Vs Global Variables

According to the Scope of Identifiers Variables are declared as of types.

```
/* 42_globalid.c */
```

```
int i=4000;
/* Global variable declaration*/
```

```
int main()
{
    int a=10, b=20;     /* Local Variable */
    int i=100;    /* Local Variable */
    printf(œ%d   %d•, a, b);
    printf(œnLocal i  : %d•, i);  /* Accessing Local variable */
    printf(œnGlobal i : %d œ, ::i);  /* Accessing Global variable */
    return 0;
}
```

Note: Scope Resolution (::) Operator can be available in C++ only.

VIII. Storage Class Specifiers

Until this point of view we are already familiar with the declaration of variables. To fully define a variable one needs to mention not only its ˜type(TM) but also its ˜***Storage Class***(TM).

According to this section variables are not only have a ˜data type(TM), they also have a ˜Storage Class(TM).

Storage Classes are of 4 Types

1. Automatic Storage Class
2. Register Storage Class
3. Static Storage Class
4. External Storage Class

1. Automatic Storage Class

Keyword auto
Storage Memory
Default Value Null
Scope Local to the block in which the variable defined
Life Until the execution of its block

Eg:

```
/* 43_auto.c */

#include <stdio.h>

int main()

{

     auto int i, j;

     printf(â€œ%d  %dâ€•, i, u);

     return 0;

}
```

2. Register Storage Class

Keyword register
Storage CPU Registers
Default Value Null
Scope Local to the block in which the variable defined
Life Until the execution of its block

Eg:

```
/* 44_register.c */

#include <stdio.h>
```

```
int main( )

{

     register int i, j;

     for(i=1;i<=10;i++)

          printf(â€œ\n%dâ€•, i);

     return 0;

}
```

3. Static Storage Class

Keyword static
Storage Memory
Default Value Zero
Scope Local to the block in which the variable defined
Life Value of the variable persists between different function calls

Eg:

```
/* 45_static.c */

#include <stdio.h>

void add();

int main()
```

```
{

     add();

     add();

     add();

     return 0;

}

void add()

{

     static int i = 1;

     printf(â€œ%d\nâ€•,i++);

}
```

4. External Storage Class

Keyword extern
Storage Memory
Default Value Zero
Scope Global
Life As long as the program(TM)s execution doesn(TM)t come to an end

Eg:

```
/* 46_extern.c */

#include <stdio.h>

int i;

void add();     /* Extern variable */

int main( )

{

     extern j=10;     /* Extern variable */

     for(i=1;i<=10;i++)

          add();

     return 0;

}

void add( )

{

     j++;

     printf(â€œ%d   %d\nâ€•, i, j);

}
```

07. Step-by-Step C/C++ --- C Programming - Arrays

1. Introduction to arrays
    2. About Arrays
    3. Array Elements
    4. Passing Arrays to Functions
    5. Types of Arrays

- Single Dimensional Arrays

    1. Append element
        2. Insert element
        3. Delete element
        4. Replace element
        5. Search element
        6. Deletion of array
        7. Sorting of an array

- Multi Dimensional Arrays
    Matrix Operations using Multi Dimensional Arrays

1. Introduction to arrays

A variable can hold a constant value. Only a single constant value and it is not possible to hold more than one value in a variable.

The following example demonstrates the scope a variable.

```
int main()

{

     int sno;
```

```
    sno = 1001;

    sno = 1008;

    sno = 1005;

    printf(â€œ%dâ€•, sno);

    return 0;

}
```

Output:

1005

The above program is able to display only 1005, but not all the values (i,e. 1001, 1008, 1005).

Can we substitute the following program in place of the above program.

```
int main()

{

    int sno;

    sno 0 = 1001;

    sno 1 = 1008;

    sno 2 = 1005;
      printf(â€œ%dâ€•, sno);
```

```
    return 0;

}
```

Output:

Nothing,

The above program displays a list of errors, because of the approach is wrong.

Let(TM)s continue with the following program to get 0 errors program.

```
int main()

{

    int sno[3];

    sno[0] = 1001;

    sno[1] = 1008;

    sno[2] = 1005;

    printf(â€œ%dâ€•, sno[2] );

    return 0;

}
```

/* 3 values to be insert */

/* First location to insert 1001 */

/* Next location to insert 1008 */

/* and Next location to insert 1005 */

/* Prints the value of 2nd location */

Output:

Nothing

The above program displays a list of errors, because of the approach is wrong.

Depending on the above program, the variable ***sno*** can hold more than one student number. It(TM)s easy, by using multi-location technique, is also known as arrays.

2. About Arrays

Arrays contain a number of data items of the same type. This type can be a simple data type, a structure, or a class. The items in an array are called elements. Number accesses elements; this number is called an index. Elements can be initialized to specific values when the array is defined.

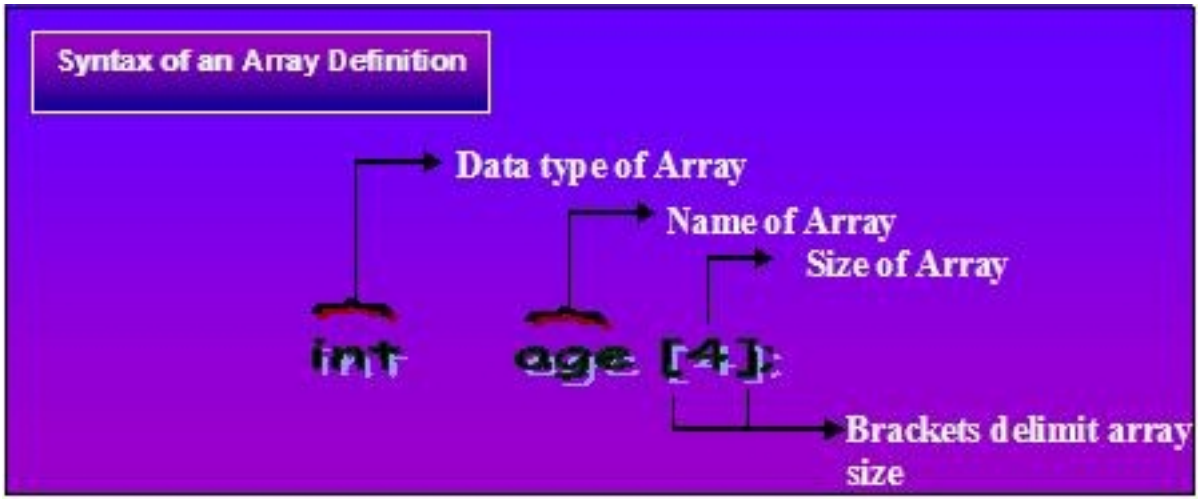
Arrays can have multiple dimensions.

A two-dimensional array is an array of array. The address of an array can be used as an argument to a function; the array itself is not copied. Arrays can be used as member data in classes. Care must be taken to prevent data from being placed in memory outside an array.

/* The following program reads 4 persons age and displays it */

```
/* 47_arrays.c */

#include <stdio.h>

int main()

{

     int age[4], i;
```

```
    for( i=0; i<4; i++)

    {

         printf(â€œEnter an age  â€œ); scanf(â€œ%dâ€•, &age[i]);

    }


    for(i=0; i<4; i++)

         printf(â€œ\nYou entered   %dâ€•, age[i]);


    return 0;

}
```

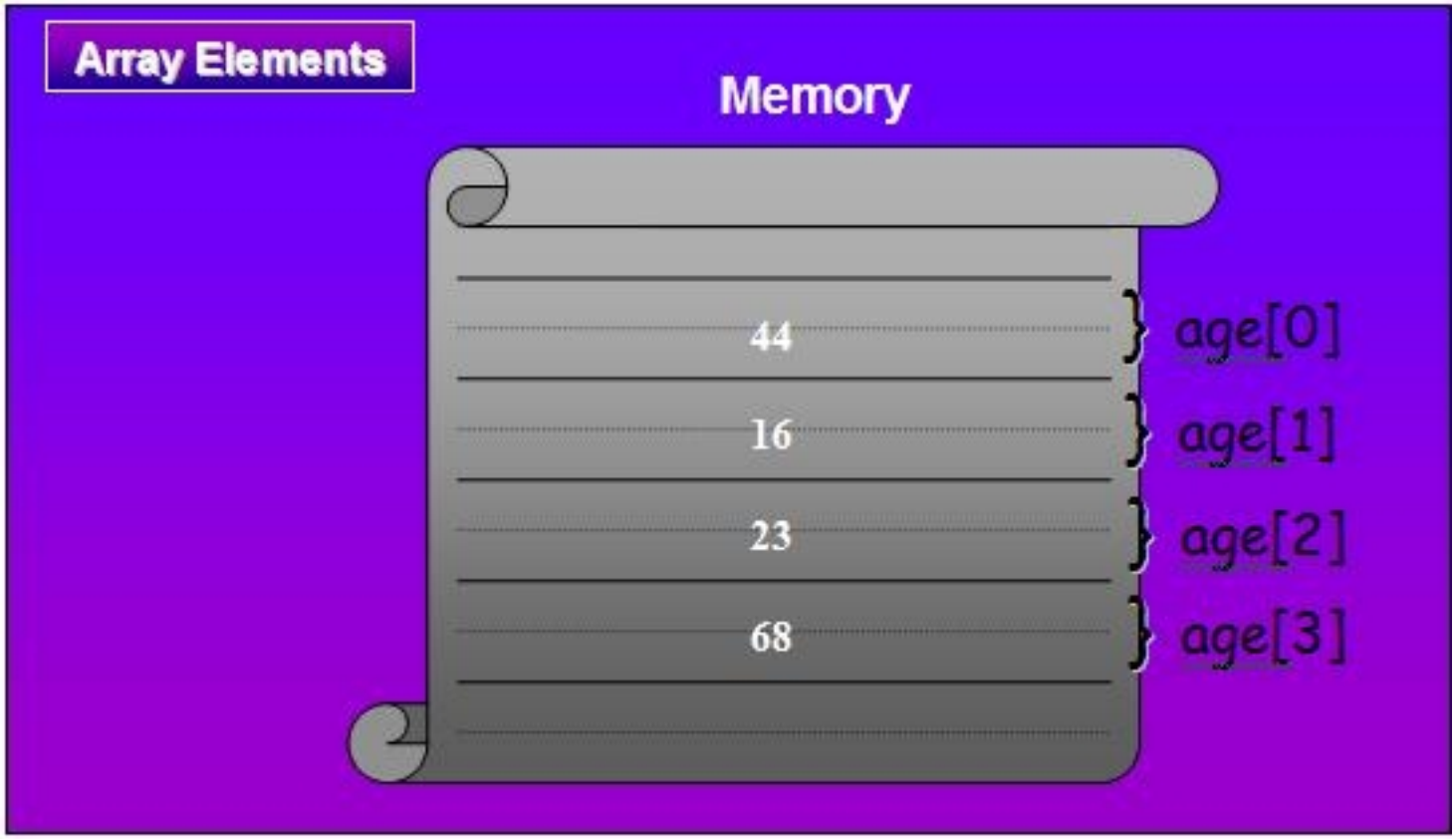


Like other variables in C, an array must be defined before it can be used to store information. And, like other definitions, an array definition specifies a variable type and a name.

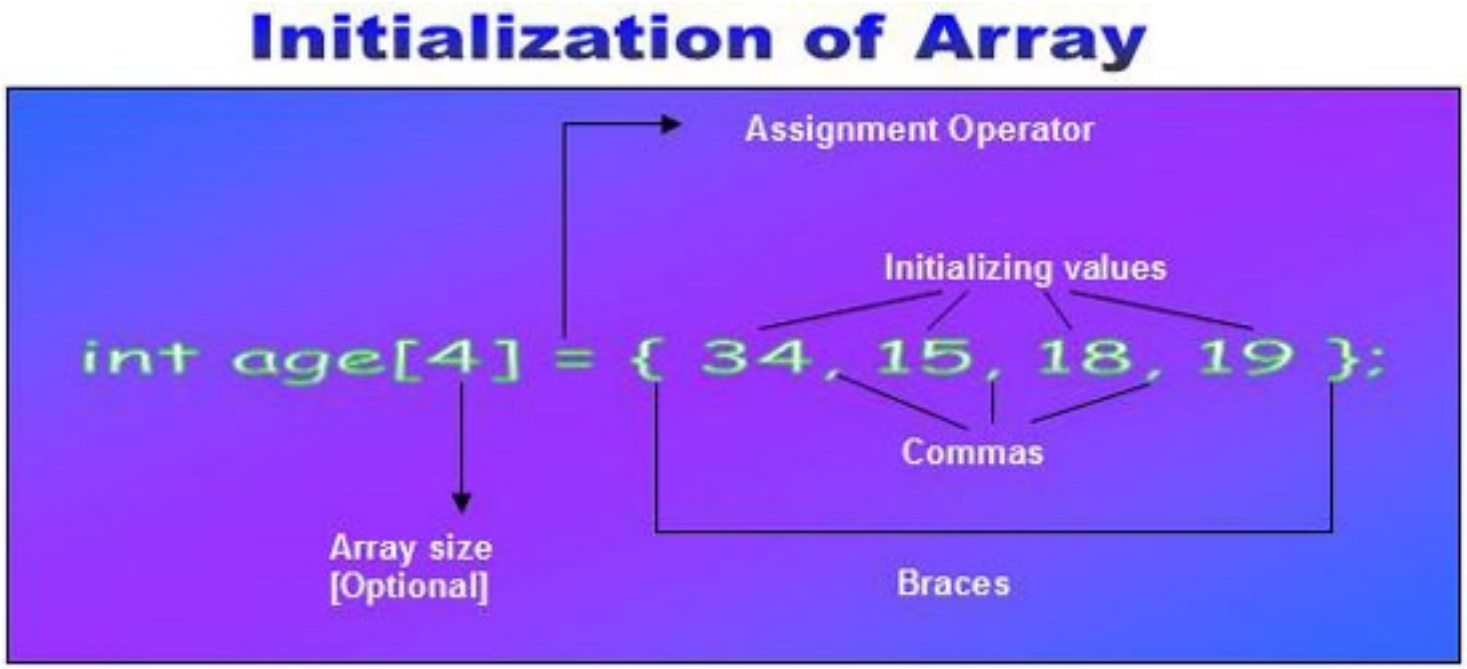
But it includes another feature: a ***size***. The size specifies how many data items the array will contain. It immediately follows the name, and is surrounded by square brackets.

3. Array Elements

The items in an array are called elements. Single Dimensional array accepts values to either row wise or column wise. It can store only one set of values. The first array element is 0, second is 1 and so on.



An array value can be initialized directly at design time.
Initialization of arrays is as follows..



4. Passing Arrays to Functions Arrays can be used as arguments to functions.
In a function declaration, the data type and sizes of the array represent array arguments.

```
void display(float [DISPLAY][MONTHS]);
```

When the function is called, only the name of the array is used as an argument.

```
display(sales);
```

Program to accept and print array of 10 elements

```
/* 48_ ele10.c */

#define  MAX 10

display(int a[MAX])

{

     int i;

     for(i = 0;i<MAX; i++)

          printf(â€œ\n%dâ€•,,a[i]);

}


int main()

{

     int x[MAX], i;

     for(i=0;i<MAX; i++)

          scanf(â€œ%dâ€•,&x[i]);


     display(x);
```

```
    return 0;

}
```

Returning array of values from functions is also possible but we must be clear with the concept of pointers. Please look in to the pointers topics for more info.

5. Classification of Arrays

Arrays are of two types.
1. Single dimensional Arrays
2. Multi Dimensional Arrays

1. Single Dimensional Arrays

A single dimensional array is a collection of elements in a row or a column fashion.
A single dimensional array can accept the following operations.
1. Append element
2. Insert element
3. Delete element
4. Replace element
5. Search element
6. Deletion of array
7. Sorting of an array

The following program is able to perform all the tasks described above.

```
/*  ARRAY FUNCTIONS  */

/* 49_sarray.c */
```

```
#define N 100

#define M 10

int i,j,r,c,r1,r2,c1,c2;
```

```
/*  Read  Array elements  */

int accept_values(int a[N])

{

     int n;

     printf("\nHow many values you wish to enter..?  ");

     scanf("%d",&n);

     printf("\nEnter the data elements..\n");

     for(i=0;i<n;i++)

          scanf("%d",&a[i]);

     return n;

}
```

```
/* Display array elements */

void display(int a[N],int n)

{

     printf("\n Array elements are...");

     for(i=0;i<n;i++)

          printf("\n%d",a[i]);

}
```

```
/* Delete array element */

int delete_cell(int a[N],int n)

{

     int pos;

     char ch;

     printf("\nEnter the position of element to be deleted: ");

     scanf("%d",&pos);

     for(i=pos-1;i<n;i++)
```

```
          a[i] = a[i+1];

     n--;

     printf("\n Do you wish to continue..(y/n)?");

     ch = getche();

     if(ch == 'y') delete_cell(a,n);

     return n;

}
```

```
/*  Insert array element */

int insert_cell(int a[N],int n)

{

     int pos, new;

     char ch;

     printf("\nEnter the element to be inserted: ");  scanf("%d",&new);

     printf("\nEnter the position of insertion: ");     scanf("%d",&pos);

     for(i=n;i>=pos;i--)

          a[i] = a[i-1];
```

```
    a[pos-1] = new;
    n++;

    printf("\n Do you wish to continue..(y/n)?");
    ch = getche();
    if(ch=='y') insert_cell(a,n);
    return n;
}
```

```
/*  To append element to an existing array */
int append_cell(int a[N],int n)
{
    int pos, new;
    char ch;
    printf("\nEnter the element to be appended: ");
    scanf("%d",&new);
```

```
     a[n] = new;

     n++;

     printf("\n Do you wish to continue..(y/n)?");

     ch = getche();

     if(ch=='y')append_cell(a,n);

     return n;

}
```

```
/* Sorting a list of elements of an array in descending order */

void sort_list_descend(int a[N],int n)

{

     int temp;

     for(i=0;i<n;i++)

          for(j=i;j<n;j++)

               if(a[i] < a[j])

               {
```

```
                    temp = a[i];

                    a[i] = a[j];

                    a[j] = temp;

               }

     printf("\nThe sorted elements in the descending order are...");

}
```

```
/* Sorting a list of elements of an array in ascending order */

void sort_list_ascend(int a[N],int n)

{

     int temp;

     for(i=0;i<n;i++)

          for(j=i;j<n;j++)

               if(a[i] > a[j])

               {

                    temp = a[i];
```

```
                    a[i] = a[j];

                    a[j] = temp;

               }

     printf("\nThe sorted elements in the ascending order is..");

}
```

```
/* To find the smallest and biggest of an existing array */

void small_big(int a[N],int n)

{

     int temp;

     for(i=0;i<n;i++)

          for(j=i;j<n;j++)

               if(a[i] < a[j])

               {

                    temp = a[i];

                    a[i] = a[j];
```

```
                    a[j] = temp;

               }

     printf("\nThe Smallest element is : %d",a[n-1]);

     printf("\nThe Biggest element is : %d",a[0]);

}
```

```
/* Search for an element in an array */

void search(int a[N],int n)

{

     int target, temp=0;

     printf("\n Enter element to be searched: ");

     scanf("%d",&target);

     for(i=0;i<n;i++)

          if(a[i] == target)

          {

               printf("\nFound at position no. %d",i+1);
```

```
                temp = 1;

          }

     if(temp == 0)

          printf("\n Not found");

}
```

```
/* Main program */
/*  To demonstrate simple array operations  */
#include<stdio.h>

#include<conio.h>
#define m 100

int main()

{

     int a[m],n; char ch;

     clrscr();

     n = accept_values(a);

     do

     {
```

```
        printf("\n 1 - Append_cell");

        printf("\n 2 - Delete_cell");

        printf("\n 3 - Insert_cell");

        printf("\n 4 - Sort_list_descend");

        printf("\n 5 - Sort_list_ascend");

        printf("\n 6 - Small_big");

        printf("\n 7 - Search");

        printf("\n 8 - Remove_list");

        printf("\n 9 - Exit");

        printf("\n Enter your choice: ");
        ch = getche();

        printf("\n");

        switch(ch)

        {

             case '1': n = append_cell(a,n);  break;

             case '2': n = delete_cell(a,n);  break;

             case '3': n = insert_cell(a,n);  break;
```

```
            case '4': sort_list_descend(a,n); break;

            case '5': sort_list_ascend(a,n); break;

            case '6': small_big(a,n);  break;

            case '7': search(a,n);   break;

            case '8': n = 0;    break;

            case '9': printf("\nThis will terminate your program."); break;

        }

        display(a,n);

        printf("\nDo you wish to run again..(y/n)?");

        ch = getche();

    }

    while(ch!='9');


    return 0;

}
```

2. Double Dimensional Arrays

A double dimensional array is a collection of elements in row and column fashion.
A Multi dimensional array can accept the following operations.

