Programming

in C

PROGRAMMING IN C

By **PANKAJ GILL**

pankajsinghgill@gmail.com

ABOUT

Hello all, I am **Pankaj Gill**. Currently I am pursuing my B.Tech from Computer Science & Engineering, Vaish College of Engineering, Rohtak. I wrote these study notes when I am learning **Programming in C** from **Mr. Pawan Zood(HCL CDC, Rohtak)**. When I am writing it, I keep some things in my mind, like the material that is going to be written are not so hard and difficult to understand. I am declaring that I didn't used any illegal material and I am not violating any law by sharing this on internet. I hope this will be proved as a great help in your studies and learning. This copy is free to share and distribution only for educational purpose. No charges should be applied for the copy of *Programming in C* by *Pankaj Gill*. I hope you will enjoy your learning. I would like to Thanks *Mr. Pawan Zood* for inspiring me to write this handbook and for helping me in making it possible.

Programming in C by Pankaj Gill is for revision purpose and it cannot be get too much helpful for the beginners. I hope you will enjoy learning with me. Good luck with Programming in C.

If you want to contact me, you can reach me at *pankajsinghgill@gmail.com.*

PANKAJ GILL

INDEX

1 PROGRAMMING IN C

A program is a set of instructions executed by computer in a sequence order.

'Or'

A program is a definite a plan prepared to achieve problem solution.

It is designing, writing, testing of problems written using some language of computer. It is a process of designing writing, testing, marinating computer program according to user requirement.

CHARACTERISTIC OF A GOOD PROGRAM

1. **Efficiency** – the extent to which inherent language features support the expression of source code that enforces good software engineering principles.

   Support for modern software engineering methods is support that encourages the use of good engineering practices and discourages poor practices. Hence, support for code clarity, encapsulation, and all forms of consistency checking are language features that provide this support. Also, support for complexity management and construction of large systems and subsystems support software engineering tenets.

2. **Real-time support** – *the extent to which inherent language features support the construction of real-time systems*.

3. **Reliability** - the extent to which inherent language features support the construction of components that can be expected to perform their intended functions in a satisfactory manner throughout the expected lifetime of the product.

   Reliability is concerned with making a system failure free, and thus is concerned with all possible errors [Levesen 86]. System reliability is most suspects when software is being stressed to its capacity limits or when it is interfacing with resources outside the system, particularly when receiving input from such resources. One way that interfacing with outside resources occurs is when users operate the system. Reliability problems often surface when novices use the system, because they can provide unexpected input that was not tested. Interfacing with outside resources also occurs when the system is interfacing with devices or other software systems. Language can provide support for this potential reliability problem through consistency checking of data exchanged. Language can provide support for robustness with features facilitating the construction of independent (encapsulated) components which do not communicate with other parts of the software except through well-defined interfaces. Language may also provide support for reliability by supporting explicit mechanisms for dealing with problems that are detected when the system is in operation (exception handling). Note that poor reliability in a safety-critical portion of the software also becomes a safety issue.

4. **Portability** - the extent to which inherent language features support the transfer of a program from one hardware and/or software platform to another.

   To make software readily portable, it must be written using non-system-dependent constructs except where system dependencies are encapsulated. The system dependent parts, if any, must be re-accomplished for the new platform, but if those parts of the software are encapsulated, a relatively small amount of new code is required to run the software on the new platform. Language support for portability can come from support for encapsulation, and it can also come from support for expressing constructs in a non-system-dependent manner. Language standardization has a significant impact on portability because non-standard language constructs can only be ported to systems that support the same non-standard constructs—for example, if both systems have compilers from the same vendor. In some circles, the issue of existing compatible support products, including compilers, on many different platforms is considered in the concept of portability. Note that a language's portability characteristics can be severely compromised by poor programming practices.

5. **Reusability**– *the extent to which inherent language features support the adaptation of code for use in another application*.

   Code is reusable when it is independent of other code except for communication through well-defined interfaces. This type of construction can occur at many levels. It is very common, for example, to reuse common data structures, such as stacks, queues, and trees. When these have been defined with common operations on the structures, these *abstract data types* are easy to reuse. When reusing larger portions of code, the biggest issue for reusability is whether the interfaces defined for the code to be reused are compatible with the interfaces defined for the system being created. This is significantly facilitated by the definition of software architecture for the domain of the system under construction. If those defining the components to be reused are aware of the architecture definition, then they can follow the standard interfaces defined in the architecture to ensure the code is reusable for other systems using the same architecture. Reuse at any level can be facilitated by language features that make it easy to write independent (encapsulated) modules with well-defined interfaces.

6. **Clarity** - The extent to which inherent language features support source code that is readable and understandable and that clearly reflects the underlying logical structure of the program.

   Most of the life cycle cost of a software system (usually between 60% and 80%) will come during the time after its initial development has been completed [Schach 93] [Sommerville 89]. This includes all efforts to change the software, whether it is to fix problems or to add new capabilities. Regardless of the purpose, changing the software implies that a significant cost will be associated with understanding the program and its structure, since this is a necessary first step before any changes can be made. Although it is always possible to use techniques to make a program easier to understand, language support for source code clarity can facilitate this process considerably. Note that clarity is a readability issue, not necessarily an ease of use issue. It is not unusual for languages with good readability to be somewhat more verbose than less readable languages.

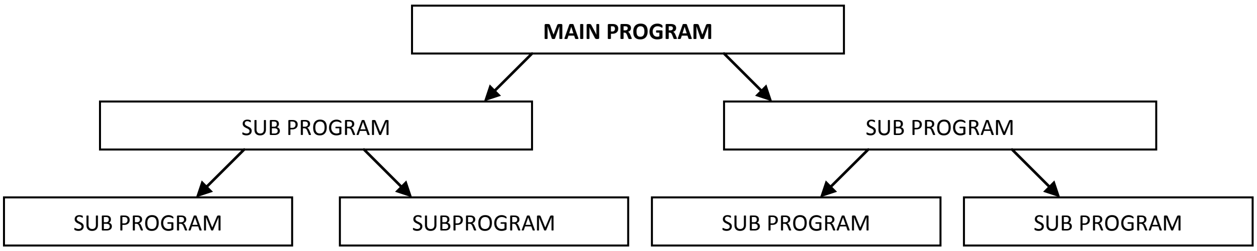
7. **Standardization** – the extent to which the language definition has been formally standardized (by recognized bodies such as ANSI and ISO) and the extent to which it can be reasonably expected that this standard will be followed in a language translator.

   Most popular languages are standardized through at least ANSI and ISO, but an important issue here is that the language definition that is supported by a compiler product may not be that which is standardized. Most languages have evolved in a manner that has produced a proliferation of different dialects before the language was standardized, and the result has been that most compiler products support non-standard features from these dialects in addition to the standard language. Some of these products also support a mode that enforces the use of only the standard language constructs, but programmer discipline is still required to use this mode.

Program Design

When we follow OOP in C++ we first design our classes based on the domain model or the design. While in C where we don't follow OOP we try to address problem at a higher level and then try designing how this can be broken in smaller modules. The difference is between conventional structured program design and OOP, and hence between bottom up and top down.

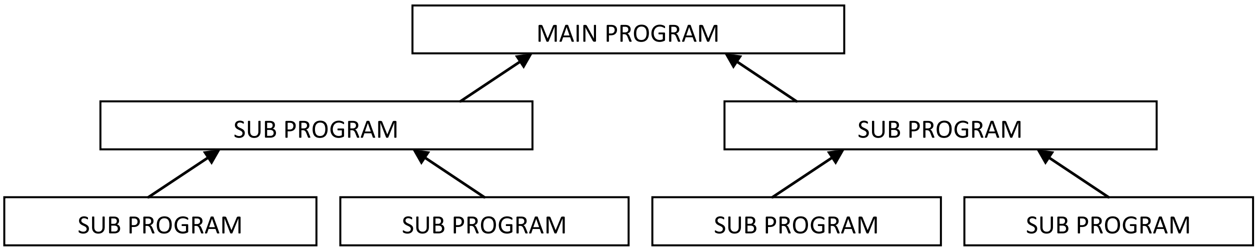
1. *Top-Down Approach* **-** you write first the main function, that calls stubs, then you subdivide these stubs in smaller stubs until a real work has to be done, that you code in the final files.



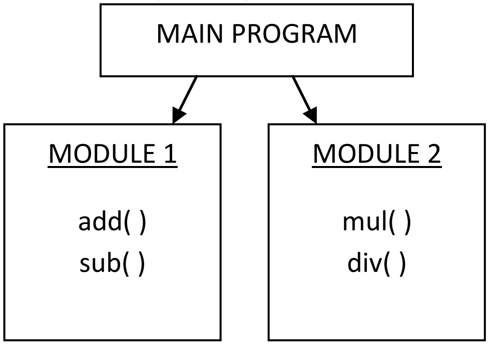
2. *Bottom-Up Approach* - In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed. This strategy often resembles a "seed"

model, whereby the beginnings are small, but eventually grow in complexity and completeness.

Object-oriented programming (OOP) is a programming paradigm that uses "objects" to design applications and computer programs.

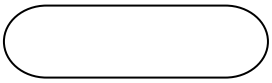


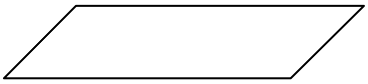
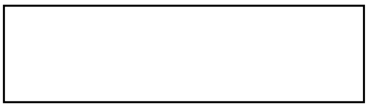
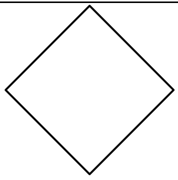
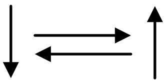
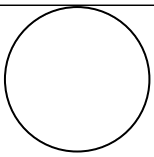
3. *Modular Approach* -Modular design is the process in which you take a large problem and break it down into smaller problems, and address each smaller problem individually. In terms of C programming, it would mean taking a large programming task and breaking it down into several smaller logically separate tasks. Each of these smaller tasks would most likely be represented as a function in the program.



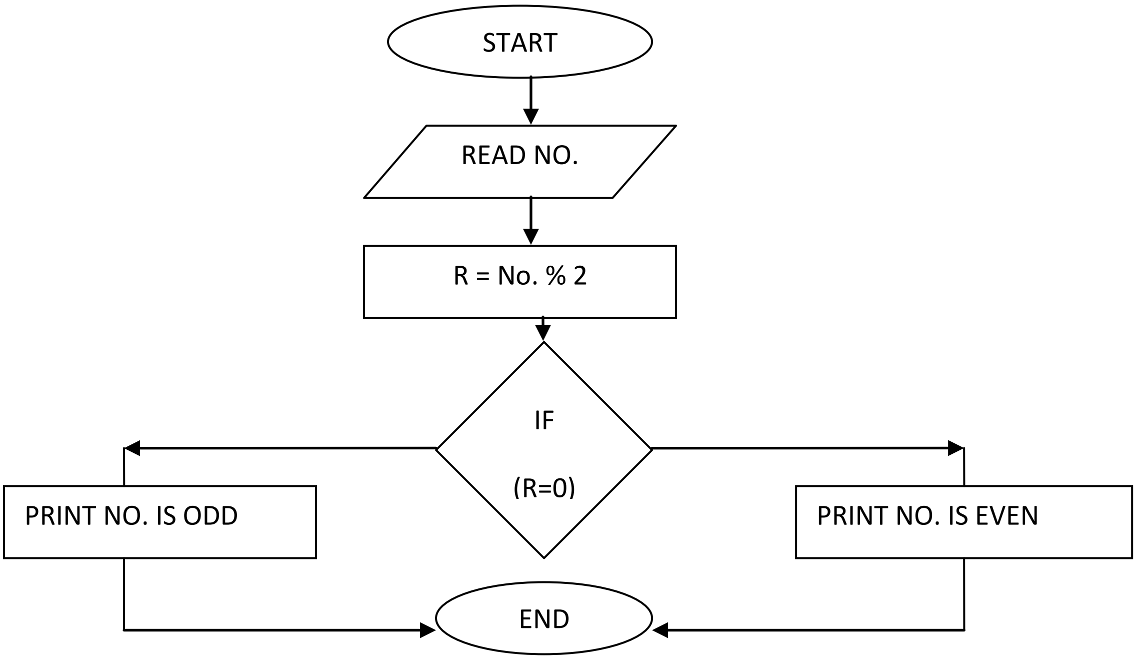
Flow Chart

It is a pictorial representation of problem. A flowchart is prepared using some set of shapes e.g. box, ellipse, rectangle etc. The boxes are different in shapes; the boxes are joined using lines. The lines are arrow headed. The process of drawing flowchart is called flowcharting.

Shape	Description
	Start/End

	Input/Output
	Processing
	Decision
	Flow Lines
	Connects

E.g. Draw a flowchart to find whether number is even or odd.



2. C LANGUAGE & STRUCTURE OF PROGRAM

Origin of C

Martin	– BPCL (Basic Programming Computer Language)		– 1967
Ken Thompson	–	B	– 1970
Dennis Ritchie	–	C	– 1972
Bjarne Stroustrup	–	C++	– 1983

C++ is based on *C & Simula 67*

Generation of Language

1. *1st Generation* – (Machine Language) Low Level Language
2. *2nd Generation* – (Assembly Language) Low/Med. Level Language
3. *3rd Generation*– (C, C++) High Level Language
4. *4th Generation* – (SQL, QBE) Very High Level Language
5. *5th Generation* – (Artificial Intelligence AI) Neural N/w

Why C is Popular

C has several features due to which it is popular –

1. C is portable language – It is not hardware dependent.
2. Easy to learn, it has only 32 keywords.
3. C is rich in function; functions are defined in header files.
4. C is rich in built in as well as user definable data type.
5. C has feature of high level as well as low level language.
6. C support GUI based primitive line, circle, rectangle can be drawn using C as well as CUI.
7. C provides facility of dynamic allocation and D allocation of memory, Very fast. Static and Dynamic
8. Major parts of OS like Linux, UNIX, and Windows are written in C.
9. C Program can be designed to interact with ROM BIOS.
10. Facility of Structure in C Language makes DBMS like app.
    1. Maintenance of record of Students. 2. Maintenance of record of Books.

Error

The occurrence of an incorrect result produced by a computer. It is the difference between the Theoretical Value to Actual Value i.e., Theoretical Value – Actual Value.

Debugging

Debugging is the process of locating and fixing errors (known as bugs), in a computer program such as a program compiled in C. When we find out the error in our program and then make the required correction in the source code or in our program than this process is known as Debugging.

Types of Error

Basically there are three types of errors:-

1. **Compile Time Error -**Compile-time error occurs when you write your code with syntax errors (miss a {, or a [ ).
2. **Run-time Error -** It occurs when your program has logic errors, thus has unexpected output.
3. **Linking Error -** Linkers Errors are relatively rare and usually result from misspelling the name of a C library function. In this case, you get an Error: undefined symbols: error message, followed by the misspelled name (preceded by an underscore). Once you correct the spelling, the problem should go away.

STRUCTURE OF A C PROGRAM

# is the preprocessor

# include directory

`<stdio.h>` is header file

To add a single line comment in the Document Section we will use // in front of our comment and to add multiple line comment we use /* in the front and */ in the end of our comment.

Document Section

```
/**********************************************
* --------Heading Comments--------      *
***********************************************/
```

Link Section

```
#include<stdio.h>

#include<conio.h>
```

Definition Section/Global Declaration

```
#define pie 3.14;

void Fun1();

int a,b;
```

Main Function Section

```
main( )

{

printf(“Hello World & Welcome to C”);

}
```

Sub Program Section

Function 1
Function 2

Function N

My First Program

Write a Program to print “Hello World & Welcome to C”.

```
/*******************************************************
* This program is made for printing “Hello World & Welcome to C” *
* The purpose of this program is for demonstrating a simple program *
**
* Usage:*
* Runs the program and the message will appear.*
*******************************************************/

#include<stdio.h>
```

```
main( )
{
      printf(“Hello World & Welcome to C”)
}
```

To compile the program you have to press *Alt+F9* when you are still in the program.

To run the program or to execute it you have to press *Ctrl+F9*.

Ques. Write a Program to get the Roll No. and Fee of a student and showing it afterwards using integer functions in your program.

```
/*****************************************************
*  This  Program  is  made  for  taking  integers  for  Roll  No.  and  Fee  *
* The Program will take intergers for Roll No. & Fee and will show it later *
*                                                                           *
*                                    Usage:                                 *
* Run the Program tells the numbers and it will show you the given values.  *
*****************************************************/

#include<stdio.h>
#include<conio.h>

void Main( )
{
      int Roll No, Fee;
      clrscr( );
      printf(“Enter your Roll No.”);
      scanf(“%d”,& Roll No);
      printf(“Enter your Fee”);
      scanf(“%d”,& Fee);
      getch( );
      printf(“Your Roll No. is %d and your fee is %d”,Roll No,Fee);
      getch( );
}
```

‘int’ function is used for telling the language how much integers we are using in the program.

‘printf’ is used to print a character string, supply the string (contained in double quotes) as the parameter to `printf`. This string is called the *format string*. The two-character sequence `\n` displays a newline, and the two-character sequence `\t` displays a tab.

‘getch’ function is used to get a character so the screen will not splash and you will notice the execution of the program. ‘getch’ function particularly is for taking a character from the user.

‘clrscr’ function is used for clearing the previous screen before printing the data.

- **How scanf() Consumes Input**

When scanf() is called, it skips over all leading white space (spaces, tabs, and newlines). Try recompiling and running scanfdemo. Each time it prompts for a number, try entering a bunch of newlines, spaces, and tabs before typing the number. The extra white space will have no effect. After scanf has skipped over white space, it reads until it finds something that doesn't belong in the type of number than it is looking for and then quits. It leaves any remaining input behind for the next call to scanf. Run scanfdemo, and when it prompts for the first number enter

1.2 3.4You will see than scanf reads the 1 as an integer, stopping when it encounters the decimal point (which can't be part of an integer). It later reads the .2 as a double, stopping when it encounters the white space. Then it reads the 3 as an integer, and the .4 as a double. Notice that if there is input left over from a previous call, scanf will use it.

Now run the program and enter x

None of the calls to scanf can get past the x, so the uninitialized values of the variables are displayed.

Be careful when you supply input to your programs that you only type in properly formatted numbers and white space. While it is possible to write programs that are more robust in the face of bad input, we will not be going into that topic.

- **Output with printf**

To print a character string, supply the string (contained in double quotes) as the parameter to printf. This string is called the *format string*. The two-character sequence \n displays a newline, and the two-character sequence \t displays a tab.

To print an int, embed the sequence %d in the format string, and include an integer expression as a second parameter. (The sequence %d is called a conversion specification.) The value of the expression will be displayed in place of the %d.

To print a double, embed one of the sequences %g, %f, or %e in the format string, and include a floating-point expression as a second parameter. The value of the expression will be displayed in place of the conversion specification. The way that the value will be displayed depends upon which of the three conversion specifications you use.

- The %g specification displays the number as a six-digit mantissa, with an exponent if necessary.
- The %f specification displays the number with six digits after the decimal point and no exponent.
- The %e specification displays the number using scientific notation, with one digit before the decimal point, six after the decimal point, and an exponent.

To display more than one number, include more than one conversion specification and more than one extra parameter.

3. TOKENS

The smallest entity in the language C which cannot get reduced any further is known as Tokens. There are five types of token present in our language C.

1. Identifier
2. Keyword
3. Literal 'or' constant
4. Operator
5. Punctuators

1. **Identifier:** The name of any variable, function, array, structure, union etc. is known as identifier. An Identifier is a sequence of letters and digits, but must start with a letter. Underscore (_) is treated as a letter. Identifiers are case sensitive. Identifiers are used to name variables, functions etc.