A multi dimensional array is commonly used in the areas of matrices to understand whole tasks in an easiest approach.

MATRIX FUNCTIONS

```
/* 50_menumat.c */

#define N 100

#define M 10

int i, j, r, c, r1, r2, c1, c2;
```

```
/*  Read the values for a MATRIX  */

void read_matrix(int A[M][M])

{

     printf("\nHow many rows?  ");

     scanf("%d",&r);

     printf("\nHow many columns?  ");

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

```
     for(i=0;i<r;i++)
          for(j=0;j<c;j++)
               scanf("%d",&A[i][j]);
}
```

```
/*  Write the values of a MATRIX  */
void disp_matrix(int A[M][M])
{
     for(i=0;i<r;i++)
     {
          for(j=0;j<c;j++)
          printf("%5d",A[i][j]);
          printf("\n");
     }
}
```

```
/* To find the TRANSPOSE for a MATRIX of ANY ORDER  */

void tra_matrix_1(int T[M][M],int A[M][M])

{

     printf("\nTranspose of A is\n");

     for(i=0;i<c1;i++)

     {

          for(j=0;j<r1;j++)

          {

               T[i][j] = A[j][i];

               printf("%5d",T[i][j]);

          }

          printf("\n");

     }

}
```

```
/* To ADD two MATRICES
( possible,only if they are of EQUAL ORDER )  */
```

```
void add_matrix(int C[M][M],int A[M][M],int B[M][M])

{

     for(i=0;i<r;i++)

          for(j=0;j<c;j++)

               C[i][j] = A[i][j] + B[i][j];

     printf("\nSum of A and B is");

}
```

```
/*  To MULTIPLY MATRICES of ANY ORDER
( provided they follow the  MATRIX MULTIPLICATION RULE )   */

void mul_matrix_1(int C[M][M],int A[M][M],int B[M][M])

{

     int k;

     printf("\nProduct of A and B is..\n");

     printf("\nMatrix C\n");

     for(i=0;i<r1;i++)

     {
```

```
          for(j=0;j<c2;j++)

          {

               C[i][j]=0;

               for(k=0;k<r2;k++)

                    C[i][j] = C[i][j]+(A[i][k]*B[k][j]);

               printf("%5d",C[i][j]);

          }

           printf("\n");

      }

}
```

```
/*  To SUBTRACT two MATRICES
( possible,only if they are of EQUAL ORDER )  */

void sub_matrix(int C[M][M],int A[M][M],int B[M][M])

{

    for(i=0;i<r;i++)

        for(j=0;j<c;j++)
```

```
            C[i][j] = A[i][j] - B[i][j];

    printf("\nDifference of A and B is");

}
```

```
/*  A MENU driven program to perform MATRIX operations  */
#include <stdio.h>

#include <conio.h>

int main()

{

    int A[M][M],B[M][M],C[M][M],T[M][M];    char ch;

    clrscr( );

    printf("\nEnter matrix A elements..\n"); read_matrix(A); r1=r; c1=c;

    printf("\n   Matrix A\n"); disp_matrix(A);

    printf("\nEnter matrix B elements..\n"); read_matrix(B); r2=r; c2=c;

    printf("\n   Matrix B\n");disp_matrix(B);

    do {

        printf("\n1:Addition");
```

```
        printf("\n2:Subtraction");

        printf("\n3:Multiplication");

        printf("\n4:Transpose");

        printf("\n5:Exit");

        printf("\nEnter your choice..");    ch = getche();

        switch(ch)

        {

            case '1':

                    if(r1==r2 && c1==c2)

                    {

                        add_matrix(C,A,B);printf("\nMatrix C\n");

                        disp_matrix(C);

                    }

                    else

                    {

                        printf("\nYour entered values of r1,r2 && c1,c2 are not equal,");

                        printf("\nhence I cannot do this Matrix Addition.");
```

```
            printf("\nPlease enter the correct matrices.");
        } break;

    case '2':
        if(r1==r2 && c1==c2)
        {
            sub_matrix(C,A,B);printf("\nMatrix C\n");
            disp_matrix(C);
        }
        else
        {
            printf("\nYour entered values of r1,r2 && c1,c2 are not equal,");
            printf("\nhence I cannot do this Matrix Subtraction.");
            printf("\nPlease enter the correct matrices.");
        } break;

    case '3':
```

```
                if(c1==r2)
                    mul_matrix_1(C,A,B);
                else
                {
                    printf("\nColumns(c1) of Matrix A are NOT EQUAL TO");
                    printf(" Rows(r2) of Matrix B.");
                    printf("\nHence I cannot do this Matrix Multiplication.");
                    printf("\nPlease enter matrices such that c1 == r2.");
                } break;

        case '4':   printf("\nOrder of Matrix A is %d x %d",r1,c1);
                    tra_matrix_1(T,A);
                    printf("\nOrder of A transpose is %d x %d",c1,r1);  break;

        case '5': printf("\nThis will terminate your program."); break;
    }
    printf("\nDo you wish to run again...[y/n]?");   ch=getche();
}while(ch!='5');
```

```
    return 0;

}
```

08. Step-by-Step C/C++ --- C Programming - Strings

Strings

- Introduction
- Characteristics of a strings
- Operations on Strings
    1. Definition of Strings
    2. Initialization of Strings
    3. Reading and printing of Strings
    4. Reading Embedded Blanks
    5. Length of a String
    6. Strings and Functions
    7. Array of Strings

Introduction

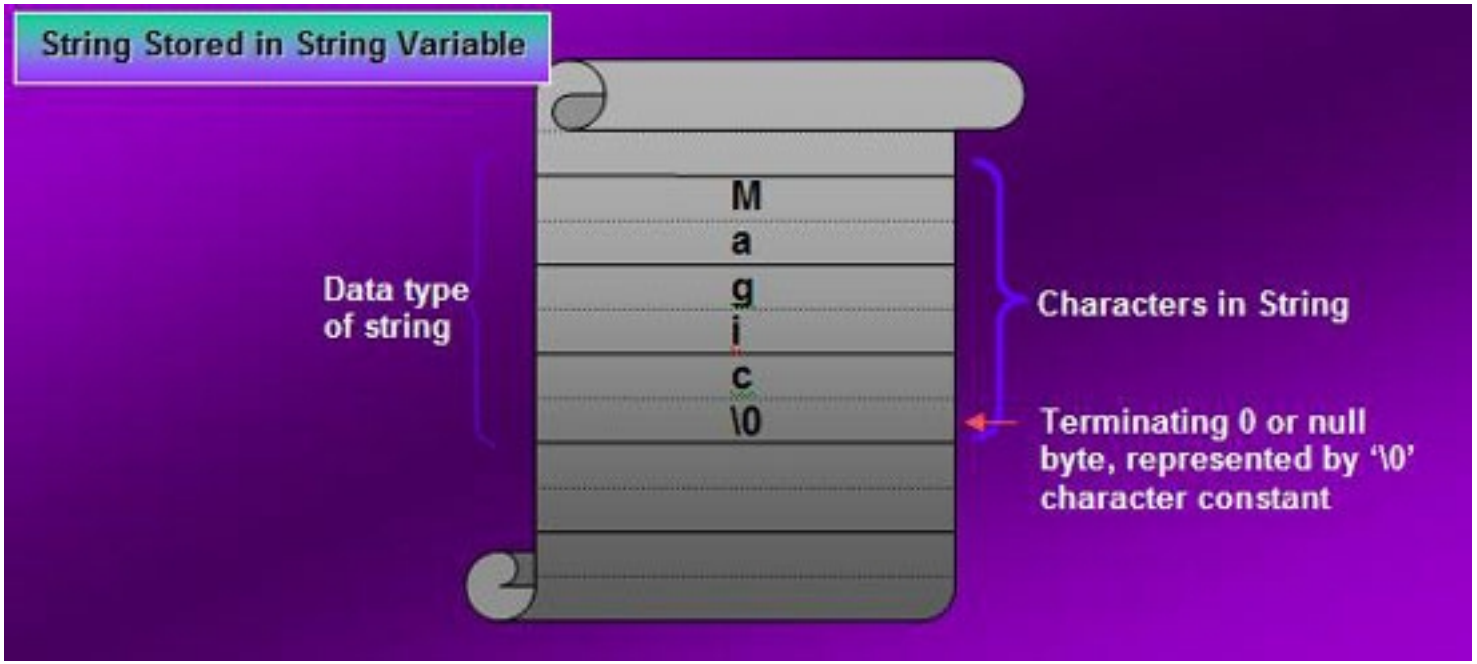
Arrays are used to examine strings, generally strings are of array type variables.
A string is a collection of characters including space where as word is a collection of characters excluding space. Every string variable must be terminating with ˜ (TM) null character and the index value is starts with 0.

Every string has the following characteristics:

1. It must be a collection of characters (i.e. characters, numbers, and special characters).
2. Every string must be ends with a NULL character (i.e. ˜ (TM))
3. A unique positive number called index identifies each character of a string.
4. Index value must be starts with 0.
5. Random access on characters in a string is possible.
6. A string must be declared with its fixed size like arrays.

For Example consider the following example:

```
char str = " magic";
```



A variety of string library functions are used to manipulate strings. An array of strings is an array of arrays of type char.

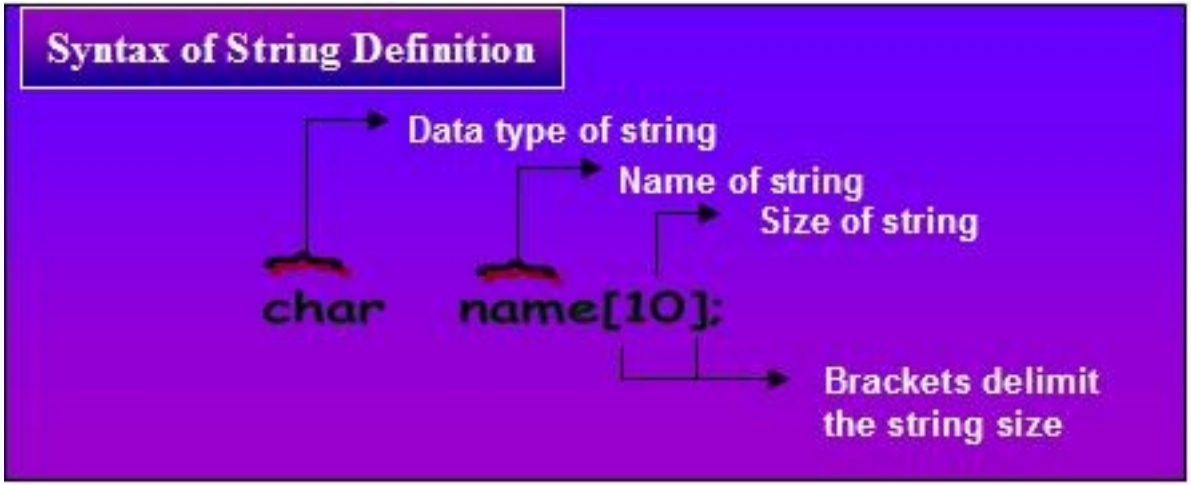
Operations on Strings

We can perform much better operations than using Library string functions.
Strings can accept the following operations.

1. Definition of Strings
2. Initialization of Strings
3. Reading and printing of Strings
4. Reading Embedded Blanks
5. Length of a String
6. Strings and Functions
7. Array of Strings

1. Definition of a String

Every variable must be declared at the beginning of the program.
Definition of string variable is as follows.



2. Initialization of Strings

Strings can be initialized in the following methods.

1. Direct Assignment

```
char name[10] = "Ashitha";
```

Assigns "Ashitha" to name rest of the place left blank. ***2. Direct Assignment without Size***

```
char name[] = "Ashitha";
```

Assigns "Ashitha" to name and fix it(TM)s width up to the size of Constant.

3. Design time Assignment

```
char name[10];
```

```
strcpy(name, "Ashitha");
```

Using Strings functions it is possible.
But C never support the assignment like :
name = "Ashitha"; ***4. Runtime Assignment***

```
char name[10];

scanf("%s", name);
```

It accepts and assigns constant value to variable at runtime.

3. Reading and Printing Strings

C provides various types of string functions to read and print a string constant. Listed below.
Input Statements
getch
getche
getchar
gets
scanf***Output Statements***
putch

putchar
puts
printf
/* Program to accept and display a string */

```
/* 51_strings.c */

#include <stdio.h>

int main()
```

```
{

     char str[20];

     scanf("%s", str);

     printf("%s" str);

     return 0;

}
```

/* Program to accept and display a string with a prompt */

```
/* 52_strings.c */

#include <stdio.h>

int main( )

{

     char str[20];

     printf("Enter a string :");  scanf("%s", str);

     printf("\nYou entered  :    %s", str);

     return 0;

}
```

4. Reading embedded blanks

scanf Accepts string, thus it will read strings consisting of a single word, but anything typed after a space is thrown away.

Eg. Enter String : Law is a bottomless pit.
You entered : Law

To read text containing blanks we use another function, gets().
/*read string with embedded blanks */

```
/* 53_gets.c */

const int MAX = 80;

int main()

{

     char str[MAX];

     print("Enter a string :");   gets(str);

     printf("You entered  :");  puts(str);

     return 0;

}
```

5. Length of String

Every string has its fixed length depending on its constant.
The following program demonstrates, How to find the length of the string
/* To find the length of a given string */

```
/* 53_length.c */

#include <stdio.h>

int main()

{

     int i=0;

     char str[50];

     printf("Enter a string  "); gets(str);

     while(str[i] != '\0')   i++;

     printf("Length is  %d", i);

     return 0;

}
```

6. Strings and Functions

A function is a self-contained block of statements that perform a specific task. The best way to organize strings.
The following are the example of string organization using functions.
/* Program to find the length of a string */

```
/* 54_len.c */

#include <stdio.h>

int len_str(char s[]);
```

```
int main()

{

     int l;     char str[50];

     printf("Enter a string  "); gets(str);

     l = len_str(str);

     printf("\nLength of string   : %d", l);

}
```

```
int len_str(char s[])

{

     int l=0;

     while(s[l] != '\0')   l++;
```

```
     return l;

}
```

/* Program to accept and print a string */

```
/* 55_str.c */

#include <stdio.h>

void disp_str(char s[]);
```

```
int main()

{

     char str[50];

     printf("Enter a string  "); gets(str);

     disp_str(str);

     return 0;

}
```

```
void disp_str(char s[])
```

```
{

     int i=0;

     while( s[i] != '\0' ) putch(s[i++]);

}
```

7. Array of Strings

Arrays are used to examine strings, generally strings are of array type variables. So, we can access array of strings.
The following examples illustrate, How Array of Strings organized.
/* Program to display an array of strings */

```
/* 56_display.c  */

#include <stdio.h>

void main()

{

     char week[7][] = { "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday" };

     int i;

     for( i = 0; i<7; i++) puts(week[i]);

}
```

/* Program to accept and display an array of strings */

```
/* 57_strings.c  */

#include <stdio.h>

void main()

{

     char names[7][10];   int i;

     for( i = 0; i<7; i++) gets(names[i]);

     for( i = 0; i<7; i++) puts(names[i]);

}
```

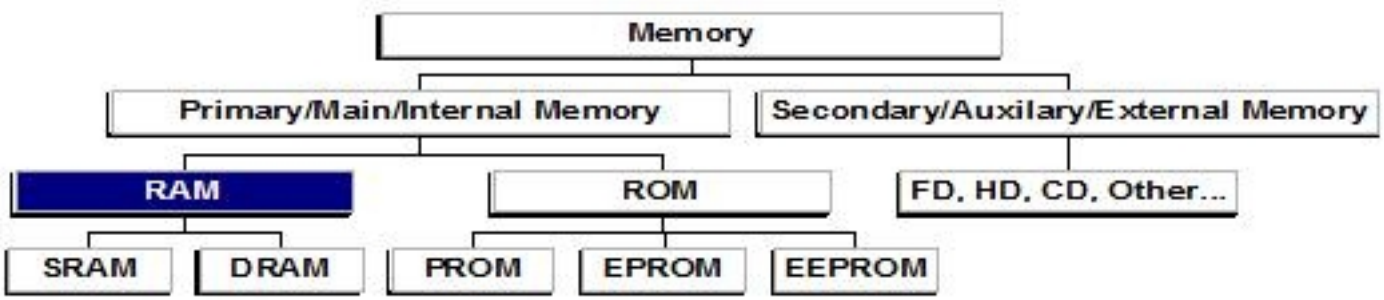
09. Step-by-Step C/C++ --- C Programming - Pointers Pointers

1. About Memory
2. Addressing Scheme
3. How to find the address of a Variable
4. Pointers
5. Pointer Arithmetic
6. Pointers and Arrays
7. Pointers and Strings
8. Glossary

1. About Memory

Computer has the feature to store data, and manipulate them. Storage of data requires a storage device, which was comfortable to store and retrieve data

quickly and accurately with out confusion. Commonly Computer has to compromise with two storage methods.



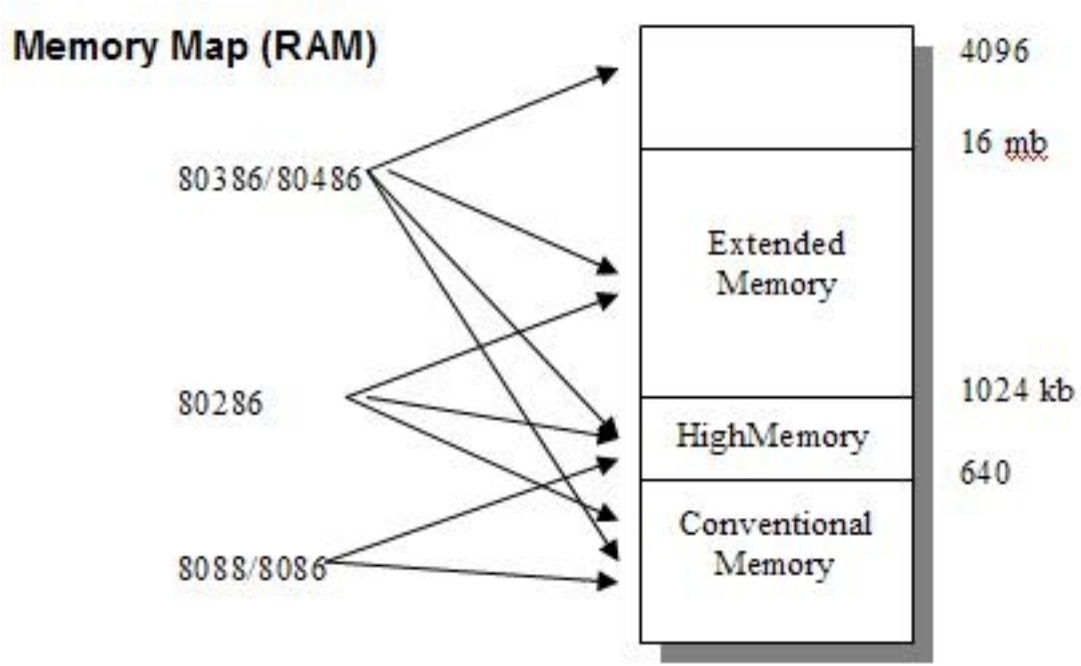
SRAM Static Random Access Memory
DRAM Dynamic Random Access Memory
EEPROM Electrically Erasable Programmable Read Only Memory

Memory chips can store data, instructions and intermediate & final results. The memory is organized into bytes, each byte capable of storing one character of information. Each byte of memory has an address or location number, which uniquely identifies it. The size of memory is measured either in kilobytes (KB), megabytes (Mb), gigabytes or terabytes (TB).

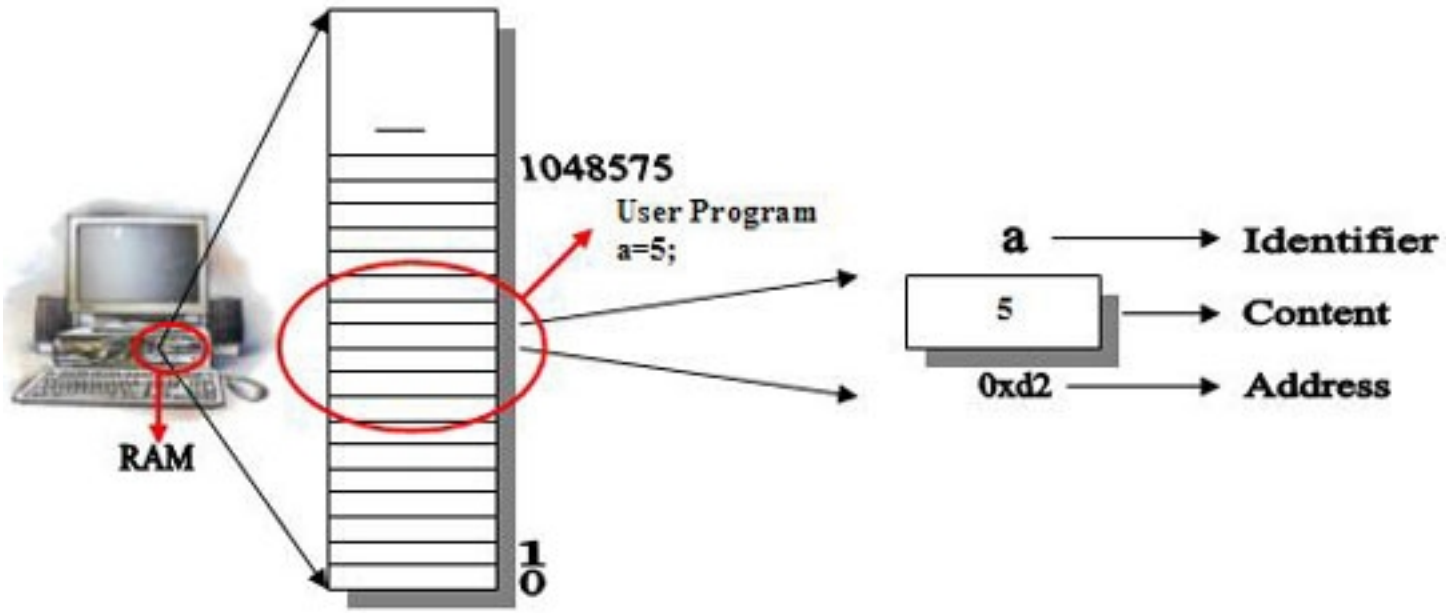
RAM:

Memory device is a storage location to store information.

The Vital Computer resource memory is allocated to each of the variable after their declaration in the C-Program. The type of the variable decides the number of bytes of memory to be allocated to each variable.



2. Addressing Scheme



The above picture tells you the following information.

1. RAM is a temporary memory and a part of the computer.
2. It can hold the value of program.
3. Every byte in RAM has identified with a unique positive number called address.
4. Addresses are as numbers, just as they are for houses on a street.
5. The number starts at 0 and go up from there 1-2-3 and so on.
6. If we have 640 KB of memory the highest address is 655, 359, for 1mb of memory it is 1,048,575.
7. Our program, when it is loaded into memory, occupies a certain range of these addresses.
8. That means that every variable and every function in out program starts at a particular address.

3. How to find the address of a Variable

In the last section point 8 tells each and every variable/function starts at a particular address. Addresses are unique positive numbers in the hexa decimal format.

Finding address of a variable is a simple task through the operator ***&*** (***address of***).

& (address of) - It can tell you the address of variable / function in the current program.

The following program demonistrates to find the address of a variable ˜a(TM)