   Valid: Root, _getchar, __sin, x1, x2, x3, x_1, If

   *Invalid:*324, short, price$, My Name

   Rules for Identifiers: –

   i) The first letter must be alphabet or _.
   ii) ANSI standards recognize a length of 31st character. However the length should be not more than 8 characters.
   iii) Spaces are not allowed.
   iv) UPPER CASE and lower case are significant.
   v) No special symbols can be used. E.g., ~!@#$%^&*()
   vi) Keywords cannot be used as Identifiers.

2. **Keywords:** A keyword is sometimes referred to as a predefined or reserve words, is a sequence of characters that is reversed by compiler and has special meaning to the language. These are reserved words of the C language. For example **int, float, if, else, for, while, double, char, do, break, continue, goto, case, default, size of, long, return, switch, void, exit** etc. Fixed meanings and cannot be changed. There are 32 keywords in C & 48 keywords in C#.

3. **Literal 'or' Constant:** The data element whose value does not change during the program execution is called constant. It is also referred as Literals. There are 2 types of constant.
   i) Numeric –
      a) Integer Constant – Integers like -1, -2, -3, 0, 1, 2, 3 etc. are counted as integer constant.
      b) Floating Constant – The integers which are having decimal places are known as floating. For example, 1.123, 2.456, 3.789 etc.

   ii) Character –

a) Single Character – The characters like A, a, B, b, C, c etc. are known as single character constant.

b) String – A sequence of characters is known as string. For example, "Pankaj", "Gill".

4. **Operators:** Arithmetic operators like +, -, *, /, % etc. Logical operators like ||, &&,! Etc. and so on.

5. **Punctuators:** The brackets ( ), {}, [] and : ; , . are known as punctuators.

4. BACKSLASH FUNCTIONS “\” IN C

Those functions can be added to the ‘printf’ message for creating the desired result. Some of the essential functions are described below. These characters can be embedded in "printf" strings

\n	-	for a new line
\a	-	for alert sound or beep
\t	-	for adding tab space (horizontal tab)
\v	-	vertical tab
\b	-	for shifting the cursor to backward character by 1 character in the line
\r	-	for shifting the cursor to the beginning of the current line(carriage return)
\”	-	to print “ in the message
\'	-	single quote
\?	-	question mark
\\	-	to print \ in the message
\p	-	backspace
\f	-	formfeed
\0NN	-	character code in octal
\xNN	-	character code in hex
\0	-	null character

5. DATA TYPE & VARIABLE

Variable

A variable are data items whose values can change during program execution. The Value can be changed any no. of types.

'Or'

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

Rules for Variable Names

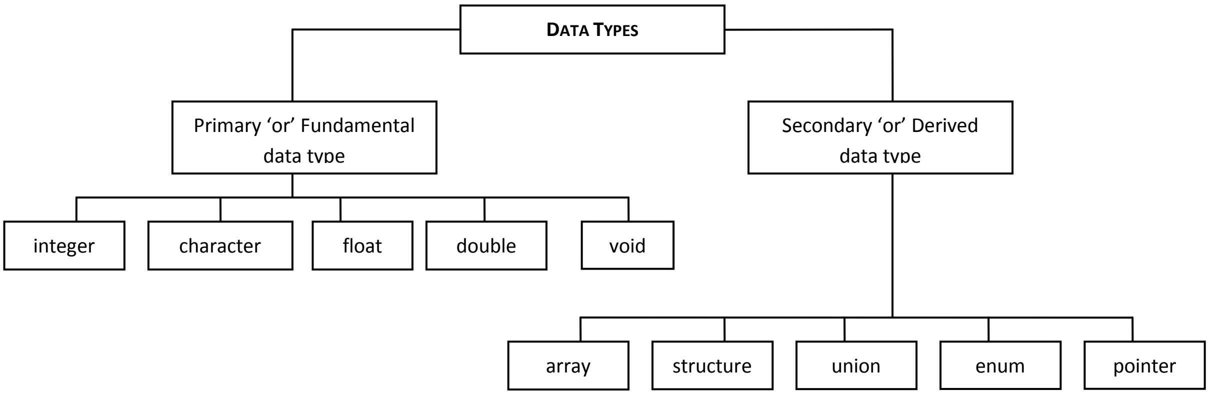
1. The Name can contain letters, digits, and the underscore character (_).
2. The first character of the name must be a letter. The underscore is also a legal first character, but its use is not recommended.
3. UPPER CASE and lower case does matters. Thus the names *COUNT* and *count* refer to two different variables.
4. C Keywords cannot be used as variable names. A keyword is a word that is part or the C language.
5. Variable length is taken as 8 character, depending on compiler it can be more.

Declaration of Variable Name

Data type	Variable name;
`int`	`a;`
`int`	`a, b, c, d, e, f;`
`float`	`a, b, c;`

Data Type

A Data Type is a type of data. A data type is a data storage format that can contain aspecific type or range of values. There are Five basic types of data types in C. Further it can be classified as.



Primary 'or' Fundamental Data Type

Exceed for type **void**, the basic data types may have various modifiers preceding them.

You use a *modifier* to alter the meaning of the base type to fit various situations more precisely. The list of modifiers is shown there.

- signed
- unsigned
- long
- short

You can apply the modifiers **signed, short, long** and **unsigned** to integer base types. You can apply **unsigned** and **signed** to characters. You may also apply **long** to **double**.

You use of **signed** on integers is allowed, but redundant because the default integer declaration assumes a signed number. The most important use of **signed** is to modify char in implementations in which **char** is unsigned by default.

The difference between signed and unsigned integers is in the way that the high order bit of the integer is interpreted. If you specify a signed integer, the computer generates code that assumes that the high-order bit of an integer is to be used as a **sign flag.** If the sign flag is 0, the number is positive; if it is 1, the number is negative.

In 16 bit integer numbers, the most significant bit represent the sign of integer. 0 fir positive and 1 for negative sign. Remaining 15 bits are used to represent the range of integer.

Signed integers are important for a great many algorithms, but they only have half the absolute magnitude of their unsigned relatives. For example, here is 32,767:

0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

If the high-order bit were set to 1, the number would be interpreted as -1. However, if you declare this to be an **unsigned int,** the number becomes 65,535 when the high order bit is set to 1.

The following table shows data and its corresponding range of values that the data type can store.

Type	Typical Size in Bits	Minimal Range
char	8	-127 to 127
unsigned char	8	0 to 255
signed char	8	-127 to 127
int	16 or 32	-32,767 to 32767
unsigned int	16 or 32	0 to 65,535
signed int	16 or 32	Same as int
short int	16	-32,767 to 32,767
unsigned short int	16	0 to 65,535
signed short int	16	Same as short int
long int	32	-2,147,483,647 to 2,147,483,647
signed long int	32	Same as long int
unsigned long int	32	0 to 4,294,967,295
float	32	6 digits of precision
double	64	10 digits of precision
long double	80	10 digits of precision

1. Integer: Integer data type store integer value, these are numeric value without any fractional part. It can be either positive or negative. They are 16 bit integers.

i) Short Signed Int:- It consume 2 bytes. Format specifier is %d.
ii) Short Unsigned Int:- It consume 2 bytes. Format specifier is %u.
iii) Long Signed Int:- It consumes 4 bytes. Format specifier is %ld.
iv) Long Unsigned Int:- It consumes 4 bytes. Format specifier is %lu.

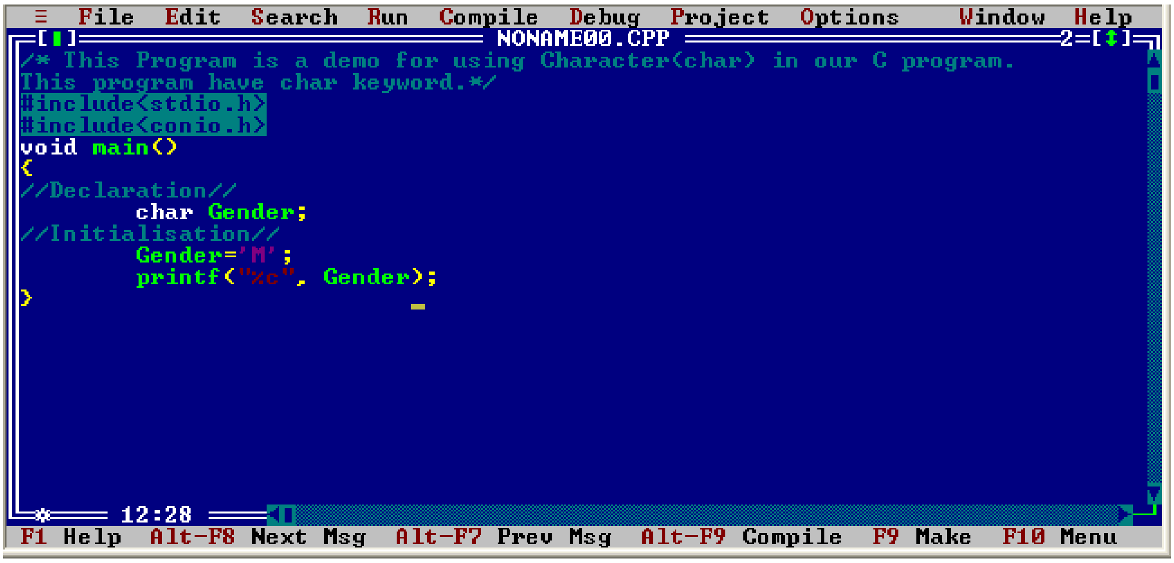
2. Character: For this data type, char keyword is used. It can used to store single character. Character can be alphabet or numeric data. It consume 1 byte of memory. For this format specifier %c is used.