```
/* 58_address.c */

#include <stdio.h>

int main()

{

     int a = 10;



     printf("\n Value of A is          :  %d", a);

     printf("\n Address of A is      :  %d", &a);

     return 0;

}
```

Replace the above marked format values with the following format to get absolute hexadecimal address value.
0x%x
Ex.

```
printf("\nAddress of A is     : 0x%x", &a);
```

Program to find the address of function ˜disp()(TM)

```
/* 59_address.c */

#include <stdio.h>

void disp()

{

     printf("\nHello");

     printf("\nHow are");

     printf("\nYou");

}
```

```

int main()

{

     disp();

     printf("\nAddress of   disp()      : 0x%x", &disp);

     return 0;

}
```

4. Pointers

According to the last section we know how to find and display the address of a variable/function.
This time we learn about how to store the address of a variable/function in another variable.

Note: Variables can hold constant values.
Try with the following:

```
int a, b;

a = 5;                /*  Valid   */

b = &a;            /* In valid */
```

Again Try with the following:

```
int a,  *b;

a = 5;             /* Valid   */

b = &a;            /* Valid */
```

b = &a; correct! Yes, variables (General variables) are unable to hold addresses. But variables proceeded with ˜*(TM) (Pointer Variables) are able to hold both constant values as well as address of another variables/functions.

Pointer: Variable that holds address values.
Variables (General)
General variable performs only one operation to hold constant values

Pointer variables (Variables preceded with ˜*(TM))
Pointer variables can perform two operations to hold constant values as well as address values of other variables/functions

Reference to / Pointer to / Content at address (*)

```
int *ptr;
```

To the uninitiated this is a rather bizarre syntax. The asterisk means pointer to. Thus the statement defines the variable ***ptr*** as a ***pointer to*** int. This is another way of saying that the variable can hold the address of integer variables.

If we called it, type pointer we could write declaration like.

```
pointer   ptr;           /*  invalid */
```

The problem is that the compiler need to know what kind of variable the pointer points to.

Declaration of a pointer variable

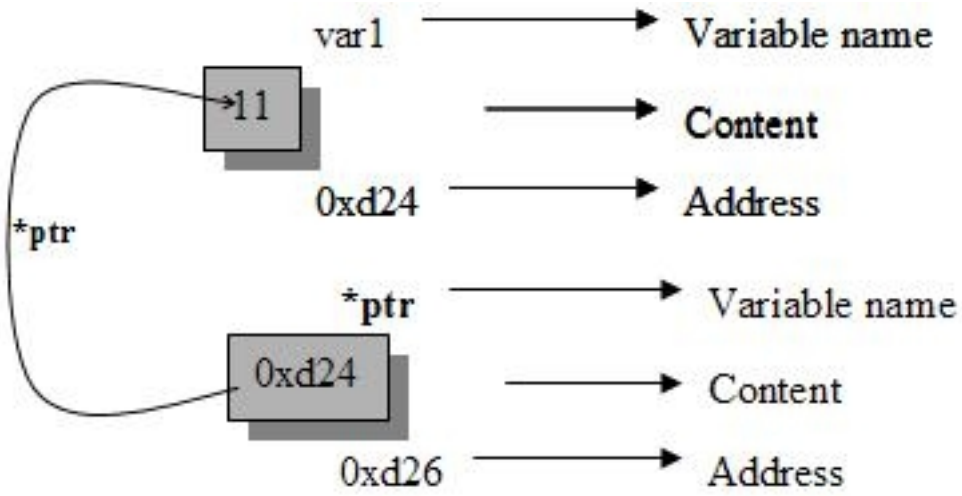
```
char  *cptr;            /* Pointer to character  */

int  *iptr;             /* Pointer to int  */

float *fptr;             /*  Pointer to float */

struct emp  *e;             /* Pointer to abstracted data emp e */
```

Accessing the variable Pointed to:Here is the special way to access the values of a variable using its address instead of its name.

```
/* 60_addr.c */

#include <stdio.h>
```

```
int main()

{

     int var1  = 11;
/*  variable var1 = 11  */
```

```
    int *ptr;            /*  Variable ptr as pointer to */

    ptr = &var;          /* Hold the address of var to ptr  */

    printf("Value of var1 is     %d", *ptr);          /* Pointer to the address of  var1  */
    return 0;
}
```



If the statement is ***printf (œ%d•, ptr);*** then it displays the value of ptr means the address of ***var1***, but the above statement can display the content of the address, which was stored in variable ***ptr***.

Program to demonstrate the use of Address_Of and Pointer_To

```
/* 61_ptrdemo.c */

#include <stdio.h>

int main()

{

     int a = 10, *p;
/*  Integer a and pointer p  */
```

p = &a; /* Assign address of a to p */

```
  printf("nValue of A      :    %d", a);          /* Content of a */
    printf("nAddress of A   :  0x%x", &a);          /* Address of a */
    printf("nValue of P  :  0x%x",  p);          /* Content of p */
    printf("nAddress of P  :  0x%x", &p);          /* Address of p */
    printf("nContent at address of a   : %d", *p);          /* Pointer to &a */
    return 0;
}
```

5. Pointer Arithmetic

All the variables can support arithmetic operations, as well as we can perform arithmetic operation on pointers also. C/C++ language can supports 4 Arithmetic operations on Pointers namely.

Operation	***Symbol***
Addition	+
Subtraction	-
Incrementation	++
Decrementation	--

Note: The main characteristic of pointer arithmetic is that the above operators in bytes with reference to its variable type.

```
/* 62_ptr.c */

/* Demonstration of pointer arithmetic */

#include <stdio.h>

int main()
```

```
{

     int a, *p;

     a = 100;

     p = &a;

     (*p)++;     /* Increment pointer to (content at address) by 1 */

     printf("%d", *p);

     return 0;

}
```

Output

101

Demonstration of Pointer arithmetic, Increment the address value

```
/* 63_ptr.c */

/* Increment the address value by 1 */

#include <stdio.h>

int main()

{

     int a, *p;
```

```
    a = 100;

    p = &a;

    *p++; /* Increment the address value in p by 1 */

    printf("%d", *p);

    return 0;

}
```

Output

Unexpected output

The above program illustrates the arithmetic operators with respective of both value and address incrementation. ***p*** is a pointer variable and ***a*** is assigned with 100, as well as ***p*** is assigned with the address of ***a***.

Now ****p++*** effects incrementing or actually skipping the memory by ***2*** bytes to get new address and their its content.

If it(TM)s ***(*p)++***, then that the content pointed by ***p*** is 100 is incremented, resulting 101.

6. Pointers and Arrays

In C/C++ language the data types pointers and arrays resembles with each other. The array element references as well as the pointer variable, both are used to hold the address of data elements in memory.

char name[20];

Or

char *name ;

char months[12][10];
Or
char **months;

There is a close association between pointers and arrays. Here is a review on arrays.

```
/* 64_ptrarr.c */

#include <stdio.h>

int main()

{

     int i, a[5] = { 56, 43, 78, 98, 12 };

     for( i = 0, i < 5; i++)

          printf("\n%d", a[i]);

     return 0;

}
```

There is a possibility to access array elements using pointer notation.
Find the output of the following program.

```
/* 65_ptrarr.c */

#include <stdio.h>
```

```
int main()

{

     int i, a[5] = { 56, 43, 78, 98, 12 };

     for( i = 0, i < 5; i++)

          printf("\n%d", *(a+ i) );

     return 0;

}
```

Follow the next program:

```
/* 66_ptrarr.c */

#include <stdio.h>

int main()

{

     int i, a[ ] = { 56, 43, 78, 98, 12 }, *p;

     p = a;

     for( i = 0, i < 5; i++)
```

```
          printf("\n%d", *(p+ i) );

     return 0;

}
```

Here is an easiest approach to print the elements of the given array (size not required).

```
/* 67_ptrarr.c */

#include <stdio.h>

int main()

{

     int i, a[ ] = { 56, 43, 78, 98, 12 }, *p;

     p = a;

     while (*p)             /* or for(int i = 0; i<5; i++ )  */

          printf("\n%d", *p++);

     return 0;

}
```

7. Pointers and Strings

A string is a collection of characters including spaces. This time we discuss about how to handle strings using pointers. No more discussions to make confusion. Here is the simple task to verify both pointer and array of strings.

There is a subtle difference between strings & pointers follow the program.

```
/* 68_ptrstr.c */

#include <stdio.h>

int main()

{

     char str1[ ] = "You would like to explore C.";

     char *str2 = "You would like to explore C.";

     puts(str1);

     puts(str2);

     str1++;  /* Invalid expression  */

     str2++;  /* Valid expression */

     puts(str2); /* prints  ou would like toâ€¦â€¦  */

     return 0;

}
```

Strings as Function Arguments

A pointer variable is more flexible than array variables, Here is the program to demonstrate & displays a string with pointer notation.

```
/* 69_ptrarr.c */

#include <stdio.h>

void disp(char *p);

int main()

{

     char str[ ] = "Hello!!..Hello!!..  Pointers can handle it?";

     disp(str);

     return 0;

}
```

```
void disp(char *p)

{

     while(*p)

     printf("%c", *p++);
```

```
}
```

Array of pointers to strings

There is a disadvantage to store an array of strings, in that the sub arrays that hold the string must all be the same length. So that space is wasted when strings are shorter than the sub arrays.
Here is the solution:

```
/* 70_strings.c */

#include <stdio.h>

int main()

{

     char *weeks[7 ] = { "Sunday", "Monday", "Tuesday", "Wednesday",
"Thursday", "Friday", "Saturday" };

     int i;

     for( i = 0; i<7; i++)

          puts(weeks[ i ] );

     return 0;

}
```

When strings are not part of an array, C/C++ places them contiguously in memory, So there is no wasted spaces.

/* An example program to hold an array of pointers of type ˜int(TM) */

```
/* 71_ptrarr.c */

#include <stdio.h>

int main()

{

     int *arr[4];   /* Array of int pointers */

     int i = 31, j = 5, k = 19, l = 71, m;



     arr[0] = &i;

     arr[1] = &j;

     arr[2] = &k;

     arr[3] = &l;



     for(m = 0; m <= 3; m++)

          printf("\n%d", *(arr[m]) );
```

```
    return 0;

}
```

A look on Library Functions

We are already familiar with standard string functions. They have string arguments that are specified using pointer notation,
If we are clear with pointers & strings concept, we are able to write our own string functions.
Here is an example program to copy string.

/ Copies one string to another with pointers */*

```
/* 72_strcpy1.c */

#include <stdio.h>

void strcpy1(char * dest, char *src);


int main()

{

    char *str1 = "How can I learn more about C/C++ !!!";

    char *str2;

    strcpy1(str2, str1);

    puts(str2);
```

```
     return 0;

}
```

```
void strcpy1(char * dest, char *src)

{

     while(*src)

          *dest++ = *src++;

     *dest = â€˜\0â€™;

}
```

8. Glossary

Address

A value that points to a location in memory. A pointer contains the address or location of a value, as opposed to the value itself.

Array

An array is a collection of data items of the same type.

Contiguous

A storage characteristic that specifies that the values are stored in consecutive locations either in memory or on disk.

Function

A series of instructions to perform a specific task, which can be combined with other functions to create a program.

Memory

Descriptive of a device or medium that can accept data, holds them, and deliver them on demand at a later time. Synonymous with storage.

Pointer

Contains the address or memory location of a value, as opposed to the value itself.

RAM

(Random Access Memory) 1. A storage device structured so that the time required retrieving data is not significantly affected by the physical location of the data. 2. The ***primary storage section*** of a personal computer.

String

An array capable of storing zero or more characters. In C. a string is declared as a character array with the NULL ( ) character appended to specify the end of the string.

Variable

A name associated with a location in memory whose value can change during program execution.

10. Step-by-Step C/C++ --- C Programming - Structure Structures

1. Introduction

```
int a[4] = { 3, 4, 5, 6 };                  /* Valid  expression */

int a[4] = { 3, 4.23, 5, 6 };                /* Invalid expression */

int a[4] = { 3, "Siglov", 5,3}               /* Invalid expression */
```

Why the last two expressions are invalid? An array can store values of same type. Must be the same type. Where as a structure can hold more than one type of data according to its definition.

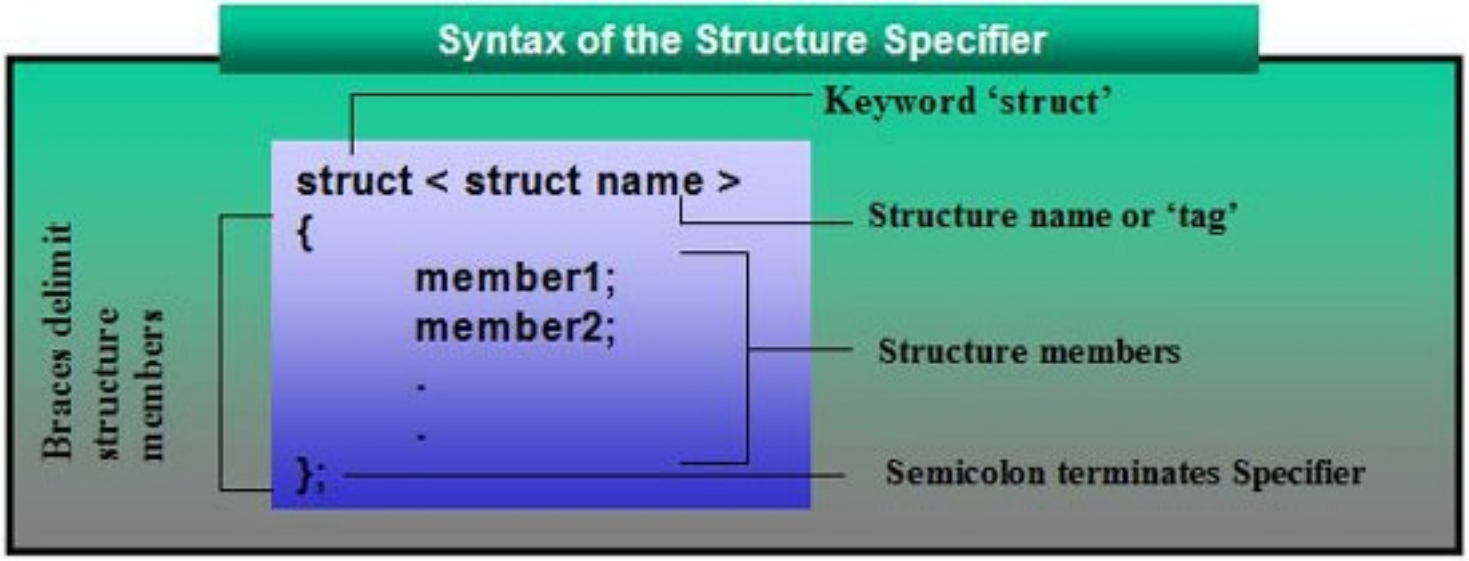
¢ A group of one or more variables of different data types organized together under a single name is called a structure or

¢ A collection of heterogeneous (dissimilar) types of data grouped together under a single name is called a structure or

¢ A structure is a collection of simple variables. The variable in a structure can be of different types. The data items in a structure are called the members of the structures.

2. Declaration of a structure

When a structure is defined the entire group is referenced through the structure name. The individual components present in the structure are called as the structure members and these can be accessed and processed separately.



Eg:

```
struct date
{

     int day;

     int month;

     int year;
```

```
};
```

```
struct student
{

    int sno;

    char name[20];

    int marks;

    float avg;

};
```

3. Defining a Structure Variable

Defining a structure variable is the same as that for defining a built-in data type such as ***int***.

```
int a;                  /*  valid */
  date d;               /* valid (But in C++ only ) */
  struct date d;        /* valid in both C and C++  */
```

4. Initializing a Structure variable

The members of the structure can be initialized like other variables. This can be done at the time of declaration or at the design time.

1. Initialization at Declaration:

```
struct ddate
{
    int day;
    int month;
    int year;
} d = { 27, 10, 2000 };
```

2. Initialization at Definition:

```
struct ddate d = { 27, 10, 2000 };
```

- *Initialization at design time:*

```
ddate d;
    d.day = 27;
    d.month = 10;
    d.year = 2000;
```

4. Initialization at run time:

```
scanf("%d%d%d", &d.day,  &d.month, &d.year);
```

Eg:

```
/* Write a  program to accept and print the details of an employee */

/* 73_struct.c  */

#include  <stdio.h>

struct emp

{
```

```
            int eno;

            char name[20];

            float sal;

};
int main()

{

            struct emp e;



            printf("Enter Employee number    :"); scanf("%d", &e.eno);

            printf("Enter Employee name        :"); scanf("%s", e.name);

            printf("Enter Employee salary        :"); scanf("%d", &e.sal);

            printf("\n\nEmployee Details are as followsâ€¦.\n");

            printf("%d    %s      %d", e.eno, e.name, e.sal);

            return 0;


}
```

5. Direct assignment of structures

Direct assignment of more than one variable is made possible using structures.

```
struct emp a, b = {1001, "Vimal", 6700.00 };

a = b;    /* Valid  */

printf("%d  %s  %d" , a.eno, a.name, a.sal );
```

Output:

1001 Vimal 6700.00

6. Calculation of structure size

Every data type in C/C++ has a specified size, i.e int has 2 bytes of size, float has 4 bytes of size and so on. Here is the way to find the size of a structure variable.

sizeof :- This function is used to find the size of a given variable.

```
printf("%d", sizeof(int));                          /* 2  */

printf("%d", sizeof(float));                       /* 4  */

printf("%d", sizeof(struct emp));             /* Displays the size of the emp  structure */
```

7. Nested Structures

Structure with in structures in known as nested structures. For accessing nested structure members we must apply the dot operator twice in calling structure members.

Eg:

```
/*  program to demonstrate nested structure with employee structure */

/*  74_nested.c */

#include  <stdio.h>

struct  emp

{

            int eno;

            char name[10];

            float sal;

           struct                                                  /*  Nested Structure  */

            {

                        street  char[10];

                        city  char[10];

            } addr;

};
int  main()

{
```

```
            struct emp e;

            printf("Enter emp_no, emp_name,  emp_sal, street, city ");

            scanf("%d%s%d%s%s", &e.eno,  e.name, &e.sal, e.addr.street, e.addr.city  );

            printf("\n\nEmployee Details are as  follows   â€¦.\n");

            printf("%d%s%d%s%s", e.eno, e.name,  e.sal, e.addr.street, e.addr.city );

            return 0;

}
```

8. Array of Structures

We can create an array of structures. The array will have individual structures as its elements.

```
/* Write a program to accept and print the details  of an employee */

/* 75_array.c  */

#include <stdio.h>

struct emp

{

            int  eno;

            char  name[20];
```

```
            float  sal;

};

int main()

{

            struct  emp e
[10]
;
    int  i;
    for(i  = 0; i<10; i++)
     {
            printf("Enter  Employee number    :"); scanf("%d",  &e[i].eno);
            printf("Enter  Employee name      :"); scanf("%s", e[i].name);
             printf("Enter Employee salary      :"); scanf("%d", &e[i].sal);
    }
     printf("nnEmployee Details are as follows¦.n");
     for(i = 0; i<10; i++)
              printf("%d    %s      %d", e[i].eno,  e[i].name, e[i].sal);
     return 0;
}
```

Nothing is new in the above program. Entire program is same as simple structured program except the marked data.

9. Arrays with in Structures

There may be a situation to utilize arrays with in structures. How to achieve arrays with in structures. Here is the approach with simple program.

```
/* Program to accept and print a student  information  */
```

```
/* 76_array.c */

#include <stdio.h>

struct stud

{

            int  sno;

            char  name[10];

            int  marks[5];                              /*  Array with in structure */

};

int main()

{

            struct  stud s;

            int  i;

            printf("Enter  Student number  "); scanf("%d",  &s.sno);

            printf("Enter Student name          "); scanf("%d", s.name);

            for(  i = 0; i<3; i++)

            {
```

```
                        printf("Enter  student marks  "); scanf("%d",  &s.marks[i]);

            }

            printf("\n\nStudent  Records is as followsâ€¦.\n");

            printf("%d  %s   %d  %d  %d", s.sno, s.name, s.marks[0], s.marks[1],  s.marks[2] );

            return  0;

}
```

10. Passing Structures to Functions

It is possible to send entire structures to functions as arguments in the function call. The structure variable is treated as any ordinary variable.

```
/* Program to pass a structure variable to function  */

/* 77_funct.c */

#include <stdio.h>

struct emp

{

            int  eno;

            char  name[10];

            float  sal;
```

```
};

void display(struct emp temp);

int main()

{

            struct  emp e;

            display(e);

            return  0;

}

void display(struct emp temp)

{

            printf("%d  %s   %d", temp.eno, temp.name, temp.sal );

}
```

11. Returning Structures from functions

We can return structures from functions. Yes structures can be returned from functions just as variables of any other type.

```
/* Returning structure object from a function */
```

```
/* 78_funct.c */

struct emprec

{

            int eno;

            char name[10];

};

struct emprec read();

void write(struct emprec t);

int main()

{

            struct emprec e;

            e = read();

            write(e);

            return 0;

}

void write(struct emprec t)

{
```

```
            printf("\n\n%d  %s", t.eno, t.name);

}

struct emprec read()

{

            struct emprec t;

            printf("Enter Employee  number   :"); scanf("%d", &t.eno);

            printf("Enter Employee  name         :"); scanf("%s", t.name);

            return t;

}
```

12. Pointer to Structure

Till now we have seen that the members of a structure can be of data types like int, char, float or even structure. C/C++ language also permits to declare a pointer variable as a member to a structure. Pointer variables can be used to store the address of a structure variable also. A pointer can be declared as if it points to a structure data type.

```
/* Program to demonstrate the process of Pointer to structure */

/* 79_pointer.c */

#include <stdio.h>
```

```
struct employee

{

            int eno;

            char name[10];

};

struct employee *emp;

int main()

{


emp  = (struct employee * )malloc(sizeof(emp));
```

```
    printf("Enter Employee  Details ..");
    scanf("%d%s%",  &emp->eno, emp->name);
    printf("nn%d   %s", emp->eno,  emp->name);
    return 0;
}
```

The marked data is essential to implement pointer to structure.
The following statement is optional, but better to utilize to organize better memory management.
emp = (struct employee *)malloc(sizeof(emp));

13. Structures Containing Pointers

A pointer variable can also be used as a member in the structure.
The following program contains pointer members contained by a pointer variable of structure.

```
/* program to demonstrate the use of structures containing Pointers */

/* 80_pointers.c  */

#include <stdio.h>

struct

{

            int *a;

            int *b;

} *temp;

int main()

{

            int x, y;

            x = 20; y = 50;

            rk -> a = &x;

            rk -> b = &y;



            printf("%d %d ",  *temp->a, *temp->b );
```

```
            return 0;

}
```

output:

20 50

14. Self Referential Structures

Structures can have members, which are of the type the same structure itself in which they are included. This is possible with pointers and the phenomenon is called as self-referential structures.

```
struct emp

{

            int eno;

            char name[10];

            struct emp *e;

};
```

Self-referential structures can be used mainly in arranging data, sorting, searching elements, insertion, deletion of elements and so on.

This way of approach leads to Data structures (i.e., Linked Lists, Stacks, Queues, Trees and Graphs).

11. Step-by-Step C/C++ --- C Programming - Unions*Unions*

1. Introduction

```
/* 81_union.c */

#include <stdio.h>

struct s_emp

{

            int eno;

            char name[20];

            float sal;

};

union u_emp

{
```

```
            int eno;

            char name[20];

            float sal;

};

int main()

{

            struct s_emp se;

            union u_emp ue;

            printf("\nSize of Employee  structure    :  %d", sizeof(se));

            printf("\nSize of Employee Union       :  %d", sizeof(ue));

            return 0;

}
```

Output: Size of Employee Structure : 26
Size of Employee Union : 20

2. About Union

When a large number of variables are requested to use in a program. They were occupies a large amount of memory. Unions provide an easiest way to save memory by using replacement technique. It uses same memory location for all type of variables.

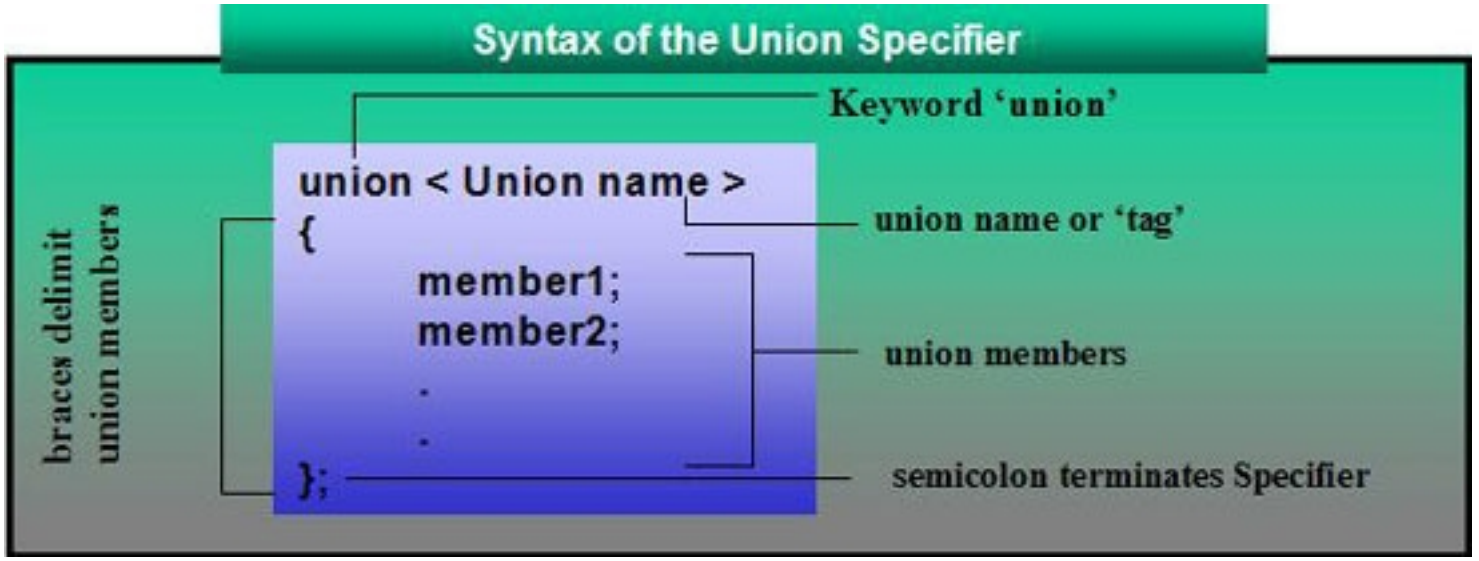
A union is a data type in C, which allows the overlay of more than one variable in the same memory area.

Characteristics of Unions:

1. Union stores values of different types in a single location in memory.
2. A union may contain one of many different types of values but only one is stored at a time.
3. The union only holds a value for one data type. If a new assignment is made the previous value has no validity.
4. Any number of union members can be present. But union type variable takes the largest memory occupied by its members.

3. Declaration of a Union

Union is a data type through which objects of different types and sizes can be stored at different times. Definition of a Union is same as a Structure. The only change in the declaration is the substitution of the keyword union for the keyword struct.



Eg:

```
union ddate
 {
   int day;
   int month;
   int year;
```

```
};union student
  {
    int sno;
    char name[20];
    int marks;
    float avg;
};
```

4. Defining a Union Variable

Defining a Union variable is the same as structure and that for defining a built-in data type such as int.

```
int a;     /*  Valid */

union date d;  /* Valid in both C and C++  */
```

Calculation of Union size

Every data type in C/C++ has a specified size, i.e int has 2 bytes of size, float has 4 bytes of size and so on. Here is the way to find the size of a Union variable.
sizeof :- This function is used to find the size of a given variable.

```
printf("%d", sizeof(int));  /* 2  */

printf("%d", sizeof(float));  /* 4  */

printf("%d", sizeof(union emp));   /* Displays the size of the emp union */
```

5. Difference between Structures and Unions

Here is the difference between Structures and Unions

Structure

Union

1. It can hold different types (variables) in a single location.

1. It can hold different types (variables) in different locations.

2. It may contain more than one type (variable) but only one is stored at a time.

2. It may contain more than one type (variable) all are stored in memory at a time.

3. Any number of union members can be present. But union type variable takes the largest memory occupied by its member.

3. It requires memory of the size of all its members.

4. On its process only one member can be accessed at any given time.

4. On its process all the members can be access at any time.

5. The scope of union is the function and the scope of its members is also same as the union itself. (They can be accessed directly in the program).

5. The scope of Structure is the function only. Structure members are unable to access directly in the program.

6. Operations on Unions

A union is also similar to structure it can perform all the operations like structures. Operations on Union are listed below.

- ¢ A union variable can be assigned to another union variable.
- ¢ A union Variable can be passed to a function as a parameter
- ¢ The address of the union variable can be extracted by using the address-of operator (&).
- ¢ A function can accept and return a union or a pointer to a union.

```
/* 82_union.c */

#include  <stdio.h>

union u_emp

{

      int eno;

      char name[20];

      float sal;

};

int main()

{

      union u_emp ue;
```

```
      printf("Enter Employee Number  : "); scanf("%d",  &ue.eno);

      printf("Enter Employee Name    : "); scanf("%s", ue.name);

      printf("Enter Employee Salary  : "); scanf("%f",  &ue.sal);

      printf("\n\nEmployee Details are as  follows...\n");

      printf("%d %s %f ", ue.eno,  ue.name, ue.sal);

      return 0;

}
```

What is the output?

Only ue.sal is correct. What about rest of variables.

At any instant only one of the union variables will have a meaningful value. Only that member, who is last written, can be read. At this point, other variables will contain garbage. It is the responsibility of the programmer to keep track of the active variable (i.e. variable which was last accessed).

Here is the best way to accept and display records of an employee.

```
/* 83_emp.c */

#include  <stdio.h>

union u_emp

{

      int eno;
```

```
      char name[20];

      float sal;

};

int main()

{

      union u_emp ue;

      printf("\nEnter Employee Number  : "); scanf("%d",  &ue.eno);

      printf("\n%d", ue.eno);

      printf("\nEnter Employee Name  : "); scanf("%s", ue.name);

      printf("\n%s", ue.name);

      printf("\nEnter Employee Salary  : "); scanf("%f",  &ue.sal);

      printf("\n%f",ue.sal);

      return 0;

}
```

7. Scope of a Union

The scope of union is different than structure. A structure variable can be accessed by the its functions only. Where as a union and its members can be accessed by its function.

```
/* 84_scope.c */
 #include <stdio.h>
 int main()
 {
     union
     {
         int i;
         char c;
         float f;
     };
     i = 10; c = ˜a(TM); f = 4.5;    /* Union members */
     printf("The value of c is  : %c", c);
     return 0;
 }
```

12. Step-by-Step CorC++ --- C Programming - FilesFile Handling

Introduction

Let(TM)s find the output of the following program.

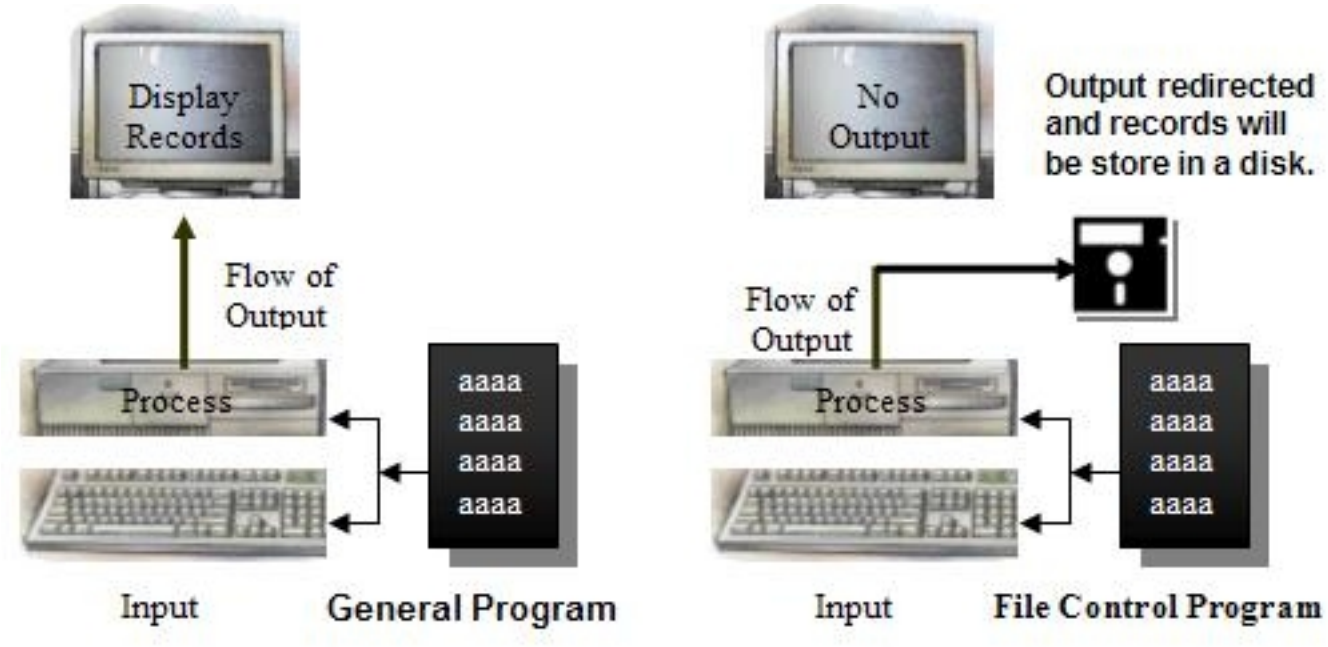
```
#include <stdio.h>
int main()
{
        int sno, sub1, sub2, sub3;
        char name[20];

        printf("Enter a student record  sno, name, sub1, sub2, sub3 respectively\n");
        scanf("%d %s %d %d %d\n", &sno, name, &sub1, &sub2, &sub3);

        printf("\nStudent record is as follows.....");
        printf("%d%s%d%d%d\n", sno, name, sub1, sub2, sub3);
        return 0;
}
```

Yes, it accepts a record of student information and displays it.

Here is the same program, but included statements with a few modifications.