2^n —> n(max.) = 8

Unsigned :- 2^8 — 256 — 0 to 255

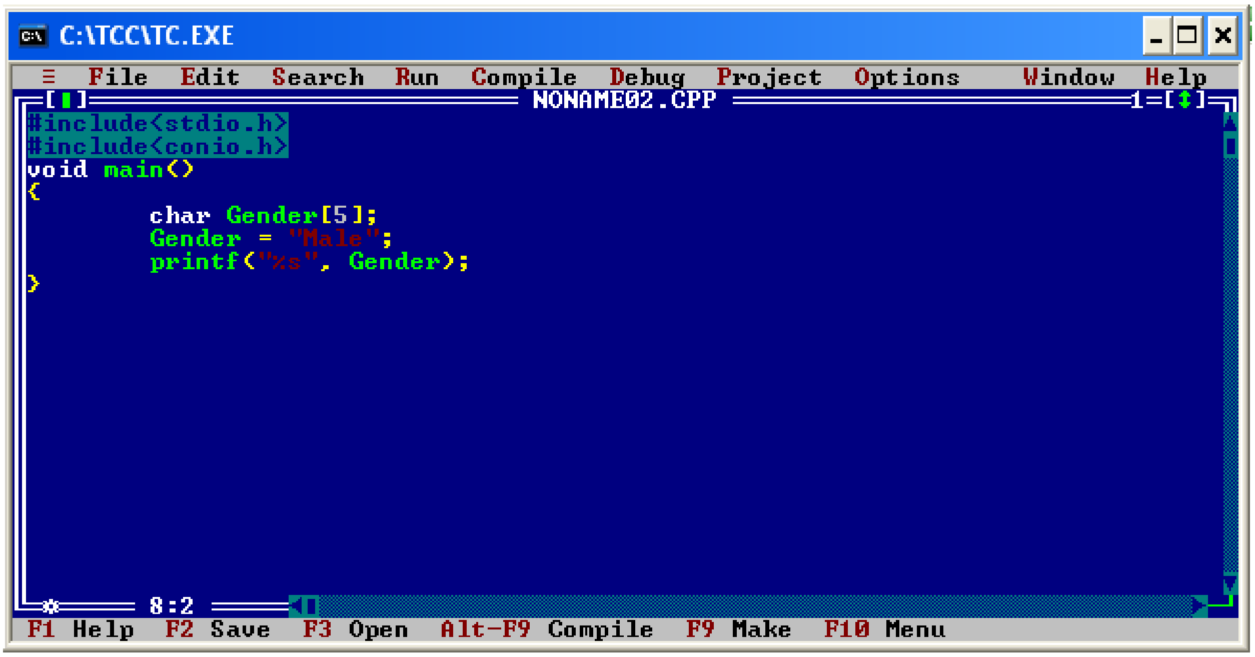
Signed :- 2^8 — 256 — -128 to 127

Sequence of characters is known as String. For String %s is used as specifier.


≡ File Edit Search Run Compile Debug Project Options Window Help
[■] NONAME00.CPP 2=[↕]
12:28
F1 Help Alt-F8 Next Msg Alt-F7 Prev Msg Alt-F9 Compile F9 Make F10 Menu

```
/* This Program is a demo for using Character(char) in our C program.
This program have char keyword.*/
#include<stdio.h>
#include<conio.h>
void main()
{
//Declaration//
        char Gender;
//Initialisation//
        Gender='M';
        printf("%c", Gender);
}
```

For using strings in program.


C:\TCC\TC.EXE
≡ File Edit Search Run Compile Debug Project Options Window Help
NONAME02.CPP
8:2
F1 Help F2 Save F3 Open Alt-F9 Compile F9 Make F10 Menu

```
#include<stdio.h>
#include<conio.h>
void main()
{
        char Gender[5];
        Gender = "Male";
        printf("%s", Gender);
}
```

3. Float: Floating point variable called real no. A number having fractional part is a floating point no. Floating point no. are defined in C using a keyword float. %f is used as specifier.

float a = 10.2057;

Floats are stored in 4 byte and accurate about 6 significant digits.

Range = 3.4e-38 to 3.4e+38

4. Double: Occupies 8 bytes, 10 significant digits and the specifier is %lf long double occupies 10 bytes and have 10 significant digits.

5. Void: void function is used to tell the C that the program is not returning any value.

6. OPERATORS & EXPRESSION

Expression

An expression is a formula consisting of 1 or more operators. The operation are represented by operators. For example, a + b;

There are two types of expressions:-

1. Simple Expressions – 5+6; 'or' a + b;
2. Complex Expressions – a * b/c + d;

Operators

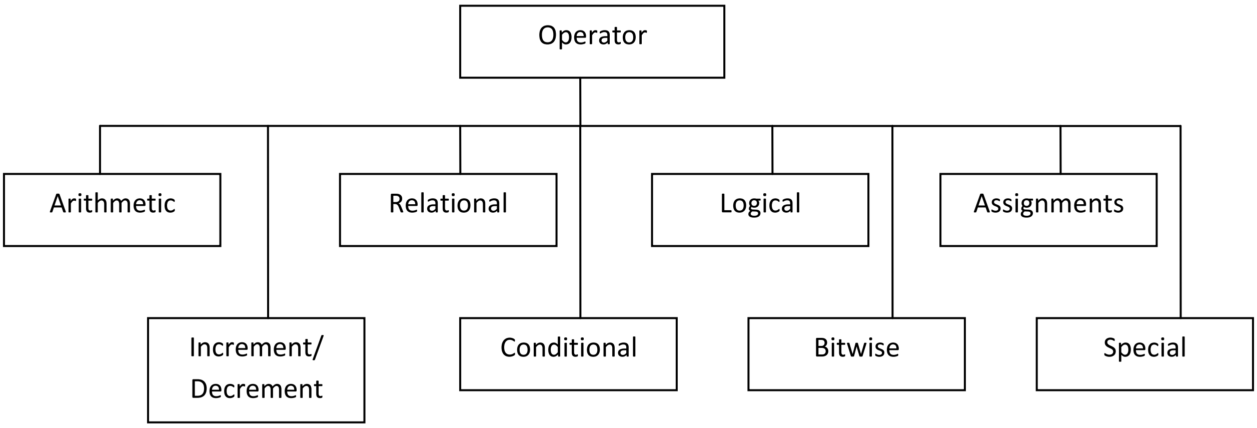
Operators specify operations to be performed. The operations produce resultant value using operators constant and variable are combined and expressions are obtained.

'OR'

An operator is a symbol that instructs C to perform some operation, or action, on one or more operands. An operand is something that an operator acts on.

```
b = a + 5;
```

Here =, + are opcode (operational code) and *b*, *a* and *5* are operands. The variable, constants those are combined by operators are called operands. There are basically 8 types of operators.



1. Arithmetic Operator
2. Relational Operator
3. Logical Operator
4. Assignments Operator
5. Increment/Decrement Operator
6. Conditional Operator
7. Bitwise Operator
8. Special Operator

From now, we will study briefly on the operators named above.

1. Arithmetic Operator: – These are also called binary operators. These operate on two or more operands. C provides 5 basic binary arithmetic operators as it is requires two values to calculate the final answer & 2 unary arithmetic operators that requires 1 operand. Arithmetic operators are also called as *Mathematical Operators*.

Binary Arithmetic Operator: **–** C's binary operators take two operands. The binary operators, which include the common mathematical operators found on a calculator, are listed in following table.

Operator	Symbol	Action	Example
Addition	+	Adds two operands	x + y;
Subtraction	-	Subtract the second operand from the first operand	x – y;
Multiplication	*	Multiplies two operands	x * y;
Division	/	Divides the first operand by the second operand	x / y;
Modulus	%	Gives the remainder when the first operand is divided by the second operand	x % Y;

Unary Arithmetic Operator: –The unary arithmetic operators are so named because they take a single operand. C has two unary arithmetic operators, listed in following table.

Operator	Symbol	Action	Example
Unary Plus	+	Multiply operand by +1	+a = 5 or a = -5 +a = 5 or +a = -5
Unary Minus	-	Multiply operand by -1	-a = 5 or -a = -5 a = -5 or a = 5

Example Program on Arithmetic Operator.

```
/* This program is a revision program for airthmetic operators
** This program is made by Pankaj Gill for revision purpose*/
#include<stdio.h>
#include<conio.h>
void main()
{
      int a,b;
      float;
//Here we used some operators and we are gonna work on them from now on//
//First we will perform airthmetic operations with integers//
      clrscr();
      printf("\t\t\a Checking Operators");
      printf("\n\nEnter Your 1st Integer a = ");
      scanf("%d",&a);
      printf("\nEnter Your 2nd Integer b = ");
      scanf("%d",&b);
      printf("\n\a\ta+b = %d",a+b);
      printf("\n\ta-b = %d",a-b);
      printf("\n\ta*b = %d",a*b);
      printf("\n\ta/b = %f",a/b);
      printf("\n\ta-b = %d",a%b);
      getch();
      printf("\n\n\nPress any key to quit");
      getch();
      printf("\a");
}
```

2. Relational Operator: – These are used to compare or determine relation among two operands. Operands can be either constant or variable. In the term relational operator, relational refrers to the relationship that values can have with one another. In the term logical operator, logical refers to the ways these relationships can be connected.

Operator	Symbol	Action	Example	Result
Equal	= =	Is op1 equal to op2	5 = = 5 5 = = 2	True (1) False (0)
Greater than	>	Is op1 is greater than op2	5 > 4 5 > 5	True (1) False (0)
Less than	<	Is op1 is less than op2	7 < 5 7 < 8	False (0) True (1)
Less than equal to	<=	Is op1 is less than or equal to op2	6 <= 6	True (1)

			6 <= 4	False (0)
Greater than equal to	>=	Is op1 is greater than or equal to op2	6 >= 5 6 >= 4	True (1) False (0)
Not equal to	!=	Is op1 is not equal to op2	5 != 4 5 != 5	True (1) False (0)

3. Logical Operator: – It is used to combine two or more relational expressions it is used in conditional expressional in C. 3 logical operators are defined.

Operator	Action	Example	Result
&&	AND	a=5, b=7, c= 9 (a<b) && (a<c) (a>b) && (a<c)	True (1) False (0)
| |	OR	(b<a) | | (a<c) (b<a) | | (c<b)	True (1) False (0)
!	NOT	!(a<b) !(a>b)	False (0) True (1)

The following shows the truth table for the logical operators. The truth table for the logical operators is shown below using 1's or 0's.

a	b	a && b	a || b	!a
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

4. Assignment Operator: – The assignment operator is the equal (=) sign. Its use in somewhat different from its use in regular math. If you write x = y; in C program, it doesn't mean "x is equal to y” Instead, it means “assign the value of y to x.” There are 5 more assignment operator when combined with arithmetic operators. All the assignment operators are defined below.

Operator	Symbol	Example	Result
Assignment Operator	=	a = 10;	
Addition Assignment Operator	+=	a+=15;	a=a+15
Subtraction Assignment Operator	-=	a-=10;	a=a-10
Multiplication Assignment Operator	*=	a*=2;	a=a*2
Division Assignment Operator	/+	a/=2;	a=a/2
Modulus Assignment Operator	%=	a%=;	a=a%2

5. Increment/Decrement Operator: – The increment and decrement operators can be used only with variables, not with constants. The operation performed is to add one to or subtract one from the operand. They are defined below in the table.

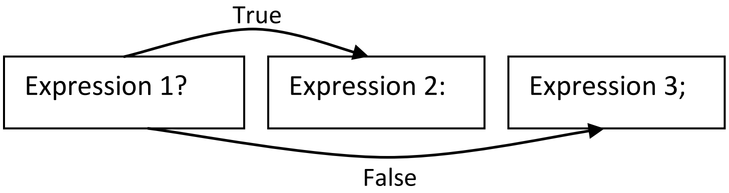
Operator	Symbol	Action	Example	Result
Increment Operator	++	Increments the operand by 1	++x, x++	1+x, x+1
Decrement Operator	- -	Decrements the operand by 1	- -y, y- -	1-y, y-1

++x is known as pre-increment operator and in this operator first the value will be incremented and later other operators can be applied.

x++ is known as post-increment operator and in this operator first the value will be assigned and later on the increment is performed.

Same goes for the decrement operator.

6. Conditional Operator: –The Conditional operator is used to apply condition whether true or false. The result will be as per calculated by the statement. Syntax for conditional operators is i.e. Expression1 ? Expression 2 : Expression 3;



Example Program of Conditional Operator :–

```
/* This program is a revision program for Conditional operators
** This program is made by Pankaj Gill for revision purpose
** This program is to find out the smaller number.*/
#include<stdio.h>
#include<conio.h>
void main()
{
      int a,b;
//Here we used some operators and we are gonna work on them from now on//
//First we will perform Conditional operations with integers//
      printf("\t\t\t Program to find out the smaller no.");
      printf("\n\nEnter First No. = ");
      scanf("%d",&a);
      printf("\nEnter Second No. = ");
      scanf("%d",&b);
      printf("\n\nPress Enter to Find out the Smaller No.");
      getch();
(a<b)?printf("\n\n\a\tFirst No. i.e. %d is smaller",a):printf("\n\n\a\tSecond No.
i.e. %d is smaller",b);
      getch();
      printf("\n\n\nPress any key to quit");
      getch();
      printf("\a");
}
```

7. Bitwise Operator: –Unlike many other languages, C supports a full complement of bitwise operators. Since C was designed to take the place of assembly language for most programming tasks, it needed to be able to support many operators that can be done in assembler, including operations on bits. Bitwise Operation refers to testing, setting, or shifting the actual bits in a byte or word, which correspond to the **char** and **int** data types and variants. You cannot use bitwise operations on **float, double, long double, void,** or other more complex types.

The following table lists the operators that apply to bitwise operations. These operations are applied to the individual bits of the operands.

Operator	**Symbol**
Left shift	<<
Right shift	>>

Exclusive OR	^
Compliment(tilde(	~
Bitwise AND	&
Bitwise OR	|

- *Left Shift & Right Shift Bitwise Operator* – The << operator shifts its first operand left by a number of bits given by its second operand, filling in new 0 bits at the right. Similarly, the >> operator shifts its first operand right. If the first operand is `unsigned`, >> fills in 0 bits from the left, but if the first operand is signed, >> might fill in 1 bits if the high-order bit was already 1. (Uncertainty like this is one reason why it's usually a good idea to use all `unsigned` operands when working with the bitwise operators.) For example, `0x56 << 2` is `0x158.` For example, if we shift 3 to bitwise left shift by 2, then it becomes 12 and then if we shift 12 to bitwise right shift by 2 then it becomes 3. i.e 3<<2=12, 12>>2=2

Integer	***Corresponding Bits***															
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
3<<2=12	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
12>>2=3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

Example programof Left Shift & Right Shift Operator : –

```
/*Program was a demo to show the use of bitwise operator******
**The Program was made by Pankaj Singh Gill for demo purpose*/
#include<stdio.h>
#include<conio.h>
void main()
{
      int a,b;
      clrscr();
      printf("\t\t\tChecking Left & Right Bitwise Operators");
      printf("\n\nEnter an Integer = ");
      scanf("%d",&a);
      printf("Enter a value to left shift the integer = ");
      scanf("%d",&b);
      printf("%d®%d=%d",a,b,a<<b);
      printf("\n\nEnter an Integer = ");
      scanf("%d",&a);
      printf("Enter a value to right shift the integer = ");
      scanf("%d",&b);
      printf("%d¯%d=%d",a,b,a>>b);
      getch();
}
```

- *Bitwise AND* - The bitwise AND operator is a single ampersand: &. A handy mnemonic is that the small version of the boolean AND, &&, works on smaller pieces (bits instead of bytes, chars, integers, etc). In essence, a binary AND simply takes the logical AND of the bits in each position of a number in binary form.

For instance, working with a byte (the char type):

```
01001000 &
10111000 =
--------
00001000
```

The most significant bit of the first number is 0, so we know the most significant bit of the result must be 0; in the second most significant bit, the bit of second number is zero, so we have the same result. The only time where both bits are 1, which is the only time the result will be 1, is the fifth bit from the left. Consequently,

```
72 & 184 = 8
```

- *Bitwise OR*- Bitwise OR works almost exactly the same way as bitwise AND. The only difference is that only one of the two bits needs to be a 1 for that position's bit in the result to be 1. (If both bits are a 1, the result will also have a 1 in that position.) The symbol is a pipe: |. Again, this is similar to boolean logical operator, which is ||.

```
01001000 |
10111000 =
--------
11111000
```

and consequently

```
72 | 184 = 248
```

Example Program of Bitwise And 'or' OR :–

```
/*Program was a demo to show the use of bitwise operator******
**The Program was made by Pankaj Singh Gill for demo purpose*/
#include<stdio.h>
#include<conio.h>
void main()
{
        int a,b;
        clrscr();
        printf("\t\t\tChecking Bitwise AND 'or' OR Operators");
        printf("\n\nEnter an Integer = ");
        scanf("%d",&a);
        printf("Enter a value for Bitwise AND to the integer = ");
        scanf("%d",&b);
        printf("%d&%d=%d",a,b,a&b);
        getch();
        printf("\n\nEnter an Integer = ");
        scanf("%d",&a);
        printf("Enter a value for Bitwise OR to the integer = ");
        scanf("%d",&b);
        printf("%d|%d=%d",a,b,a|b);
        getch();
        printf("Press any key to QUIT");
        getch();
}
```

- *The Bitwise Complement* - The bitwise complement operator, the tilde, ~, flips every bit. A useful way to remember this is that the tilde is sometimes called a twiddle, and the bitwise complement twiddles every bit: if you have a 1, it's a 0, and if you have a 0, it's a 1. This turns out to be a great way of finding the largest possible value for an unsigned number: unsigned int max = ~0;

Example program of Compliment Bitwise Operator :–

```
/*Program was a demo to show the use of bitwise operator******
**The Program was made by Pankaj Singh Gill for demo purpose*/
#include<stdio.h>
#include<conio.h>
void main()
{
      int a;
      clrscr();
      printf("\t\t\tChecking Compliment Bitwise Operator");
      printf("\n\nEnter an Integer = ");
      scanf("%d",&a);
      printf("The Compliment(~) of %d is %d",a,~a);
      getch();
      printf("\n\nPress any key to QUIT");
      getch();
}
```

- *Bitwise Exclusive-Or (XOR)* - There is no boolean operator counterpart to bitwise exclusive-or, but there is a simple explanation. The exclusive-or operation takes two inputs and returns a 1 if either one or the other of the inputs is a 1, but not if both are. That is, if both inputs are 1 or both inputs are 0, it returns 0. Bitwise exclusive-or, with the operator of a carrot, ^, performs the exclusive-or operation on each pair of bits. Exclusive-or is commonly abbreviated XOR. For instance, if you have two numbers represented in binary as 10101010 and 01110010 then taking the bitwise XOR results in 11011000. It's easier to see this if the bits are lined up correctly:

```
         01110010 ^
         10101010
         --------
         11011000
```

Example Program of Bitwise Exclusive-OR(XOR):-

```
/*Program was a demo to show the use of bitwise operator******
**The Program was made by Pankaj Singh Gill for demo purpose*/
#include<stdio.h>
#include<conio.h>
void main()
{
       int a,b;
       clrscr();
       printf("\t\t\tChecking Exclusive-OR(X-OR) Bitwise Operators");
       printf("\n\nEnter an Integer = ");
       scanf("%d",&a);
       printf("Enter a value to X-OR the integer = ");
       scanf("%d",&b);
       printf("%d^%d=%d",a,b,a^b);
```

```
        getch();
        printf("\n\nPress any key to QUIT");
        getch();
}
```

8. Special Operators: –There are 5 types of special operators.

Symbol	Operator name
.	Dot operator
,	Comma operator
&	Address operator
*	Pointer operator
sizeof( )	Size of ( )operator

*The & and * Pointer Operators* – A pointer (Pointers are discussed in-depth in later chapter) is the memory address of some object. A pointer variable is a variable that is specificially declared to hold a pointer to an object of its specified type. Knowing a variable's address can be of great help in certain types of routines. However, pointers have three main functions in C:

- They can provide a fast means of referencing array elements.
- They allow functions to modify their calling parameters.
- They support linked lists and other dynamic data structures.