```
#include <stdio.h>
int main()
{
        int sno, sub1, sub2, sub3;
        char name[20];

        FILE *fp = fopen("stud.dat", "a+");

        printf("Enter a student record  sno, name, sub1, sub2, sub3 respectively\n");
        scanf("%d %s %d %d %d\n", &sno, name, &sub1, &sub2, &sub3);

        printf("\nStudent record is as follows.....");
        fprintf(fp,"%d%s%d%d%d\n", sno, name, sub1, sub2, sub3);
        return 0;
}
```

Above two programs are same, but the second program contains a highlighted statement (***FILE *fp = fopen(æstud.dat•, æa+•);***) and a few modifications like ***˜f***printf(TM), ***˜fp(TM).*** Only few modifications included. These modifications affect data to transfers from console to diskette in the file ***stud.dat***. This process is known as ***file control/file management/file organization***.This is an easiest way to transfer the output from monitor to file using file control statement.



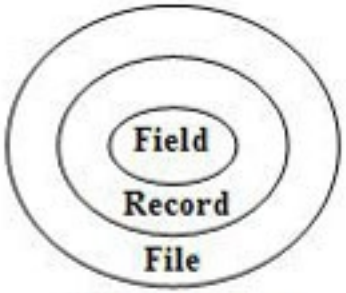
Actually file processing involved with a lot of operations as well as methods to implement. Here is the actual process to handle files.

File Handling

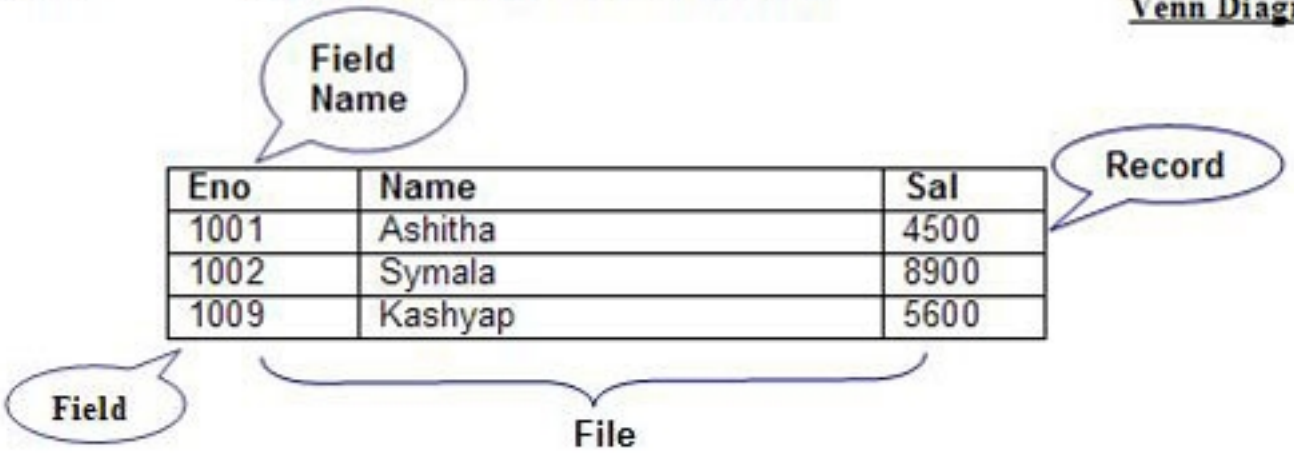
Generally every program has to present the resulted values on the screen (1st program illustrates this). But those values are removed from the memory whenever the program is terminated. If we want to keep records permanently, save them in a file. Every file has a few operations, here is a few;

- Create file
- Open file
- Close file

File A file is a collection of records
Record Record is a collection of fields
Field A Field is an individual data element.



Venn Diagram



Eno	Name	Sal
1001	Ashitha	4500
1002	Symala	8900
1009	Kashyap	5600

Here is the list of file processing statements.

File Operations	Command
Open an existing file	fopen
Close file	close

Record Operations	Command
Add record	fprintf
Retrieve record from the begin	fscanf
Insert record	fwrite
Retrieve record from pointer	fread

Record Navigation	Command
Places the pointer to the beginning of the file	rewind
Move the pointer from one record to another	fseek
To find the record pointer position	ftell
Is end of file	feof, eof

Miscellaneous I/O Operations	Command
Read/write character on file	fgetc / fputc, fgetchar / fputchar
Read/write string on file	fgets / fputs

File Operations

fopen

Opens the stream filename in the mode mode & if succeeded, Returns a pointer to the newly open stream; or Null other wise.

Syntax

FILE *fopen(const char *filename, const char *mode);

E.g.

```
FILE *fp = fopen("stud.dat",    "r");   /*  Read from file  */
     FILE *fp = fopen("emp.dat", "w");  /*     Write to file  */
   FILE *fp = fopen("emp.dat", "a+");          /*     Read and Write on file */
```

Mode:

The mode string used in calls to fopen, is one of the following values:

Mode	Description
r	Open for reading only
w	Create for writing (If a file by that name already exists, it will be overwritten).
a	Append; open for writing at end of file, or create for writing if the file does not exist.
r+	Open an existing file for update (reading and writing)
w+	Create a new file for update (reading and writing). If a file by that name already exists, it will be overwritten.
a+	Open for append; open for update at the end of the file, or create if the file does not exist.

To specify that a given file is being opened or created in text mode, append "***t***" to the string (***rt, w+t,*** etc.).

To specify binary mode, append "***b***" to the string (***wb, a+b,*** etc.).

fclose

Closes the file pointed to by fp & returns 0 on success, EOF is returned in case of error

Syntax

Int fclose(FILE *fp);

e.g.

Fclose(fp); fclose(stud); fcloseall();

fprintf

Sends formatted output to a stream. Uses the same format specifiers as printf, but sends output to the specified stream. Returns the number of bytes output or EOF in case of error.

Syntax

Fprintf(fptr, œControl String•, list);

E.g

Fprintf(fp, œ%d %s %d %d %d•, sno, name, sub1, sub2, sub3);
fprintf(emp, œ%d %s %d•, eno, name, sal);

fscanf

This function is used to read a formatted data from a specified file.

Syntax:

Fscanf(fptr, œControl String•, list);

E.g

Fscanf(fp, œ%d %s %d %d %d•, &sno, name, &sub1, &sub2, &sub3);
fscanff(emp, œ%d %s %d•, &eno, name, &sal);

fwrite

Fwrite appends a specified number of equal-sized data items to an output file.

Syntax:

Size_t fwrite(const void *ptr, size_t size, size_t n, FILE*stream);

Argument What It Is/Does

Ptr Pointer to any object; the data written begins at ptr
Size Length of each item of data
N Number of data items to be appended
stream Specifies output file
The total number of bytes written is (***n * size***)

fread

Fread retrieves a specified number of equal-sized data items from an input file.

Syntax

Size_t fread(void *ptr, size_t size, size_t n, FILE*stream);

Argument What It Is/Does

Ptr Pointer to any object; the data written begins at ptr

size Length of each item of data

n Number of data items to be appended

stream Specifies output file

The total number of bytes written is (***n * size***)

rewind

Repositions file pointer to stream's beginning

Syntax

Void rewind(FILE *stream);

E.g. fewind(fp);

Rewind(stream) is equivalent to fseek(stream, 0L, SEEK_SET)

except that rewind clears the end-of-file and error indicators, while fseek only clears the end-of-file indicator. After rewind, the next operation on an update file can be either input or output.

fseek

The file pointer for the stream is positioned at offset number of bytes calculated from the position specified by whence. Offset may be zero, negative, or positive. The defined symbols SEEK_CUR, SEEK_SET & SEEK_END are used as whence specifiers to indicate current position. BOF & EOF respectively. Returns 0 if successful or nonzero on failure.

Syntax

Int fssek(FILE *stream, long offset, int whence);

ftell

Returns the current file pointer position on success or Negative value on error.

Syntax

Long ftell(FILE *stream);

feof

It is a macro to return nonzero if end-of-file has been reached on the stream.

Syntax

Int feof(FILE *stream);

eof

Checks whether the position marker in the file given by its handle is at the end-of-file. If yes, returns 0, 1 is returned if position marker is NOT at eof & an error is indicated by setting of errno & return value of -1.

Syntax

Int eof(int handle);

fgets / fputs

The function fgets/fputs gets/puts a string(of size n bytes) on the file pointed to by stream and returns end-of-file on error.

Syntax

Char *fgets(char *s, int n, FILE *stream);

fgetc/fputc

Syntax

Reads/writes a character from a stream.

Int fgetc/fputc(FILE *stream);

fgetchar/ fputchar

These are equivalent to the above fgetc/fputc.

Write a program to read a student data and store it in a data file.

```
/* Program to  create a student data file */
```

```
/* 85_write.c */

#include  <stdio.h>

#include  <ctype.h>

#include  <conio.h>

int main()

{

      int sno, sub1, sub2, sub3;

      char name[10],ch;

      FILE *fp = fopen("stud.dat",  "w");

      do{

            clrscr();

            printf("Enter Student  number        ");  scanf("%d", &sno);

            printf("Enter Student name          "); scanf("%s", name);

            printf("Enter 3 Subjects Marks      ");

            scanf("%d%d%d",  &sub1, &sub2, &sub3);

            fprintf(fp, "%d %s %d %d  %d\n", sno, name, sub1, sub2, sub3);

            printf("\n\nDo you want to  cont... (y/n)"); ch = getche();
```

```
      }while(toupper(ch) != 'N');

      fclose(fp);

      return 0;

}
```

Write a program to retrieve data from a student data file.

```
/* Program to  retrieve data from a student data file */

/* 86_read.c */

#include  <stdio.h>

#include  <conio.h>

int main()

{

      int sno, sub1, sub2, sub3;

      char name[10];

      FILE *fp = fopen("stud.dat",  "a+");

      clrscr();
```

```
	printf("Student Records are as  follows....\n");

	do{

		fscanf(fp, "%d%s%d%d%d\n",  &sno, name, &sub1, &sub2, &sub3);

		printf("%5d%15s%3d%3d%3d\n",  sno, name, sub1, sub2, sub3);

	}while(!feof(fp));

	fclose(fp);

	return 0;

}
```

13. Step-by-Step C/C++ --- C++ Programming - OOPsOOP (Object Oriented Programming) in C++

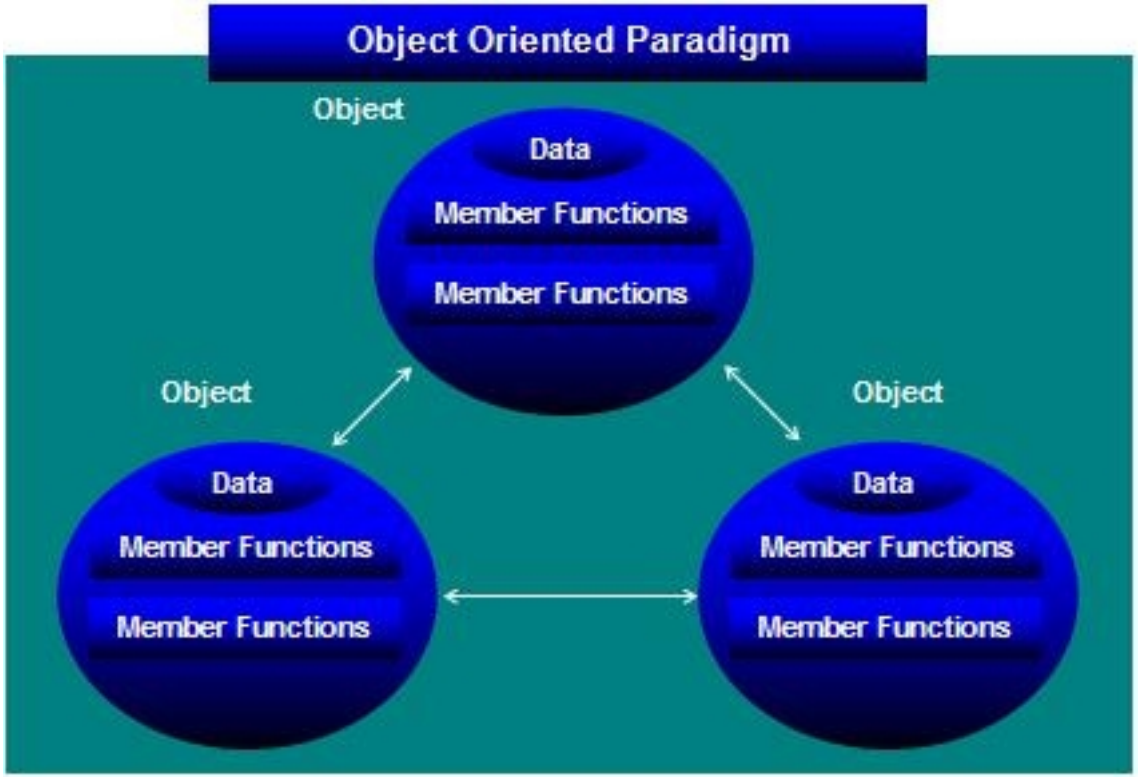
1. Object Oriented Paradigm
2. Characteristics of Object-Oriented Language
    - Objects
    - Classes
    - Data abstraction
    - Data encapsulation
    - Inheritance
    - Polymorphism
    - Dynamic binding
    - Message passing
3. History of C++
4. Classes and Objects
5. Member functions defined outside the class
6. Array of Objects
7. Objects as Arguments

1. Object Oriented Paradigm

The basic idea behind the Object Oriented Paradigm is to combine into a single unit of both data and the functions that operate on that data. Such a unit is called an object.

Through this method we cannot access data directly. The data is hidden, so, is safe from
Accidental alteration. Data and its functions are said to be encapsulated into a single entity. Data encapsulation and data hidings are key terms in the description of object-oriented language.

A C++ program typically consists of a number of objects, which communicate with each other by calling one another(TM)s member functions. The organization of a C++ program is shown in this figure.



2. Characteristics of Object-Oriented Language

Here are few major elements of Object-Oriented languages.

- Objects
- Classes
- Data abstraction
- Data encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

Objects

Object is an instance of a class. Combining both data and member functions. Objects are the basic run-time entities in an object-oriented system.

Classes

A template or blueprint that defines the characteristics of an object and describes how the object should look and behave.

Data Abstraction

Identifying the distinguishing characteristics of a class or object without having to process all the information about the class or object. When you create a class " for example, a set of table navigation buttons " you can use it as a single entity instead of keeping track of the individual components and how they interact.

Data Encapsulation

An object-oriented programming term for the ability to contain and hide information about an object, such as internal data structures and code. Encapsulation isolates the internal complexity of an object's operation from the rest of the application. For example, when you set the Caption property on a command button, you don't need to know how the string is stored.

Inheritance

An object-oriented programming term. The ability of a subclass to take on the characteristics of the class it's based on. If the characteristics of the parent class change, the subclass on which it is based inherits those characteristics.

To inherit the qualities of base class to derived class.

Polymorphism

An object-oriented programming term. The ability to have methods with the same name, but different content, for related classes. The procedure to use is determined at run time by the class of the object. For example, related objects might both have Draw methods. A procedure, passed such an object as a parameter, can call the Draw method without needing to know what type of object the parameter is.

Dynamic Binding

Dynamic refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

Message Passing

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language therefore involves the following basic steps:

1. Creating classes that define objects and their behavior.
2. Creating objects from class definitions.
3. Establishing communication among objects.

3. History of C++

Year	Language	Developed by	Remarks
1960	ALGOL	International Committee	Too general, Too abstract
1963	CPL	Cambridge University	Hard to learn, Difficult to implement
1967	BCPL	Martin Richards	Could deal with only specific problems
1970	B	Ken Thompson	AT & T Bell Labs Could deal with only specific problems
1972	C	Dennis Ritchie	AT & T Bell Labs Lost generality of BCPL and B restored
Early 80(TM)s	C++	Bjarne Stroustrup	AT & T Introduces OOPs.

C++ is an object-oriented programming language. Initially named ˜C with Classes(TM), C++ was developed by ***Bjarne Stroustrup*** at AT & T Bell laboratories in Murry Hill, New Jersey, USA, in the early eighties.

Stroustrup, an admirer of Simula67 (an OOP language) and a strong supporter of C, wanted to combine the best of both languages and create a more power and elegance of C. The result was C++.

C++ is a truly Object Oriented Language, So. It must be a collection of classes and objects.

4. Classes and Objects

A class is a way to bind the data and its associated functions together. It allows the data to be hidden, if necessary, from external use. When defining a class, we are creating a new abstract data type that can be treated like any other built-in data type. Generally, a class specification has two parts:

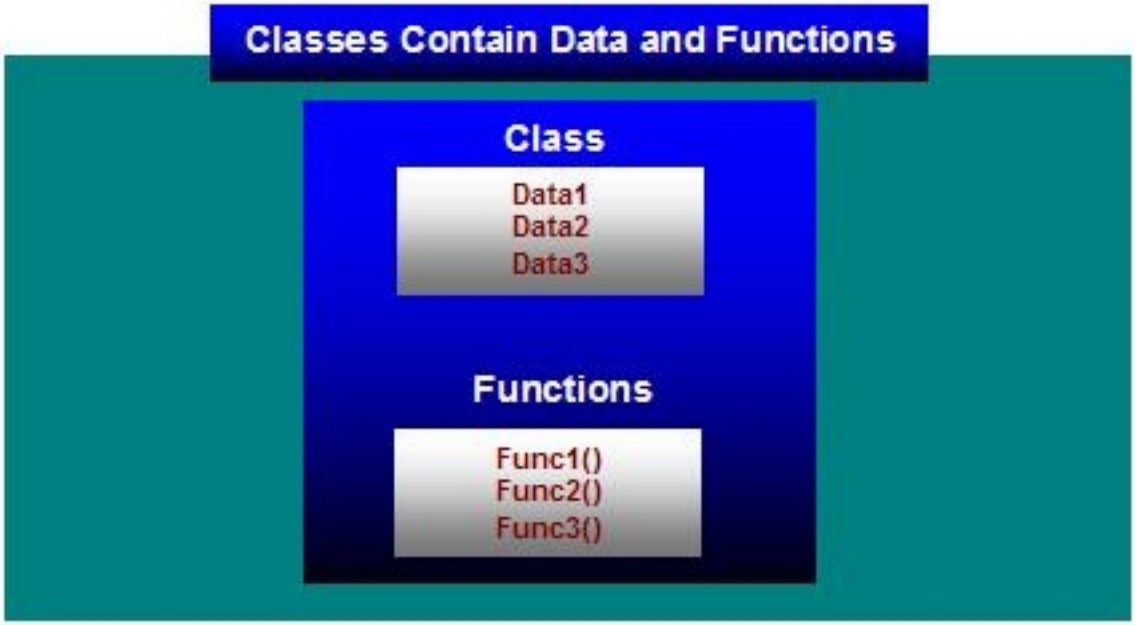
1. Class declaration
2. Class function definition

The declaration specifies the type and scope of both data and member functions of class. Where as definition specifies the executable code of the function.

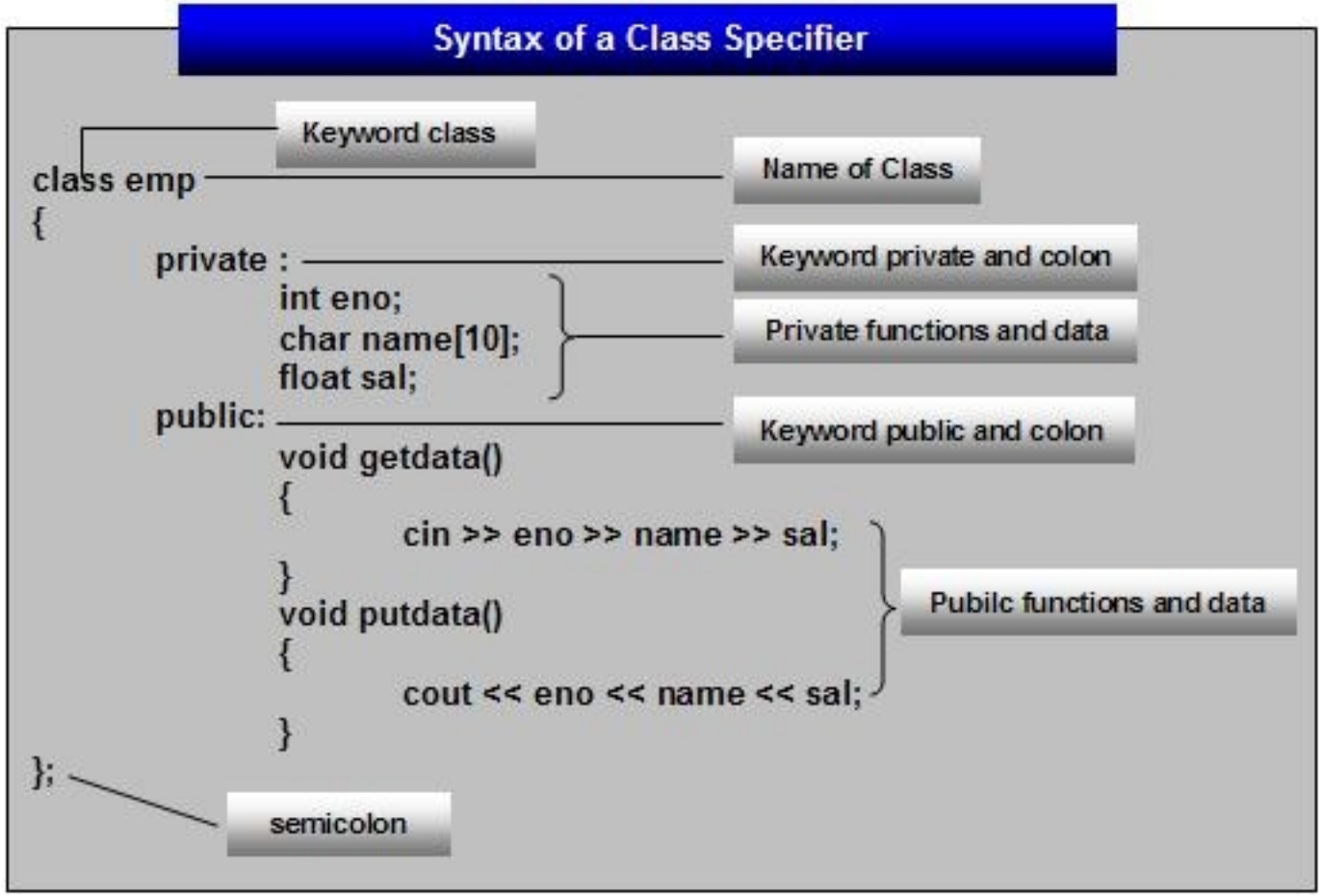
The general form of a class declaration is:

```
class class_name

{

     private:

          variable declarations;

          function declarations;

     public:

          variable declarations;

          function declarations;

};
```

The class declaration is similar to struct declaration. The key word class specifies that the data and functions be of private by default. Where as a struct key word specifies that the data and functions be of public by default. The keywords private and public are known as visibility labels.



Here is an example class to implement an employee class.



The following is the complete program of emp class.

```
// Program to accept and display employee  information

#include <iostream>
```

```
using namespace std;

class emp                                          //  class definition

{

            private  :                                    // private data, functions

                        int  eno;

                        char  name[10];

                        float  sal;

            public  :                                    //  public data, functions

                        void  getdata()

                        {  cin >> eno >> name >> sal;   }

                        void  putdata()

                        {  cout << eno << name << sal;   }

};

int main()

{

            emp  e;

            e.getdata();
```

```
            e.putdata();

            return  0;

}
```

5. Member functions defined outside the class

There is a possibility to define member functions outside of the class using scope resolution operator (::).

```
// Program to accept and display employee  information

#include <iostream>

using namespace std;

class emp                                          //  class definition

{

            private  :                        // private data,  functions

                        int  eno;

                        char  name[10];

                        float  sal;

            public  :                                    //  public data, functions
```

```
                        void  getdata();

                        void  putdata();

};

void emp::getdata()

{   cin  >> eno >> name >> sal;   }

void emp::putdata()

{   cout  << eno << name << sal; }

int main()

{

            emp  e;

            e.getdata();

            e.putdata();

            return  0;

}
```

6. Array of Objects

C++ compiler also supports array of objects.
Below example illustrates the advantage of Objects using arrays.

```
// Program to accept and display employee  information

#include <iostream>

using namespace std;

class emp                                          //  class definition

{

            private  :                        // private data,  functions

                        int  eno;

                        char  name[10];

                        float  sal;


            public  :                                    //  public data, functions

                        void  getdata()

                        {  cin << eno << name << sal;   }

                        void  putdata()

                        {  cout >> eno >> name >> sal;   }

};
```

```
int main()

{

            emp  e[10];                    // declaration  of array of objects

            for(i  = 0; i <10; i++)       // accessing  objects properties and methods

                        e[i].getdata();

           for(i = 0;  i< 10; i++)

                        e[i].putdata();

           return 0;

}
```

7. Objects as Arguments

Passing Objects to functions is similar to passing structures, arrays to functions. The following program demonstrates how objects passed to functions.

```
// Program to accept and display employee  information

#include <iostream>

using namespace std;

class emp                                          //  class definition

{
```

```
            private  :                        // private data,  functions
                        int  eno;
                        char  name[10];
                        float  sal;
            public  :                                  //  public data, functions
                        void  getdata()
                        {  cin >> eno >> name >> sal;   }
                        void  putdata()
                        {  cout << eno << name << sal;   }
};
void operate(emp t);
int main()
{

            emp  e;
            operate(e);
```

```
}

void operate(emp t)

{

            t.getdata();

            t.putdata();

            return  0;

}
```

8. Returning Objects from functions

We saw objects being passed as arguments to functions, now we will discuss about how to return objects from functions.

```
// Program to accept and display employee  information

#include <iostream>

using namespace std;

class emp                                          //  class definition

{

            private  :                        // private data,  functions

                        int  eno;
```

```
                        char  name[10];
                        float  sal;
            public  :                                   //  public data, functions
                        void  getdata()
                        {  cin >> eno >> name >> sal;   }
                        void  putdata()
                        {  cout << eno << name << sal;  }
};
emp get();
void put(emp t);
int main()
{


            emp  e;
            e =  get();
            put(e);
```

```
            return  0;

}

emp get()

{

            emp  t;

            t.getdata();

            return  t;

}

void put(emp t)

{

            t.putdata();

}
```

9. Constructor

The following example shows two ways to give values to the data items in an object. Sometimes, however, it(TM)s convenient if an object can initialize itself when it(TM)s first created, without the need to make a separate call to a member function.

Automatic initialization is carried out using a special member function called a constructor. A ***constructor*** is a member function that is executed automatically whenever an object is created.

```
// Program to accept and display employee  information using constructors

#include <string.h>

#include <iostream>

using namespace std;

class emp                              //  class definition

{

      private  :                        // private data,  functions

           int  eno;

           char  name[10];

           float  sal;

      public    :                       //  public data, functions

            emp()  { ; }                 //  constructor without arguments

            emp(int  teno, char tname[10], float tsal) // constructor with arguments

            {

                 eno  = teno;

                 strcpy(name,  tname);

                 sal  = tsal;
```

```
            }

            void  getdata()

            {  cin >> eno >> name >> sal;   }

            void  putdata()

            {  cout << eno << name << sal << endl;  }

};

int main()

{

            emp  e1(1001, "Magic", 6700.45);

            emp  e2;

            e2.getdata();

            e1.putdata();

            e2.putdata();

            return  0;

}
```

The above example program accepts values in two ways using constructors and using member functions. An object, whenever it was declared it

automatically initialized with the given values using constructors. Where as object ***e2*** is accessible by its member function only.

One more example to distinguish the use of constructor.

```
//  Objects  represents a counter variable

#include <iostream>

using namespace std;

class counter

{

            private  :

            int count;                                  // variable count

            pubilc  :

                  counter()           { count = 0; }    //  constructor

                        void  inc_count() { count++; }    // increment count

                        int  get_count()   { return count; }  // return count

};

int main()

{
```

```
            counter  c1, c2;                    //  define and initialize

            cout  << â€œ\nC1 = â€œ << c1.get_count();       //  display

            cout  << â€œ\nC2 = â€œ << c2.get_count();
            c1.inc_count();                            //  increment c1

            c2.inc_count();                            //  increment c2

            c2.inc_count();                            //  increment c2

            cout  << â€œ\nC1 = â€œ << c1.get_count();       //  display again

            cout  << â€œ\nC2 = â€œ << c2.get_count();

            return  0;

}
```

A constructor has the following characteristics.

- Automatic initialization
- Return values were not accepted
- Same name as the class
- Messing with the format

10. Destructors

A destructor has the same name as the constructor (which is the same as the class name) but preceded by a tilde:

```
// Demonstration of a destructor

#include <iostream>
```

```
using namespace std;

class temp

{

            private  :

                        int  data;

            public  :

                        temp()  { data = 0;  }                  // Constructor (same name as class)

                        ~temp()  { }                               //  destructor (same name with tilde)

}

int main()

{

            temp  t;

            return  0;

}
```

11. Constructor Overloading

The ability to have functions with the same name, but different content, for related class. The procedure to use is determined at run time by the class of the

object.

```
// Demonstration of a constructor overloading

#include <iostream>

using namespace std;

class ttime

{

       private  :

               int  hh, mm, ss;

       public  :

               ttime()  {hh = 0; mm = 0; ss = 0; } // Constructor  with initialization

               ttime(int  h, int m, int s)        //  Constructor with 3 arguments

               {

                    hh  = h; mm = m ; ss = s;

               }

               ttime(int  h, int m)             //  Constructor with 2 arguments

               {
```

```
                    hh  = h; mm = m; ss = 0;

                }

                ttime(int  h)                    //  Constructor with 1 argument

                {

                     hh  = h; mm = 0; ss = 0;

                 }

                 ~ttime()  { }

                 void  get_time()

                 {

                       cin  >> hh >> mm >> ss;

                 }

                 void  put_time()

                 {

                       cout  << endl << hh << "   " << mm <<"   " <<  ss;

                 }

};

int main()
```

```
{

            ttime  t1, t2(12, 12, 12), t3(4, 5), t4(11);       //  Calling constructors

            t1.get_time();

            t1.put_time();

            t2.put_time();

            t3.put_time();

            t4.put_time();

            return  0;

}
```

12. Static Class Data

If a data item in a class is defined as ***static,*** then only one such item is created for the entire class, no matter how many objects there are. A static data item is useful when all objects of the same class must share a common item of information. A member variable defined as ***static*** has similar characteristics to a normal static variable: It is visible only within the class, but its lifetime is the entire program.

```
// Demonstration of a static data

#include <iostream>

using namespace std;
```

```
class temp

{

    private  :

        static  int count;         // Only  one data item for all objects

    public  :

        temp()  { count++; }       //  increment count when object created

        int  getcount() { return count; }        //  return count

};

int main()

{

    temp  t1, t2, t3;        //  create three objects

    cout  << â€œ\nCount is    â€œ <<  t1.getcount( ); // each object

    cout  << â€œ\nCount is    â€œ <<  t2.getcount( ); // sees the same

    cout  << â€œ\nCount is    â€œ <<  t3.getcount( ); // value of count

    return  0;

}
```

Out put of the above program is as follows: (if it(TM)s still static)

Count is 3
Count is 3
Count is 3

Out put of the above program (if it(TM)s automatic)

Count is 1
Count is 1
Count is 1

13. Static Member Functions

Like static member variable, we can also have static member functions. A member function that is declared static has the following properties.
- A static functions can have access to only other static members (functions or variables) declared in the same class.
- A static member function cab be called using the class name (instead of its objects) as follows:

```
class-name :: function-name;
```

```
// Program to demonstrate static member function

#include <iostream>

using namespace std;

class test

{

     int  code ;

     static  int count;   // static member variable
```

```
    public:

         void  setcode() { code = ++count; }

         void  showcode() { cout << "Object number :" << code <<  endl; }

         static  void showcount()                    //  static member function

         {

               cout  << "Count  :" <<  count << endl;

         }

};

int test :: count;

int main()

{

           test  t1, t2;

           t1.setcode();

           t2.setcode();

           test::showcount();  // accessing static function

           test  t3;
```

```
            t3.setcode();

            test::showcount();

            t1.showcode();

            t2.showcode();

            t3.showcode();

            return  0;

}
```

14. Friend Functions

```
class abc
{
   ....
   ....
public:
   ....
   ....
   friend void xyz(void);  // declaration
};
```

Private members cannot be accessed from outside the class. That is, a non-member function can(TM)t have an access to the private data of a class. However, there could be a situation where we would like two classes to share a particular function. It(TM)s simply achieved through ***Friend*** functions. A friend function possesses certain special characteristics:

- It is not in the scope of the class to which it has been declared as ***friend.***
- Since it is not in the scope of the class, it cannot be called using the object of the class. It can be invoked like a normal function without the help of any object.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name.
- It can be declared either in the public or the private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.

```
// Program to demonstrate friend function

#include <iostream>

using namespace std;

class test
```

```
{

      int  a;

      int  b;

      public:

            void  setvalue() { a = 25; b = 40; }

            friend  float sum(test s);             // FRIEND declared

};
float sum(test s)

{

     return  float (s.a + s.b ) / 2.0;        //  s.a & s.b are the private members

                                             //  of class test but they were accessible

                                             //  by friend function

}

int main()

{

      test  x;

      x.setvalue();
```

```
      cout  << "Mean value  = "  << sum(x) << endl;

      return  0;

}
```

One more example to implement a ***friend functions as a bridge between two classes***.
The following program creates two objects of two classes and a function friendly to two classes.
In this example friend function is capable of accessing both classes data members and calculates the biggest of both class data members.

```
#include <iostream>

using namespace std;

class second;

class first

{

      int  a;

      public:

         first(int  temp) { a = temp; }

         friend  void max(first, second);

};

class second
```

```
{

      int  b;

      public:

          second(int  temp) { b = temp; }

          friend  void max(first, second);

};
void max(first f, second s)

{

      if  ( f.a > s.b )                    //  both first, second data members can be

           cout  << â€œMax â€œ<<  f.a;          // accessed thru friend max  function

      else

           cout  << â€œMax â€œ<< s.b;

}

int main()

{

      first  f(20);

      second  s(30);
```

```
      max(f,  s);

      return  0;

}
```

Ref: Object-oriented Programming in Turbo C++: Robert Lafore

*14. Step-by-Step C/C++ --- C++ Programming - Inheritance**Inheritance*

Introduction

Inheritance is the most powerful feature of Object Oriented programming. Inheritance is the process of creating new classes, called derived classes from

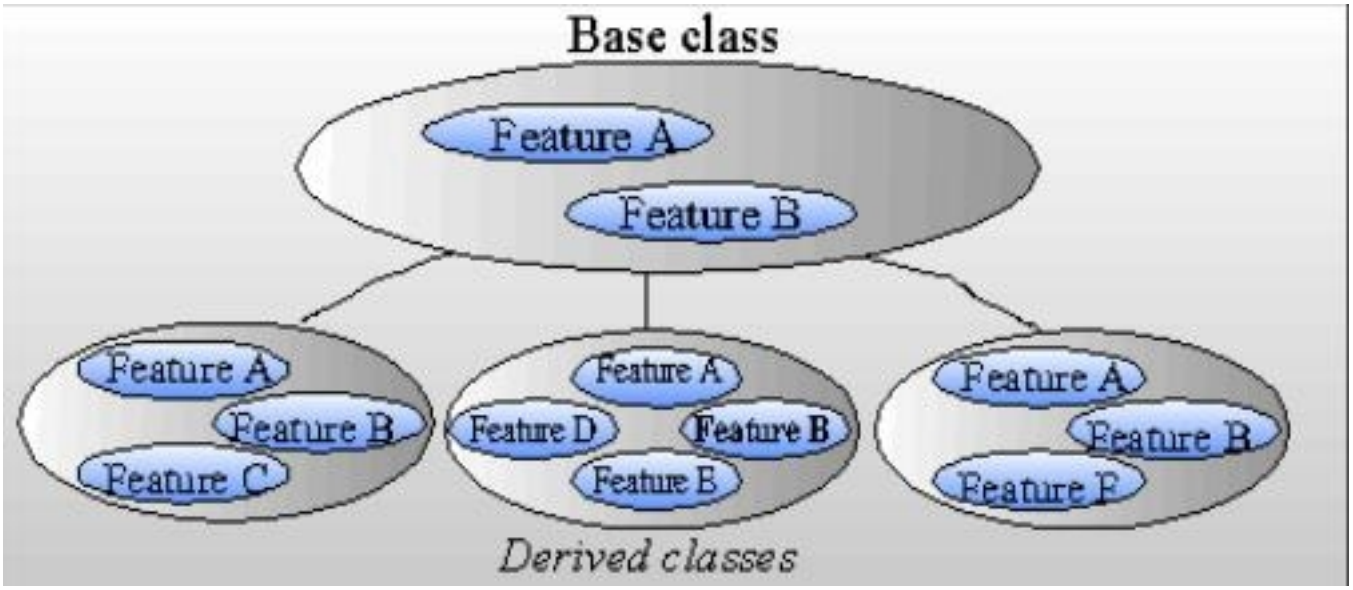
existing or bases classes. The derived class inherits all the capabilities of the base class but can add embellishments and refinements of its own.
A class, called the derived class, can inherit the features of another class, called the base class.

To inherit the qualities of base class to derived class is known as inheritance.

Its noun is heritage. We know in our daily lives, we use the concept of classes being derived into subclasses. For E.g. Vehicle is class it's again divided into Cycles, Bikes, Autos, trucks, busses and so on.

Here Vehicle is known as Base class and the derived items are known as derived classes or subclasses.

Generally every base class has a list of qualities and features. The main theme in this inheritance is to share all the common characteristics of base class to derived classes.



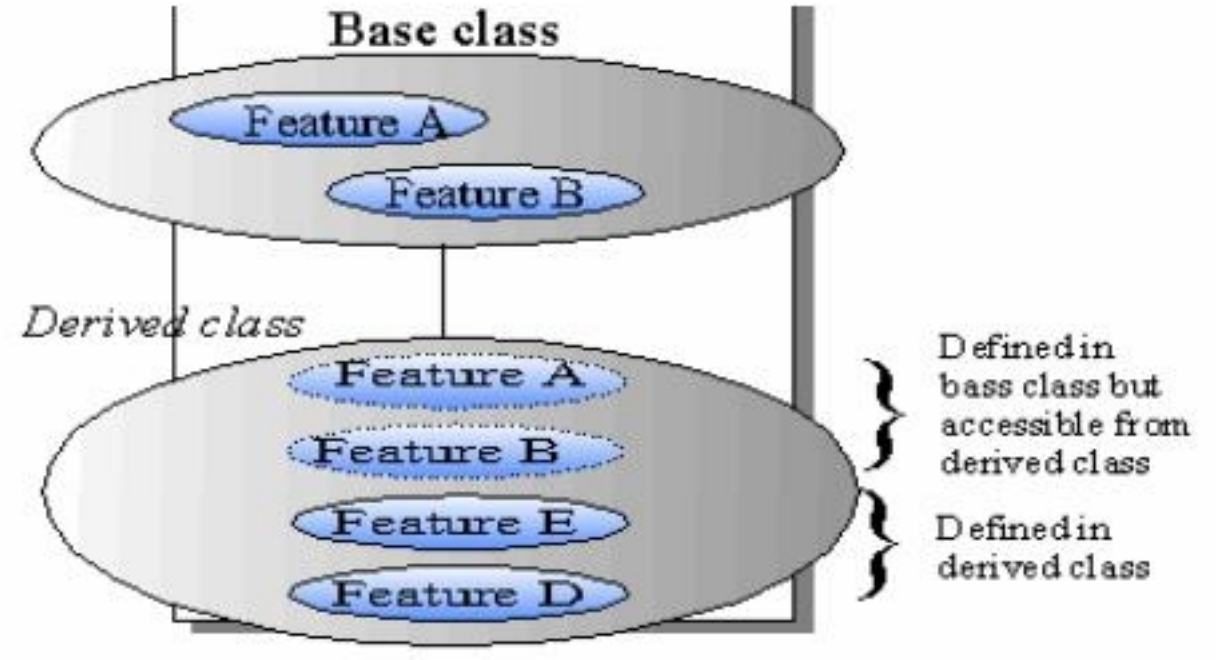
Inheritance has an important feature to allow reusability. One result of reusability is the ease of distributing class libraries. A programmer can use a class created another person or company, and, without modifying it, derive other classes from it that are suited to particular situations.

Derived class and Base class

A class, called the derived class, can inherit the features of another class, called the base class.
The derived class can add other features of its own, so it becomes a specialized version of the base class. Inheritance provides a powerful way to extend the capabilities of existing classes, and to design programs using hierarchical relationships.

Accessibility of base class members from derived classes and from objects of derived classes is an important issue. Objects of derived classes can access data or functions in the base class that are prefaced by the keyword protected from derived classes but not. Classes may be publicly privately derived from base classes. Objects of a publicly derived class can access public members of the base class, while objects of a privately derived class cannot.

Diagram shows how Derived class inherits.



A class can be derived from more than one base class. This is called multiple inheritances. A class can also be contained within another class.

Specifying the Derived Class

Class declaration is so easy using the keyword ***class*** as well as the derived class declaration is also easy but the class must be ends with its base class id and access specifier.

Syntax to declare a derived class:

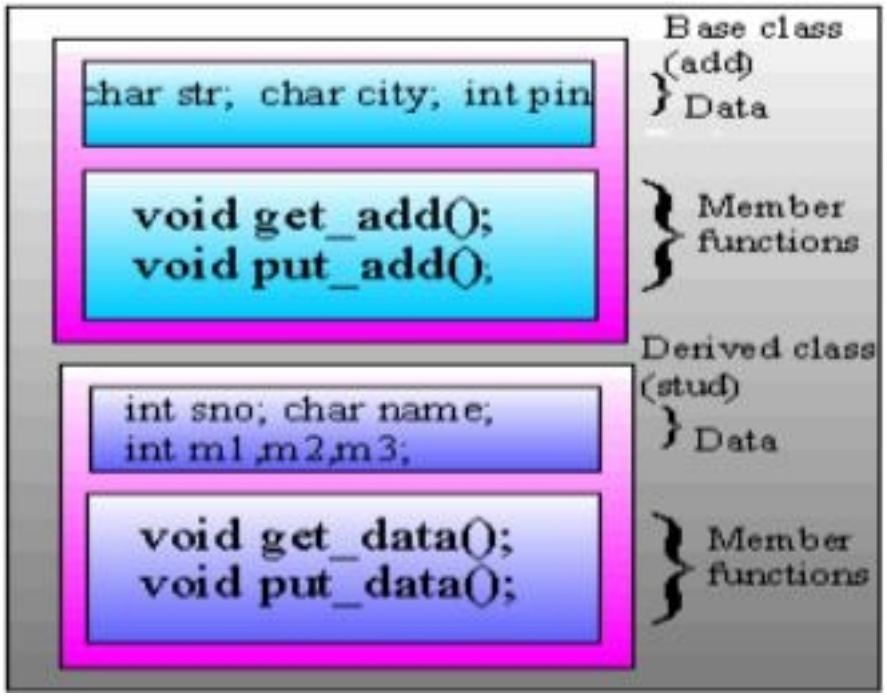
Class <Class_name> : ***<Access_Specifier> <Base_Class_Name>*** **',**

For. E.g. ***class*** result ***: public stud;***