The Comma (,) Operator –The comma operator combines together several expressions. The left side of the comma operator is always evaluated as void. This means that the expression on the right side becomes the value of the total comma-separated expression. For Example. a = (b=3, b+1);

The Dot (.) Operator – The .(dot) operator is used when working with a structure or union directly(Structure and Unions are discussed in details in later chapters).

Size of ( ) Operator – It is used to measure the size of memory occupied by any variable.

Revision Program for Operators :–

```
/* This program is made by Pankaj Gill for revision purpose
** This program is to find out the size of operator.*/
#include<stdio.h>
#include<conio.h>
void main()
{
      int a;
      float f;
      char c;
      double d;
      long double e;
//Here we used some operators and we are gonna work on them from now on//
//First we will perform Conditional operations with integers//
      clrscr();
      printf("\t\t\t Program to find out the bytes occupied");
      printf("\n\nThe size of integer a is %d bytes",sizeof(a));
      printf("\nThe size of float f is %d bytes",sizeof(f));
      printf("\nThe size of character c is %d bytes",sizeof(c));  printf("\nThe
size of double d is %d bytes",sizeof(d));
      printf("\nThe size of long double e is %d bytes",sizeof(e));
      printf("\n\n\t\aTotal size occupied is
%d",sizeof(a)+sizeof(f)+sizeof(c)+sizeof(d)+sizeof(e));
      getch();
      printf("\n\n\nPress any key to quit");
      getch();
      printf("\a");
}
```

Ques. Make a program to find out the area of a circle.

```
//This Program is made by Pankaj Singh Gill to find out area of the circle//
#include<stdio.h>
#include<conio.h>
void main()
{
      float r,y;
      y=3.14;
      clrscr();
      printf("\t\t\tArea of a Circle");
      printf("\n\nEnter the radius of cirlce = ");
      scanf("%f",&r);
      printf("\nArea of your Cirlce is %f",y*r*r);
      printf("\n\nPress any key to QUIT");
      getch();
}
```

Ques. To find the character of ASCII Code.

```
#include<stdio.h>
#include<conio.h>
void main()
{
      int a;
      char c,d;
      clrscr();
      printf("Enter the ASCII no. = ");
      scanf("%d",&a);
      c=a;
      printf("\nThe Character on this ASCII is %c",c);
      getch();
}
```

Ques. To find out the Simple Interest.

```
#include<stdio.h>
#include<conio.h>
void main()
{
      long p;
      int r,t;
      clrscr();
printf("Enter Principal Amount = 
Rs.\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b");
      scanf("%ld",&p);
      printf("Enter Rate of Interest =    %\b\b\b\b");
      scanf("%d",&r);
      printf("Enter Time period in Years = ");
      scanf("%d",&t);
printf("Simple interest on amount %ld with interest rate %d in %d years is 
%d",p,r,t,(p*r*t)/100);
      getch();
}
```

Ques. Make a program to find out Compound Interest.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
      int P,R,T;
      float CI,A;
      clrscr();
      printf("Enter P,R,T\n");
      scanf("%d%d%d",&P,&R,&T);
      A=P+pow(1+0.01*R,T);
      CI=A-P;
      printf("Amount is %f\n",A);
      printf("CI is %f",CI);
      getch();
}
```

Ques. Make a program to swap 2 integers by the help of third variable.

```
//This Program is made by Pankaj Singh Gill//
//Program to swap 1st variable to 2nd by the help of 3rd variable//
#include<stdio.h>
#include<conio.h>
void main()
{
      int a,b,c;
      clrscr();
      printf("\t\tSwapping 2 variable with the help of 3rd variable");
      printf("\nEnter 1st No. = ");
      scanf("%d",&a);
      printf("Enter 2nd No. = ");
      scanf("%d",&b);
      printf("\nHere 1st No. is %d and 2nd No. is %d",a,b);
      printf("\n\nPress any to Swap 1st to 2nd No.");
      getch();
      c=a;
      a=b;
      b=c;
      printf("\n\nSwapped & 1st No. is %d and 2nd No. is %d",a,b);
      printf("\n\n\tPress any key to QUIT");
      getch();
}
```

Ques. **Make a program to swap 2 integers without the help of third variable.**

```
//This Program is made by Pankaj Singh Gill to swap 2 nos. without using 3rd
variable//
#include<stdio.h>
#include<conio.h>
void main()
{
      int a,b;
      clrscr();
      printf("\t\t\tSwapping Integers without using 3rd integers");
      printf("\nEnter 1st No. = ");
      scanf("%d",&a);
      printf("Enter 2nd No. = ");
      scanf("%d",&b);
      printf("\nHere 1st No. is %d and 2nd No. is %d",a,b);
      printf("\n\nPress any to Swap 1st to 2nd No.");
      getch();
      a=a+b;
      b=a-b;
      a=a-b;
      printf("\n\nSwapped & 1st No. is %d and 2nd No. is %d",a,b);
      printf("\n\n\tPress any key to QUIT");
      getch();
}
```

Ques. **Make a program to find out whether the no. is even or odd by using conditional operators.**

```
/*This Program is for determination of even/odd number *
**This Program is made by Pankaj Singh Gill ***********/
#include"stdio.h"
#include"conio.h"
void main()
{
      int num;
      clrscr();
      printf("Enter No. = ");
      scanf("%d",&num);
      num%2==0?printf("%d is Even",num):printf("%d is Odd",num);
      getch();
}
```

7. CONSOLE INPUT/OUTPUT

Console I/O in general means communications with the computer's keyboard and display. However, in most modern operating systems the keyboard and display are simply the default input and output devices, and user can easily redirect input from, say, a file or other program and redirect output to, say, a serial I/O port:

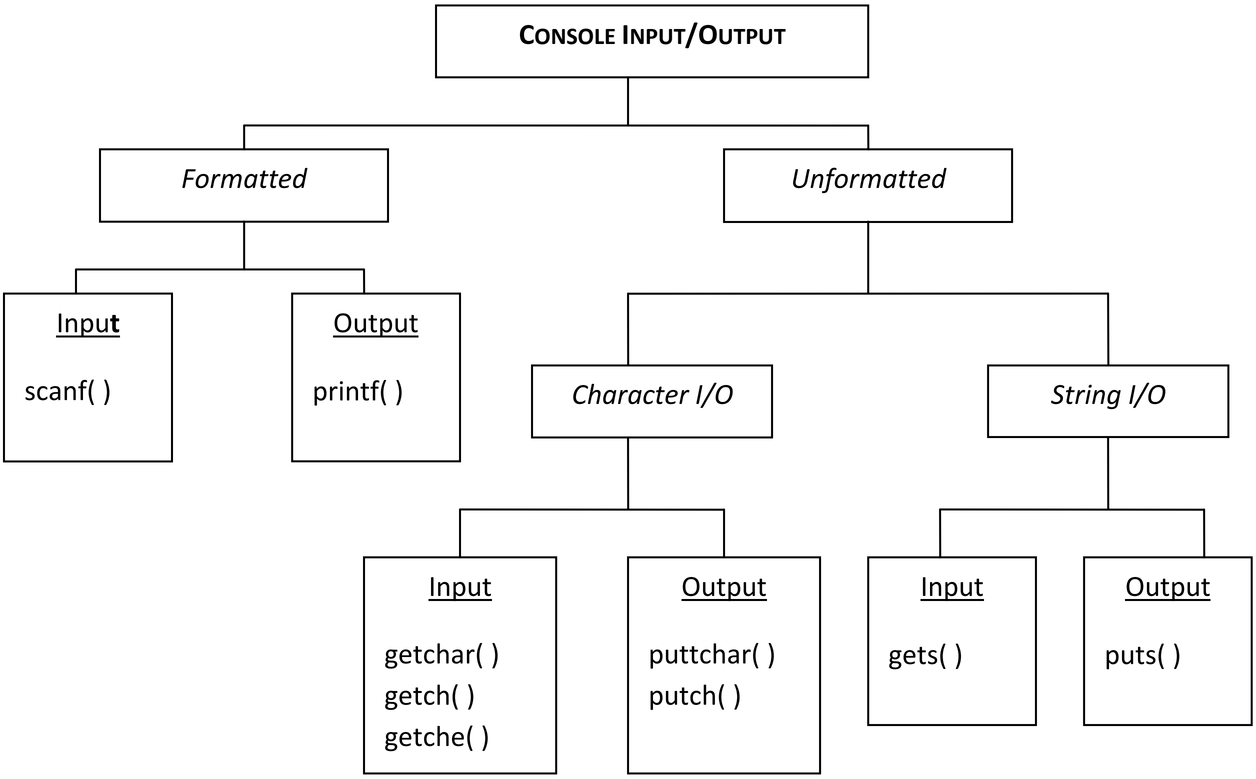
```
type infile > myprog > com
```

The program itself, "myprog", doesn't know the difference. The program uses console I/O to simply read its "standard input (stdin)" -- which might be the keyboard, a file dump, or the output of some other program -- and print to its "standard output (stdout)" -- which might be the display or printer or another program or a file. The program itself neither knows nor cares.

Console I/O requires the declaration:

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

Now, we will study about the types of Console Input/Output in C language.



Useful functions include:

Name	Action
printf()	Print a formatted string to stdout.
scanf()	Read formatted data from stdin.
putchar()	Print a single character to stdout.
getchar()	Read a single character from stdin.
puts()	Print a string to stdout.
gets()	Read a line from stdin.
getch()	Get a character from the keyboard(no need to press Enter).
getche()	Get a character from the keyboard and echo it.

The available format codes are:

Symbol	Name
%d	decimal integer
%ld	long decimal integer
%c	character
%s	string
%e	floating-point number in exponential notation
%f	floating-point number in decimal notation
%g	use %e and %f, whichever is shorter
%u	unsigned decimal integer
%o	unsigned octal integer
%x	unsigned hex integer

Using the wrong format code for a particular data type can lead to bizarre output. Further control can be obtained with modifier codes; for example, a numeric prefix can be included to specify the minimum field width:

```
%10d
```

This specifies a minimum field width of ten characters. If the field width is too small, a wider field will be used. Adding a minus sign:

```
%-10d
```

-- causes the text to be left-justified. A numeric precision can also be specified:

```
%6.3f
```

This specifies three digits of precision in a field six characters wide. A string precision can be specified as well, to indicate the maximum number of characters to be printed. For example:

```
/* prtint.c */

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

```
{
     printf( "<%d>\n", 336 );
     printf( "<%2d>\n", 336 );
     printf( "<%10d>\n", 336 );
     printf( "<%-10d>\n", 336 );
}
```

This prints:

```
<336>
<336>
<       336>
<336       >
```

Similarly:

```
/* prfloat.c */
#include <stdio.h>
void main()
{
     printf( "<%f>\n", 1234.56 );
     printf( "<%e>\n", 1234.56 );
     printf( "<%4.2f>\n", 1234.56 );
     printf( "<%3.1f>\n", 1234.56 );
     printf( "<%10.3f>\n", 1234.56 );
     printf( "<%10.3e>\n", 1234.56 );
}
```

That prints:

```
<1234.560000>
<1.234560e+03>
<1234.56>
<1234.6>
<  1234.560>
< 1.234e+03>
```

And finally we will make a program to show the demonstration of Console I/O.

```
//Program for demonstration of Console I/O Functions//
#include<stdio.h>
#include<conio.h>
void main()
{
      char a,b,c;
      char d[10];
      clrscr();
      printf("\n\nEnter String = ");
      gets(d);
      printf("\nd = ");
      puts(d);
      printf("Enter 1st Character = ");
```

```
        a=getchar();
        printf("a = ");
        putchar(a);
        getch();
        printf("\n\nEnter 2nd Character = ");
        b=getch();
        printf("\nb = ");
        putch(b);
        getch();
        printf("\n\nEnter 3rd Character = ");
        c=getche();
        printf("\nc = ");
        putch(c);
        getch();
        printf(“Press any key to QUIT”);
        getch();
}
```

Note : We cannot use the gets(),puts() function with all the other Input/Output function in a same program, If we want to use these functions then we should have to place gets() & puts() above all the other functions.

Ques. Make a program to reverse the 5 digit Number.

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

        int number, rev_num, next_digit,last_digit;
        printf ("Enter the number that is to be reversed: ");
        scanf("%d", &number);
        last_digit = number - ((number / 10) * 10); //units place//
        rev_num = last_digit;
        next_digit = (number / 10) - ((number / 100) * 10); //tenth's place//
        rev_num = (rev_num * 10) + next_digit;
        next_digit = (number / 100) - ((number / 1000) * 10); //hundred's place//
        rev_num = (rev_num * 10) + next_digit;
        next_digit = (number / 1000) - ((number / 10000) * 10); //thousand's place//
        rev_num = (rev_num * 10) + next_digit;
        next_digit = (number / 10000) - ((number / 100000) * 10); //ten thousand's
        place//
        rev_num = (rev_num * 10) + next_digit;
        printf ("The Reversed Number is: %d",rev_num);
        getch();
        printf(“Press any key to QUIT”);
        getch();

        }
```

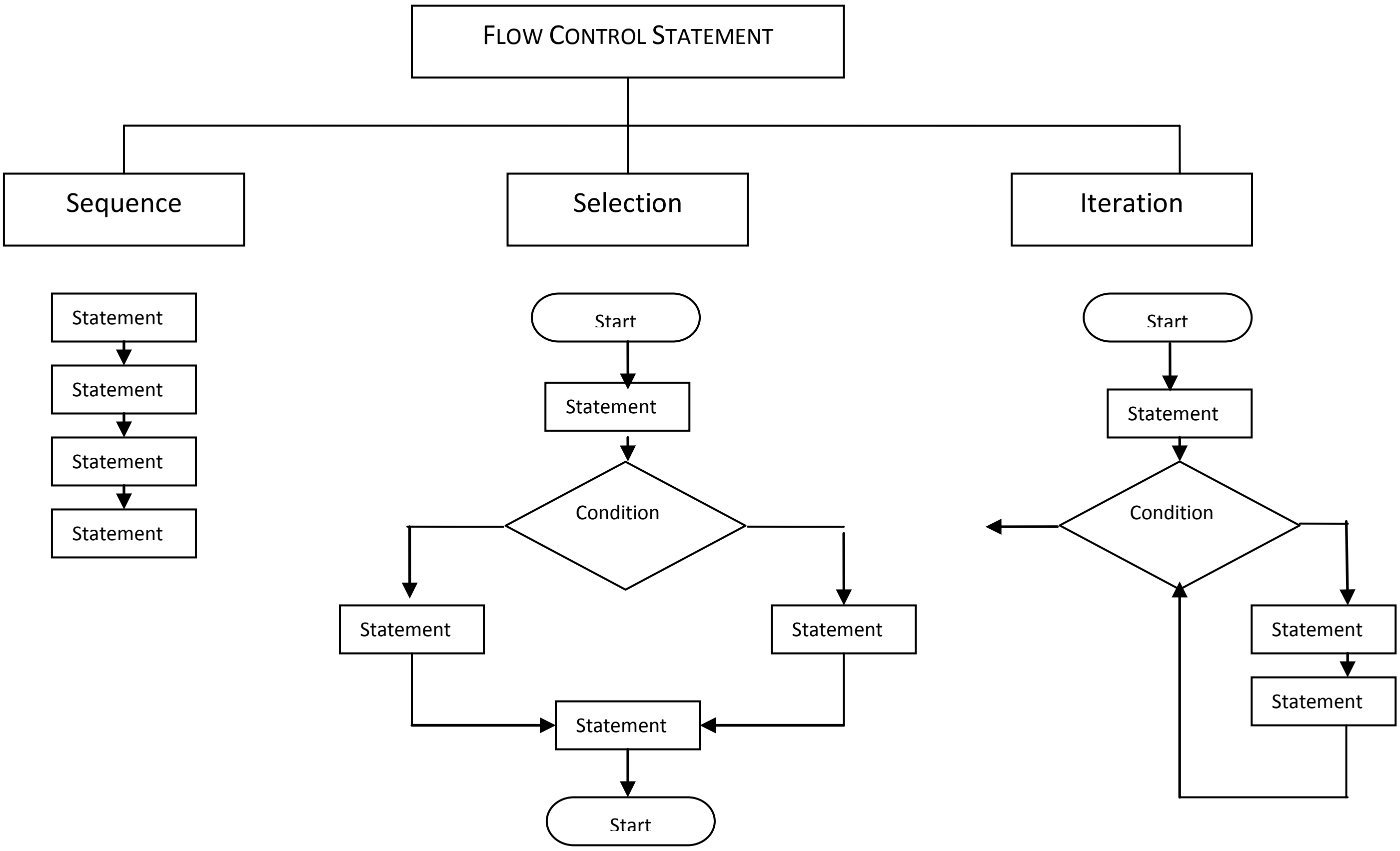
Ques. **Make a program to find out the sum of five digits number.**

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

```
void main()
{
       int n,n1,n2,n3,n4,n5,i,y;
       clrscr();
       printf("Enter Number to Find Sum of Digits.\n");
       scanf("%d", &n);
       n1=n/10000;
       n2=(n-n1*10000)/1000;
       n3=(n-n1*10000-n2*1000)/100;
       n4=(n-n1*10000-n2*1000-n3*100)/10;
       n5=(n-n1*10000-n2*1000-n3*100-n4*10);
       printf("\n\n\nSum of Digits of This Numbers are %d\n\n\n",n1+n2+n3+n4+n5);
       getche();
}
```

8. FLOW CONTROL STATEMENTS

Calculations and expressions are only a small part of computer programming. In addition to these, program control flow statements are needed. They specify the order in which statements are to be executed. The Flow Control Statements are explained in those types.



Now, we are going to study about *Selection Statements* or *Branching Statements*.

1. ***The if Statement***: – The if statement allows us to put some decision –making into our programs. The general format of the if statement is:

```
If (condition)
statement;
```

If the condition is true(nonzero), the statement will be executed. If the condition is false(0), the statement will not be executed. For example,

```
if (marks >= 40)
{
      printf("Congrats!!! You passed the EXAM.");
}
```

2. **The *else* Statement: –**It is the alternate form of the if statement. General format of the statement is:

```
if (condition)
{
      statement;
}
else
{
      statement;
}
```

If the condition is true(1), the first statement is executed. If it is false(0), the second statement is executed. For example,

```
if (marks >= 40)
{
      printf("Congrats!!! You passed the EXAM.");
}
else
{
      printf("You are Failed. Better luck Next Time.");
}
```

Ques. WAP to find S.I. when rate is 8% if time is less then or equal to 3 & rate is 12% if time is greater than 3

```
#include<stdio.h>
#include<conio.h>
void main()
{
      floatp,r,t,si;                //variable declaration
      clrscr();
      printf("Enter Principal Amount\t: ");
      scanf("%f",&p);
      printf("Enter Time Period\t   : ");
      scanf("%f",&t);
      if(t<=3)                            //Case Rate
      {
            r=8;
            printf("\n\nRate will be %.2f percent",r);
      }
      Else                                //Default Case
      {
            r=12;
            printf("\n\nRate will be %.2f percent",r);
      }
      si=(p*r*t)/100;
      printf("\nSimple Interest to be charged is %.2f",si);
      printf("\n\n\t\t\tPress any key to QUIT");
      getch();
}
```

Ques. WAP to calculate the total electricity bill. If the person is male than he will get a discount of 3% on the bill when the bill is more than Rs. 4000, and if the person is female than she will get 3% discount if the bill amount is less than Rs. 4000 and 5% if the bill amount is more than Rs. 4000.