```
/*  program to accept and display a student  record  */

#include  <iostream>
  using namespace std;
class add
{
    private  :
        char  str[20];
        char  city[20];
        int  pin;
    public  :
        void  get_add()
        {
            cout  << "Enter Address     street,city,pin";
            cin  >> street >>city>>pin;
        }
        void  put_data()
        {
             cout  << "Address is    "<< str
                 <<endl<<city  <<endl<<pin;
        }
};
class stud  : public add
{
     private  :
          int  sno;
          char  name[20];
          int  m1,m2,m3;
     public :
          void  get_data()
          {
               cout  << "Enter Student No.  ";  cin >> sno;
               cout  << "Enter Student Name   "; cin >> name;
```

```
            cout  << "Enter Student 3subjects marks   ";
            cin  >> m1 << m2 << m3;
        }
        void  put_data()
        {
            cout  << "Student number  :" << sno;
            cout <<  "Student name      :" << name;
            cout <<  "Student marks     :" << m1 <<  "  "  <<m2<<"   "<<m3;
        }
};
int main()
{
    stud s;
    s.get_add();
    s.get_data();
    s.put_add();
    s.put_data();
    return  0;
}
```

Diagramed explanation for the above program



Derived Class Constructors

If a class is declared with its own constructors it is a base class of another. The derived class is also having its own constructors. If an object is declared which is the constructor will be executed? No doubt it executed the constructor of the derived class. It you still want to execute the constructor of Base class or both Derived and Base class constructors simply call the Base constructor in Derived class constructor.

```
/*  Constructors in derived class */

#include  <iostream>

using namespace std;
```

```
class  Add

{

     protected :                              // NOTE : not private

            unsigned int a;

     public :

            Add()   {  a = 0; }                 //  constructor , no args

            Add( int c ) { a = c; }            // constructor , one args

            int get_val(){     return a; }      // return A value

            Add operator ++ ()                 // increment count

            {

                 a++;                          // increment count,  return

                 return  Add(a);                // an unnamed  temporary object

            }                                  // initialized to this  count

};


class  Sub : public Add

{
```

```
     public:

          Sub() : Add() { }                    // Constructor, no args

          Sub(int c) : Add(c) { }              // Constructor, one args

          Sub operator -- ()                   // decrement value of A,  return

          {                                    // an unnamed temporary object

               a--;                            //initialized to  this Value

               return  Sub(a);

          }

};

int  main()

{

     Sub ob1;                                 // class Sub

     Sub ob2(100);

     cout << "\nOb1  =" << ob1.get_val();      // display

     cout << "\nOb2  =" << ob2.get_val();      // display

     ob1++; ob1++; ob1++;                     // increment ob1

     cout << "\nOb1  =" << ob1.get_val();      // display
```

```
      ob2--; ob2--;                          // decrement ob2

      cout << "\nOb2  =" << ob2.get_val();    // display

      Sub ob3=ob2--;                         //  create ob3 from ob2

      cout << "\nOb3  =" << ob3.get_val();    // display

      return 0;

}
```

ACCESS SPECIFIERS

Access specifiers are used to control, hide, secure the both data and member functions. Access specifiers are of 3 types
- Public Access Specifier
- Private Access Specifier
- Protected Access Specifier.

Public :

If a member or data is a public it can be used by any function with in class and its derived classes also.
In ***C++*** members of a ***struct*** or ***union*** are public by default.
Public Member of a class can be inherited to the derived class when the class is inherited publicly but not the member functions(privately).

Private :

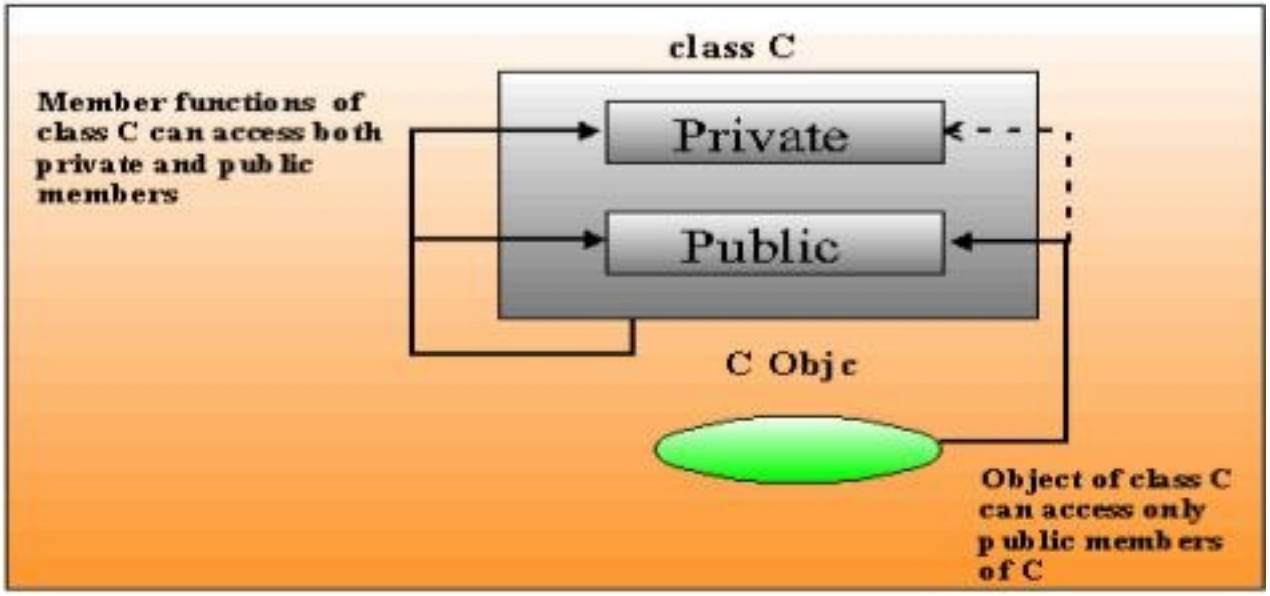
Member functions and friend of the class in which it is declared can only use it.
Members of a class are private by default.
Private member of a class doesn(TM)t be inherited to a derived class when the base class is inherited publicly or privately. It there is need we have to write member function, which are returns, those values.

Protected :

It is access as the same as for private in addition, the member can be used by member functions and friends of classes derived from the declared class but not only in Objects of the derived type.

The protected member of a class can be inherited to the next derived class only. But not to the later classes.

Access Specifiers without Inheritance



More About Protected Access Specifier

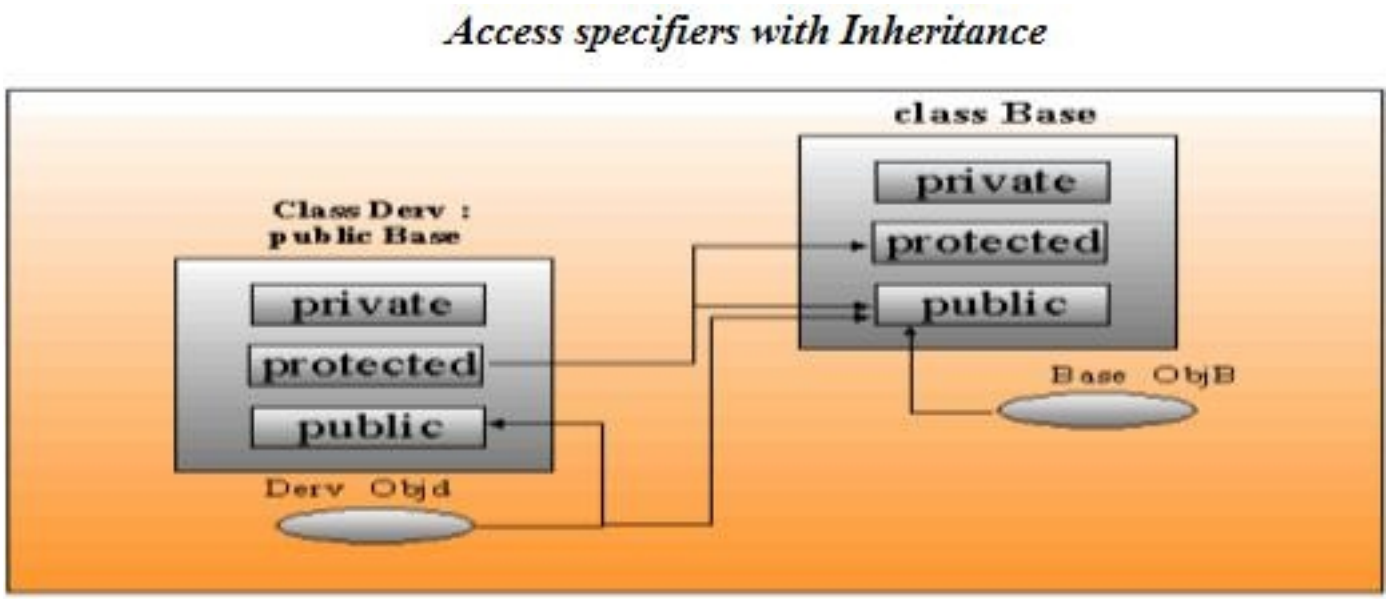
To provide the functionality without modifying the class. ***Protected*** can accessed by it self and derived class-protected members only but in objects or the subderived class or the outside class.

Scope of Access Specifiers

Access Specifier	***Accessible from Own class***	***Accessible from derived class***	***Accessible from Objects outside class***

Public Yes Yes Yes
Protected Yes Yes No
Private Yes No No

Access specifiers with Inheritance



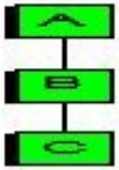
Types of Inheritance

Types of Inheritance

1. **Single Inheritance :**
   If a class is derived from a base class is called single inheritance.
   A → B

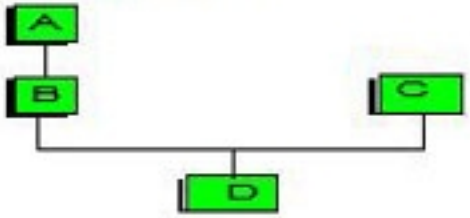
2. **Multiple Inheritance :**
   If a class is derived from more than one Base class is called Multiple Inheritance.

3. **Multilevel Inheritance :**
   If a class is derived from a derived class is called Multilevel Inheritance.



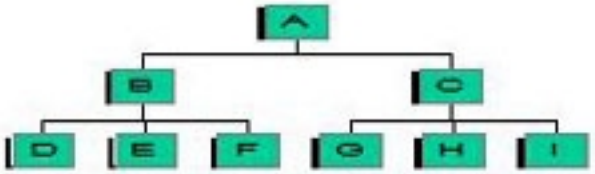
4. Hybrid Inheritance :

The class with a combination of both Multilevel and Multiple Inheritances is known as Hybrid Inheritance.



5. Hierarchy Inheritance :

It consists of a Base class and its multiple derived classes. The Base class has the ability to control all the derived classes.




```
/*
Program to demonstrate Multiple  Inheritance
*/
#include <iostream>
      using namespace std;
 class M
      {
```

```
    protected  :
         int  m;
    public  :
         void  getm()
         {
              cout  << "nEnter M value  :";
              cin  >> m;
         }
     };
class N
     {
    protected  :
         int  n;
    public  :
         void  getn()
         {
              cout  << "nEnter N value  :";
              cin  >> n;
         }
     };
class P : public N, public M
     {
    public  :
         void  disp()
         {
              cout  << "n  M   =   " << m;
              cout  << "n  N   =   " << n;
              cout  << "n  M*N =  " << m*n;
         }
     };
int main()
     {
     P  p;
```

```
    p.getm();
    p.getn();
    p.disp();
    return  0;
  }
```

If a base class is publicly inherited then the public members, member function can be accessible to the member functions of the derived class and to the Objects also where as If a base class is inherited privately then the public member of base class are inherited to the member functions of the derived class only but not to the objects.

```
/*
 A  program to demonstrate Multilevel Inheritance
 */
class student
    {
       int  rno;
    public:
       void  getrno()
       {
          cout  << "Enter  Number :";
          cin  >> rno;
       }
       void  showrno()
       {
          cout  << "Student Number:" <<  rno;
       }
    };
 class test : public student
    {
       int  m1,m2;
    public  :
       void  getmarks()
```

```
        {
            cout  << "Enter marks 1  :" ;  cin >> m1;
            cout  << "Enter marks 2  :" ;  cin >> m2;
        }
        int  retm1()
        {
            return  m1;
        }
        int  retm2()
        {
            return  m2;
        }
    };
class result : public test
    {
        int  tot;
    public:
        void  get()
        {
            getrno();
            getmarks();
        }
        void  showresult();
        void  show()
        {
            showrno();
            showresult();
        }
    };
void result::showresult()
    {
    int  s1,s2;
    s1=retm1();
```

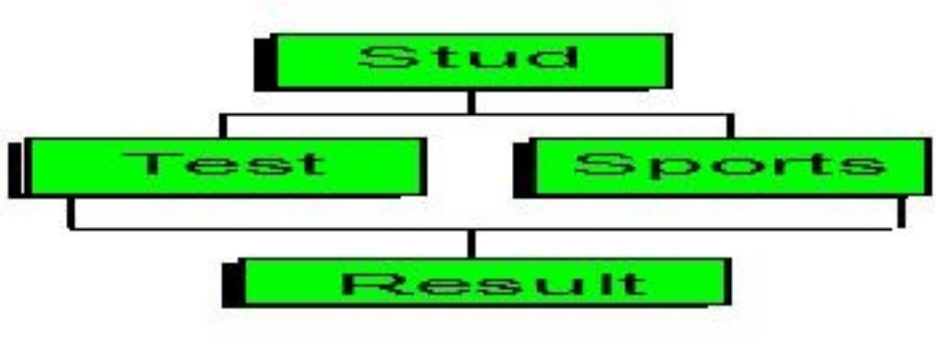
```
    s2=retm2();
    tot=s1+s2;
    cout  << "nMarks  " <<  s1 << "  "<< s2;
    cout  << "nTotal marks  "  << tot;
    }
int main()
    {
    result  a;
    a.get();
    a.show();
    return  0;
    }
```

```
    /*
     Program to demonstrate Hybrid Inheritance
     */
#include <iostream>
using namespace std;
class student
{
        int  rno;
    public:
        void  getrno()
        {
            cout  << "Enter  Number :";
            cin  >> rno;
        }
        void  showrno()
        {
            cout  << "nStudent Number   :" <<  rno;
        }
};
```

```
class test : public student
{
    protected  :
        int  m1,m2;
    public  :
        void  getmarks()
        {
            cout  << "Enter marks 1  :" ;  cin >> m1;
            cout  << "Enter marks 2  :" ;  cin >> m2;
        }
        void  showmarks()
        {
            cout  << "nMarks  of 2  subjects "  <<  m1 << "  " << m2;
        }
};
class sports
{
    protected  :
        int  score;
    public  :
        void  getscore()
        {
            cout  << "Enter   Score  :";
            cin  >> score;
        }
};
class result : public test, public sports
{
    public  :
        void  getdata()
        {
            getrno();
            getmarks();
```

```
            getscore();
        }
        void  putdata()
        {
            showrno();
            showmarks();
            cout  << "nScore  is  " << score;
            cout  << "n Total marks   "  << m1+m2;
        }
};
int main()
{
    result  r;
    r.getdata();
    r.putdata();
    return  0;
}
```

Pictorial representation of the above program:



In the above figure student class inherited to result in two ways. One is via test another one is via sports then two sets of members, member functions of common base class student are inherited to the derived class result at the time of execution the system will get confuse to use what set of member functions of base class.

This can be avoided by making the common base class as virtual base class.

Eg:

```
class student {    };

class test : virtual public student {     };

class sports : virtual public student {     };

class result : public  test, sports  {    };
```

Ref: Object-oriented Programming in Turbo C++: Robert Lafore

15. Step-by-Step C/C++ --- C++ Programming - Operator Overloading

Operator Overloading

1. Introduction
2. Operator

- ¢ Rules of Operator Overloading
- ¢ Restrictions on Operator Overloading

3. Overloading Unary Operators
4. Overloading Binary Operators
5. Operator Overloading with Strings

1. Introduction

```
// Assign a variable to another

#include  <iostream>

using namespace std;

int  main()

{

    int a = 10, b;

    b = a;
//  valid
```

```
cout << b;
return 0;
}
```

```
// Assign an object to another

#include  <iostream>
```

```
using namespace std;

class  emp

{

    public:

    int eno;

    float sal;

};

int  main()

{

    emp e1= { 1001, 2300.45 },e2 ;

    cout << endl << e1.eno  << e1.sal;

    e2 = e1;
// valid
```

```
    cout << endl << e2.eno  << e2.sal;
    return 0;
}
```

Expressions are common in every language; an expression is a collection of operands and operators. Where as an operation is a collection of expressions. The above two programs demonstrate how variables/objects were assigned together.

Both programs are valid, they demonstrates the use of equalto (=) operator.

```
// using operator + to perform an arithmetic operation with variables
#include <iostream>
using namespace std;
int main()
{
        int a = 10, b = 15, c;
        c = a + b;              // valid expression
        cout << c;
        return 0;
}
```

```
// using operator + to perform an arithmetic operation with objects
#include <iostream>
using namespace std;
class emp
{
        public:
        int eno;
        float sal;
};
int main()
{
        emp e1= { 1001, 2300.45 }, e2= e1, e3 ;
        cout << endl << e1.eno << e1.sal;
        e3 = e1 + e2;                   // Illegal structure operation
        cout << endl << e3.eno << e3.sal;
}
```

Operator overloading is one of the most exciting feature of object-oriented programming. It is used to overcome the situation like the above ***illegal structure operation***. It can transform complex, obscure program listing into intuitively obvious ones.

Through Operator overloading we can see how the normal C++ operators can be given new meanings when applied to user-defined data types. The keyword operator is used to overload an operator, and the resulting operator will adopt the meaning supplied by the programmer.

For example using object we can perform direct string assignment operation.

```
//  Program to assign a string to other

#include  <string.h>

#include  <stdio.h>
```

```
#include  <iostream>

using namespace std;

class  string

{

     char *str;

     public:

          string() { }

          string(char *s) { str =  s; }

          void putstring()

          {

               cout  << str;

          }

};

int  main()

{

     string s1("Computer");
```

```
    string s2;

    s2 = s1;

    s2.putstring();

    return 0;


}
```

2. Operator

type operator operator-symbol (parameter-list)

The ***operator*** keyword declares a function specifying what ***operator-symbol*** means when applied to instances of a class. This gives the operator more than one meaning, or "overloads" it. The compiler distinguishes between the different meanings of an operator by examining the types of its operands.

Rules of Operator Overloading

- You can overload the following operators:

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

- If an operator can be used as either a unary or a binary operator, you can overload each use separately.

- You can overload an operator using either a non-static member function or a global function that's a friend of a class. A global function must have at least

one parameter that is of class type or a reference to class type.

- If a unary operator is overloaded using a member function, it takes no arguments. If it is overloaded using a global function, it takes one argument.

If a binary operator is overloaded using a member function, it takes one argument. If it is overloaded using a global function, it takes two arguments.

Restrictions on Operator Overloading

- You cannot define new operators, such as **.

- You cannot change the precedence or grouping of an operator, nor can you change the numbers of operands it accepts.

- You cannot redefine the meaning of an operator when applied to built-in data types.

- Overloaded operators cannot take default arguments.

- You cannot overload any preprocessor symbol, nor can you overload the following operators:

.

.*

::

?:

The assignment operator has some additional restrictions. It can be overloaded only as a non-static member function, not as a friend function. It is the only operator that cannot be inherited; a derived class cannot use a base class's assignment operator.

3. Overloading Unary Operators

Let(TM)s start off by overloading a ***unary operator***. Unary operators act on only one operand. (An operand is simply a variable acted on by an operator).

Examples of unary operators are the increment and decrement operators ++ and --, and the unary minus.

Example:
The following example demonstrates the use of increment operator ++.

```
#include  <iostream>

using namespace std;

class  counter

{

      private:

            unsigned int count;

      public:

            counter(){ count = 0; }

            int get_count()  { return count; }

            counter operator ++()

            {

                  count++;

                  counter  temp;

                  temp.count =  count;
```

```
                return temp;

            }

};

int  main()

{

      counter c1, c2;                          // c1 =  0,  c2 = 0

      cout << "\nC1 = "  << c1.get_count();       // display

      cout << "\nC2 = "  << c2.get_count();



++c1;
```

// c1 = 1

c2 = ++c1 ; // c1 = 2, c2 = 2

cout << "nC1 = " << c1.get_count(); // display again
cout << "nC2 = " << c2++.get_count(); // c2 = 3
return 0;
}

One more example to overloading unary minus.

```
#include <iostream>
```

```
using namespace std;

class subtract

{

          int a;

          int b;

     public:

          void getdata(int x, int y)

          {

               a = x; b = y;

          }

          void putdata()

          {

               cout<< endl << "A  = " << a <<"B = " << b;

          }

          void operator -()

          {

               a = -a;    b = -b;
```

```
        }
};
int main()
{
    subtract s;
    s.getdata(34, -6);
    cout << endl << "S : ";
    s.putdata();
    -s;
    cout << endl << "S : ";
    s.putdata();
    return 0;
}
```

4. Overloading Binary Operators

But operators can be overloaded just as easily as unary operators. We will look at examples that overload arithmetic operators, comparison operators, and

arithmetic assignment operators.

We have just seen how to overload a unary operator. The same mechanism can be used to overload a binary operator.

```
// Overloading + operator
```

```
#include <iostream>
using namespace std;
class time
{
      int hh;  int mm;  int ss;
  public:
      time( ) { }
      time(int h, int m, int s)
      {
          hh =h; mm = m; ss = s;
      }
      void disp_time()
      {
          cout << endl << hh<< " : "
          << mm << " : " << ss;
      }
      time operator+(time);
};

time time::operator+(time t)
{
    time temp;
    temp.hh = hh + t.hh;
    temp.mm = mm + t.mm;
    temp.ss = ss + t.ss;
```

```
    return temp;
}

int main()
{
    time t1(12,1,24) , t2(5, 23, 45), t3;
    t3 = t1 + t2;
    t3.disp_time();
    return 0;
}
```

5. Operator Overloading with Strings

C/C++ deals with strings quite differently; we never copy, concatenate, or compare strings using operators like other languages. C/C++ has built functions to perform the above operations. But C++ provides the facility to do every thing on strings using operators. That means we have to provide extra responsibility to operators to perform such things.

The following example demonstrates the comparison between two strings using comparison operator ==.

```
// Program to compare two strings using operator overloading

#include <string.h>
#include <stdio.h>
#include <iostream>
using namespace std;

enum boolean{ false, true };

class string
{
   char *str;
```

```
    public:
        string() { *str = NULL; }
        string(char *s) { str = s; }
        int operator ==(string ts)
        {
            if (strcmp(str, ts.str) >= 0)
                return true;
            else
                return false;
        }
};

int main()
{
    string s1("Computer");
    string s2("Computers");

    if(s1 == s2)
        cout << "Equal";
    else
        cout << "Not Equal";

    return 0;
}
```

```
// concatenation of two strings

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

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

 class string
 {
       char *str;
   public:
       string()
       {
           str = new char[30] ;
           *str = NULL;
       }
       string(char *s) { str = s; }
       string operator +(string ts)
       {
           string t;
           strcat(t.str, str);
           strcat(t.str, ts.str);
           return t;
       }
       void putstring()
       {
           cout << endl << str;
       }
};

int main()
{
    string s1("Computer");  string s2("Institute");
    s1.putstring();  s2.putstring();
    string s3;

    s3 = s1 + s2;
```

```
    s3.putstring();
    return 0;
}
```

Ref: Object-oriented Programming in Turbo C++: Robert Lafore

*16. Step-by-Step C/C++ --- C++ Programming - Polymorphism**Polymorphism*

1. Function Overloading
2. Polymorphism
3. Types of polymorphism
4. Normal member functions accessed with pointers
5. Virtual Function
6. Pure Function
7. Assignment and Copy-Initialization
8. The COPY Constructor
9. ˜this(TM) Pointer

1. Function Overloading

If a function with its name differed by arguments behavior is called functions polymorphism or function overloading.

```
// An example program to demonstrate the use of function overloading

#include <iostream>
 using namespace std;
 void  printline()
 {
     for(int i=0;i<=80; i++) cout << "-";
 }

 void printline(int n)
```

```
{
    for(int i =0 ;i<=n;i++) cout << "-";
}

void printline(int n,char ch)
{
    for(int i=0;i<=n; i++) cout << ch;
}

int main()
    {
        printline();
        printline(5);
        printline(10, ~*(TM));
        return 0;
    }
```

Polymorphism

Polymorphism is one of the crucial features of OOP. It simply means one name, multiple forms. We have already seen how the concept of polymorphism is implemented using overloaded functions and operators. The overloaded member functions are selected for invoking by matching arguments, both type and number. The compiler knows this information at the compile time and therefore compiler is able to select the appropriate function for a particular call at the compile time itself. This is called ***early binding*** or ***static binding*** or ***static linking***. Also known as ***compile time polymorphism***, ***early binding*** simply means that an object is bound to its functions call at compile time.

Now let us consider a situation where the function name and prototype is the same in both the base and derived classes. For example, considers the following class definitions.

```
#include <iostream>

using namespace std;
```

```
class   A

{

     int x;

     public : void show();

};



class   B : public A

{

     int y;

     public :  void show();

};



int main()

{

     B b;

     b.show();
```

```
    return 0;

}
```

How do we use the member function ***show( )*** to print the values objects of both the classes A and B ? Since the prototype of ***show( )*** is the same in the both places, the function is not overloaded and therefore static binding does not apply. In fact, the compiler does not know what to do and defers the decision.

It would be nice if the appropriate member function could be selected while the program is running. This is known as ***runtime polymorphism***. How could it happen? C++ supports a mechanism known as ***virtual*** function to achieve runtime polymorphism. At runtime, when it is known what class objects are under consideration, the appropriate version of the function is called.

Since the function is linked with a particular class much later after the compilation, this process is termed as ***late binding***. It is also known as ***dynamic binding*** or ***dynamic linking*** because the selection of the appropriate function is done dynamically at runtime.

3. Types of Polymorphism

Polymorphism is of two types namely.

1

Compile time polymorphism
Or Early binding
Or Static binding
Or Static linking polymorphism.

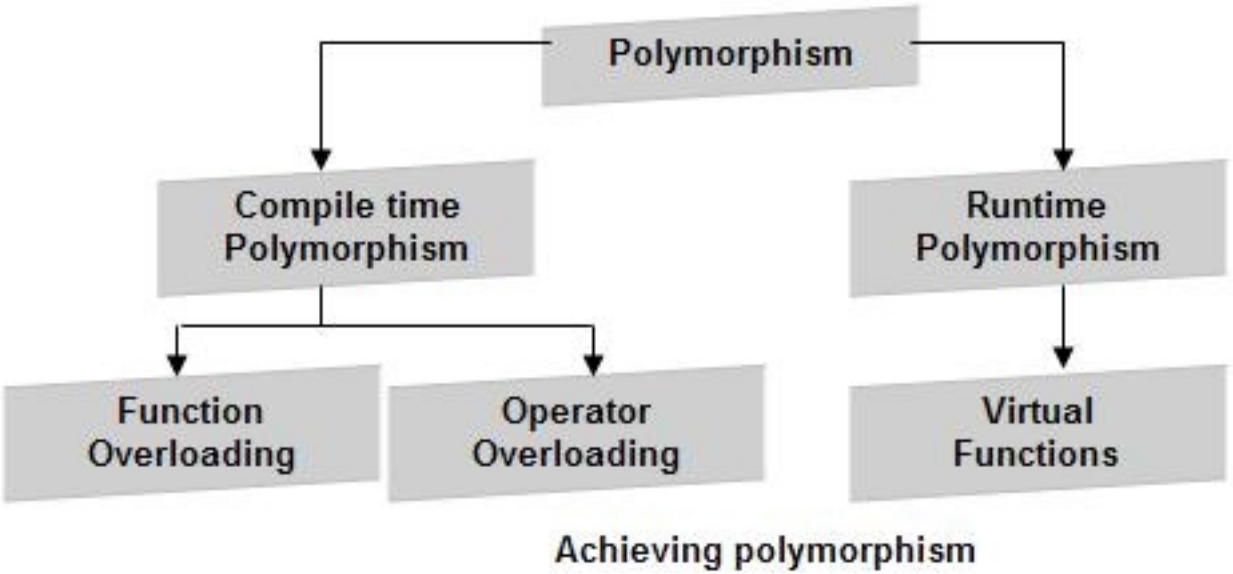
An object is bound to its function call at compile time.

2

Runtime polymorphism
Or late binding
Or Dynamic binding

Or Dynamic linking polymorphism.

The selection and appropriate function is done dynamically at run time.



Achieving polymorphism

Dynamic binding is one of the powerful features of C++. This requires the use of pointers to objects. We shall discuss in detail how the object pointers and virtual functions are used to implement dynamic binding.

4. Normal Member Functions Accessed with Pointers

The below program consist of a base class

```
/*
Normal
 functions accessed from pointer */
```

```
 /* Polymorphism with classes (without using  VIRTUAL polymorphism */
 #include  <iostream>
 using namespace std;

 class  BASE
      {
     public :
        void disp() { cout  << "nYou are in BASE class "; }
      };

class  DERIVED1 : public BASE
      {
     public :
        void disp() { cout  << "nYou are in DERIVED1 class"; }
      };

 class  DERIVED2 : public BASE
      {
     public :
        void disp() { cout  << "nYou are in DERIVED2 class"; }
      };

int  main()
      {
      DERIVED1 d1;            // Object of derived class 1
      DERIVED2 d2;            // Object of derived class 2
      BASE *b;                // pointer to base class

      b=&d1;                  // Assign address of d1 in pointer b
      b->disp();              // call to disp()
           b=&d2;                   // Assign address of d2 pointer b
      b->disp();              // call to disp()
      return 0;
```

```
    }
```

The above program demonstrates:
- ¢ A BASE class
- ¢ DERIVED1, DERIVED2 classes derived from BASE
- ¢ Derived classes objects (d1,d2)
- ¢ BASE class pointer *b

Output

```
You are in BASE class
   You are in BASE class
```

5. Virtual Function

Virtual means ***existing in effect but not in reality.***
A member function can be made as virtual function by preceding the member function with the keyword ***virtual.***

```
/*  Polymorphism with Classes (Virtual  polymorphism)  */
```

```
#include <iostream>
      using namespace std;
      class B
      {
          public :
              void show(){  cout << "nclass B method Show()  "; }
              virtual void disp() { cout << "nclass B method disp()"; }
 };
```

```
class D : public B
    {
        public :
            void show(){cout << "nclass D method Show()  "; }
            void disp(){ cout << "nclass D method disp()";  }
    };

int main()
    {
        D d1;
        d1.show();
        d1.disp();      // Base class member

    B b;
        D d;
        B *Bptr;
        Bptr = &b;
        Bptr->show();
        Bptr->disp();    //   Base class member

    Bptr=&d;
        Bptr->show();   // derived class members
        Bptr->disp();   // Base class member
        return 0;
    }
```

Output

```
class D method Show()
   class D method disp()
```

6. Pure Function

A function defined in a base class and has no definition relative to derived class is called pure function. In simple words ***a pure function is a virtual function with no body.***

```
#include <iostream>

using namespace std;

class B

{

     public :

          void show(){  cout << "\nclass B method Show()  "; }

          virtual void disp() = 0;  // pure virtual function

};



class D : public B

{

     public :

          void show(){cout << "\nclass D method Show()  "; }

          void disp(){ cout << "\nclass D method disp()";  }

};
```

```
int main()
{
     D d1;
     d1.show(); // O/P : Class D method show()
     d1.disp();     // O/P : Class D method disp()

     D d;
     B *Bptr;

     Bptr=&d;
     Bptr->show(); // O/P : Class B method show()
     Bptr->disp(); // O/P : Class D method disp()
     return 0;
}
```

Bptr -> show() is the default executable function from Base

Bptr -> disp() is the default executable function from Base but it is declared as a virtual pure function so at runtime Derived class's ***disp()*** will be called.

```
/* Program to demonstrate the advantage of pure virtual functions  */


#include <iostream>

using namespace std;

enum boolean { false, true };

class NAME

{

     protected :  char name[20];

     public  :

          void getname()

          {  cout << "Enter name   :"; cin >> name; }



          void showname()

          {  cout << "\nName is  "<< name; }


          boolean virtual isGradeA() = 0;  // pure virtual function

};
```

```
class student : public NAME

{

     private : float avg;

     public :

          void getavg()

          {

               cout << "\nEnter Student Average :";

               cin >> avg;

          }


          boolean isGradeA()

          { return (avg>=80) ? true : false ; }

};


class employee : public NAME

{

     private : int sal;

     public :
```

```
          void getsal()
          { cout << "\nEnter salary  "; cin >> sal;  }

          boolean isGradeA()
          { return (sal>=10000) ? true : false ; }
};

int main()
{
     NAME *names[20]; // no of pointer to name
     student *s;  // pointer to student
     employee *e;  // pointer to employee
     int n = 0;  // no of Names on list
     char choice;
     do{
          cout << "Enter Student or Employee (s/e)  ";
          cin >> choice;
```

```
        if(choice=='s')
        {
             s = new student; // make a new student
             s->getname();
             s->getavg();
             names[n++]=s;
        }
        else
        {
             e = new employee; // make a new employee
             e->getname();
             e->getsal();
             names[n++]=e;
        }
        cout << "Enter another  (y/n)  ?"; // do another
        cin >> choice;
    } while(choice=='y');
```

```
    for(int j=0; j<n; j++)
    {
         names[j]->showname( );
         if(names[j]->isGradeA( )==true)
              cout << "He is Grade 1 person";
    }
    return 0;
}
```

7. Assignment and Copy-Initialization

The C++ compiler is always busy on your behalf, doing things you can(TM)t be bothered to do. If you take charge, it will defer to your judgement; otherwise it will do things its own way. Two important examples of this process are the ***assignment operator and the copy constructor.***

You(TM)ve used the assignment operator many times, probably without thinking too much about it. Suppose ***a1*** and ***a2*** are objects. Unless you tell the compiler otherwise, the statement.

a2 = a1; // set a2 to the value of a1

Will cause the compiler to copy the data from ***a1,*** member-by-member, into ***a2.*** This is the default action of the assignment operator, =.

You(TM)re also familiar with initializing variables, initializing an object with another object, as in

alpha a2(a1); // initialize a2 to the value of a1

Causes a similar action. The compiler creates a new object, ***a2,*** and copies the data from ***a1,*** member-by-member, into ***a2***. This is the default action of the copy constructor.

Both these default activities are provided, free of charge, by the compiler. If member-by-member copying is what you want, you need take no further action. However, if you want assignment of initialization to do something more complex, then you can override the default functions. We(TM)ll discuss the techniques for overloading the assignment operator and the copy constructor separately.

Overloading the Assignment Operator

```
// Overloading the Assignment ( = ) Operator

#include <iostream>

using namespace std;

class alpha

{

     private:

          int data;

     public:

          alpha() { }     // no-arg constructor

          alpha( int d )

          { data = d; }  // one-arg constructor

          void display()
```

```
          { cout << data; }     // Display data

          alpha operator =(alpha & a)  // overloaded = operator

          {

               data = a.data;   // not done automatically

               cout << "\n Assignment operator invoked ";

               return alpha(data);

          }

};


int main()

{

     alpha a1(37);

     alpha a2;

     a2 = a1;     // Invoke overloaded =

     cout << "\n a2 = ";  a2.display();   // display a2


     alpha a3 = a2;     // does NOT invoke =
```

```
    cout << "\n a3 = "; a3.display();  // display a3

    return 0;

}
```

Output:

a2 = 37
a3 = 37

8. The COPY Constructor

As we discussed, we can define and at the same time initialize an object to the value of another object with two kinds of statement:

alpha a3(a2); // Copy initializing
alpha a3 = a2; // copy initialization, alternate syntax

Both styles of definition invoke a ***copy constructor:*** that is, a constructor that copies its argument into a new object. The default copy constructor, Which is provided automatically by the compiler for every object, performs a member-by-member copy. This is similar to what the assignment operator does; the difference is that the copy constructor also creates also creates a new object.

The following example demonstrates the copy constructor.

```
#include <iostream>

using namespace std;
```

```
class alpha

{

     private :

          int data;

     public:

          alpha( ) { }    // no-args constructor

          alpha(int d) { data = d; }  // one-arg constructor

          alpha(alpha& a)  // copy constructor

          {

               data = a.data;

               cout << "\nCopy constructor invoked";

          }

          void display( )

          {  cout << data; }

          void operator = (alpha& a)   // overloaded = operator

          {

               data = a.data;
```

```
            cout << "\nAssignment operator invoked";

        }

};


int main()

{

    alpha a1( 37 );

    alpha a2;


    a2 = a1;    // invoke overloaded =

    cout << "\na2 = "; a2.display();  // display a2



alpha a3( a1 );    // invoke copy constructor
```

```
    //  alpha a3 = a1;      // equivalent definition of a3

    cout << "na3 = "; a3.display();  // display a3
    return 0;
}
```

The above program overloads both the assignment operator and the copy constructor.

The overloaded assignment operator is similar to that in the past example.

9. ˜this(TM) Pointer

C++ uses a unique keyword called ***this*** to represent an object that invokes a member functions. ***This*** is a pointer that points to the object for which ***this*** function was called.

This pointers simply performs make task ***to return object it self.***

The following program defines ***i, j*** objects and ***i*** is assigned with the value of 5 and the entire object of ***i*** is assigned by its member function to ***j***

```
#include <iostream>

using namespace std;

class A

{

     int a;

     public:

          A() { }

          A(int x)

          {  a = x; }

          void display()

          {
```

```
            cout << a;

        }

        A get()

        {




 return *this;
```

// Return it self

}

};

int main()

{

A i(5);

A j;

j = i.get();

j.display();

return 0;

}

Ref: Object-oriented Programming in Turbo C++: Robert Lafore