```
#include <stdio.h>
#include <conio.h>
void main()
{
      char name[30],gender, phone[11], city[50];
      int unit, rent=1000, bill;
      clrscr();
      printf("\t\t\t\tELECTRICITY BILL");
      printf("\n\nName : ");
      scanf("%s",&name);
      printf("\nGender(M/F) : ");
      checkgen:
      scanf("%c",&gender);
      if((gender=='f')||(gender=='F')||(gender=='m')||(gender=='M'))
      {
            goto proceed;
      }
      else
      {
            goto checkgen;
      }
      proceed:
      printf("\nCity : ");
      scanf("%s",&city);
      printf("\nPhone no. : ");
      scanf("%s",&phone);
      clrscr();
      printf("\t\t\t\tELECTRICITY BILL");
      printf("\n\nName : %s\t Gender : %c\t Phone No. : %s\nCity :
%s",name,gender,phone,city);
      printf("\n\nUnits Consumed : ");
      scanf("%d",&unit);
      start:
      if(unit>0)
      {
            if (unit<=100)
            {
                  bill=rent+(unit*4);
            }
            else if((unit>100)&&(unit<=150))
            {
                  bill=rent+(unit*5);
            }
            else if((unit>150)&&(unit<=200))
            {
                  bill=rent+(unit*6);
            }
            else if((unit>200)&&(unit<=300))
            {
                  bill=rent+(unit*7);
            }
            else if(unit>300)
            {
                  bill=rent+(unit*8);
            }
      }
      else
      {
            goto start;
      }
      clrscr();
      printf("\t\t\t\tELECTRICITY BILL");
      printf("\n\nName : %s\t Gender : %c\t Phone No. : %s\nCity : %s\nUnits :
%d\tRent : 1000",name,gender,phone,city,unit);
      if((gender=='f')||(gender=='F'))
      {
            if(bill<=4000)
            {
                  printf("\n\n\n\t\t\t\tDiscount = 3%");
                  printf("\n\t\t\t\tBill : %d \n\t\t\t\tDiscounted Bill :
%d",bill,(bill-((bill*3)/100)));
            }
            else
            {
                  printf("\n\n\n\t\t\t\tDiscount = 5%");
                  printf("\n\t\t\t\tBill : %d \n\t\t\t\tDiscounted Bill :
%d",bill,(bill-((bill*5)/100)));
            }
      }
      else
      {
            if(bill<=4000)
            {
                  printf("\n\n\n\t\t\t\tDiscount = Null \n\t\t\t\tTotal Bill :
%d",bill);
            }
            else
            {
                  printf("\n\n\n\t\t\t\tDiscount = 3%");
                  printf("\n\t\t\t\tBill : %d \n\t\t\t\tDiscounted Bill :
%d",bill,(bill-((bill*3)/100)));
            }
      }
      getch();
```

```
}
```

***The Nested if* Statement: –** A nested if is an if that is the target of another if or else. Nested ifs are very common in programming. In a nested if, an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else. General format of the statement is:

```
if (condition)
{
     if (condition 1)
     {
          statement;
     }
     else
     {
          statement;
     }
}
else
{
     statement;
}
```

We will make the program based on this type of statement later.

3. ***The else-if* Statement**: – The else-if Statement is also called if-else-if ladder or if-else-if staircase because of its appearance. The general format for the if-else if statement is:

```
if(condition 1)
   simple or compound statement
else if(condition 2)
   simple or compound statement
else if( condition 3)
   simple or compound statement
   .....
else if( condition n )
   simple or compound statement
```

The conditions are evaluated from the top downward. As soon as true condition is found, the statement associated with it is executed and the rest of the ladder is bypassed. If none of the conditions are true, the final else is executed. That is, if all other conditional tests fail, the last else statement is performed. If the final else is not present, no action takes place if all other conditions are false. Let us make an example program of if-else-if statement.

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

{
  int i = 2;
  if(i == 0){
     printf(" i == 0. \n");
  }else if(i == 1){
     printf(" i == 1. \n");
  }else if(i == 2){
     printf(" i == 2. \n");
  }
}
```

4. ***The Switch* Statement**: – The switch statement is similar to a chain of if/else statements. The general format of a switch statement is:

```
switch(expression)
{
     case value1:
          statement;
     case value2:
          statement;
     case value3:
          statement;
}
```

The switch statement evaluates the value of an expression and branches to one of the case labels. Duplicate labels are not allowed, so only one case will be selected. The expression must evaluate an integer, character, or enumeration.

The case labels can be in any order and must be constants. The default label can be put anywhere in the switch. No two case labels can have the same value.

When C sees a switch statement, it evaluates the expression and then looks for a matching case label. If none is found, the default is used. If no default is found, the statement does nothing.

Break

Exit form loop or switch. We use break statement in switch to stop execution of the statements. If break statement is not used in the program where switch is used than C will execute every statement that is written after the true statement. We use break after the statement to stop the program producing extra statements. General format is:

```
break;
```

Later we will make a program based on switch and break statements. Now we are taking a brief pause.

A Brief Pause

The control of flow statements that we've just seen is quite adequate to write programs of any degree of complexity. They lie at the core of C and even a quick reading of everyday C programs will illustrate their importance, both in the provision of essential functionality and in the structure that they emphasize. The remaining statements are used to give programmers finer control or to make it easier to deal with exceptional conditions. Only the switch statement is enough of a heavyweight to need no justification for its use; yes, it can be replaced with lots of ifs, but it adds a lot of readability. The others, break, continue and goto, should be treated like the spices in a delicate sauce. Used carefully they can turn something commonplace into a treat, but a heavy hand will drown the flavor of everything else.

Ques. **Make a program that uses *else-if* statement to validate the users input to be in the range 1-10.**

```
#include <stdio.h>
void main()
{
        int number;
        int valid = 0;
        while( valid == 0 ) {
        printf("Enter a number between 1 and 10 = ");
        scanf("%d", &number);
        if( number < 1 )
        {
                printf("Number is below 1. Please re-enter\n");
                valid = 0;
        }
        else if( number > 10 )
        {
                printf("Number is above 10. Please re-enter\n");
                valid = 0;
        }
        else
        {
                valid = 1;
        }
        printf("The number is %d\n", number );
}
```

Ques. Make a program using *Nested-if* statement to find the greater number among 3 numbers without the help of logical operators.

```
/*Program to find Greater No. using Nested if **
**Program is made by Pankaj Singh Gill ********/
#include<stdio.h>
#include<conio.h>
void main()
{
      int a,b,c;
      clrscr();
      printf("Enter three number\n\n");
      scanf("%d%d%d",&a,&b,&c);
      if(a>b)
      {
            if(a>c)
            {
                  printf("\n a is Greater");
            }
            else
            {
                  printf("\n c is Greater");
            }
      }
      else
      {
            if(b>c)
            {
                  printf("\n b is Greater");
            }
            else
            {
                  printf("\n c is Greater");
            }
      }
      getch();
      printf("Press any key to QUIT");
      gettch();
}
```

Ques. Make a program using *Nested-if* statement to find the greater number among 3 numbers and with the help of logical operators.

```
/*Program to find Greater No. using Nested if **
**Program is made by Pankaj Singh Gill ********/
#include<stdio.h>
include<conio.h>
void main()
{
      int a,b,c;
      clrscr();
      printf("Enter Three Numbers \n");
      scanf("%d%d%d", &a, &b, &c);
      if((a>b)&&(a>c))
      {
            printf("Greater Number is %d", a);
      }
      else if((b>a)&&(b>c))
      {
            printf("Greater Number is %d", b);
      }
      else
      {
            printf("Greater Number is %d", c);
      }
      getch();
      printf(“Press any key to QUIT”);
      getch();
}
```

Ques. **Make a program to find out that the no. entered is divisible by another no. or not, using *if else* statement .**

```
#include<stdio.h>
#include<conio.h>
void main()
{
      int a,b,c;
      clrscr();
      printf("Enter first no. = ");
      scanf("%d", &a);
      printf("\nEnter second no. = ");
      scanf("%d", &b);
      c=a%b;
      if(c==0)
      {
            printf("\nNumber is divisible by 2nd Number.");
      }
      else
      {
            printf("\nNumber is not divisible by 2nd Number.");
      }
      getch();
}
```

Ques. **Make a program to check for upper and lowercase letters.**

```
#include <stdio.h>
void main()
{
      char cResponse = '\0';
      printf("Enter the letter A: ");
      scanf("%c", &cResponse);
      if ( cResponse== 'A' || cResponse == 'a' )
      {
            printf("\nCorrect response\n");
      }
      else
     {
            printf("\nIncorrect response\n");
      }
      getch();
}
```

Ques. **Make a program to conclude the result of a student by taking marks of 5 different subjects.**

```
/* Programmer Name - Pankaj Gill
** Date - 26/02/2012
** Aim - To find out the result of a student */
#include<stdio.h>
#include<conio.h>
void main()
{
      float a,b,c,d,e;
      clrscr();
      start:
      printf("\t\t\t\t Exam Result");
      printf("\n\nNote: Maximum Marks of one subject = 100");
      printf("\n\n\n\n\nEnter Your Marks obtained in English        = ");
      scanf("%f",&a);
      printf("Enter Your Marks obtained in Hindi          = ");   scanf("%f",&b);
      printf("Enter Your Marks obtained in Math           = ");
      scanf("%f",&c);
      printf("Enter Your Marks obtained in Science        = ");
      scanf("%f",&d);
      printf("Enter Your Marks obtained in Social Studies = ");
      scanf("%f",&e);
      if((a>100)||(b>100)||(c>100)||(d>100)||(e>100))
      {
            printf("\n\n\"Invalid Input\" Please Enter Marks under 100");
            getch();
            clrscr();
            goto start;
      }
      else if((a>=33)&&(b>=33)&&(c>=33)&&(d>=33)&&(e>=33))
      {
            printf("\n\n\t\tCongrats!!! You got passed in the Examination...");
            printf("\n\nTotal Marks = %d /500",(a+b+c+d+e));
            printf("\nPercentage  = %d",((a+b+c+d+e)/500)*100);
            getch();
      }


else
if(((a>=33)&&(b>=33)&&(c>=33)&&(d>=33)&&(e<33))||((a>=33)&&(b>=33)&&(c>=33)&&(d<33
)&&(e<=33))||((a>=33)&&(b>=33)&&(c<33)&&(d>=33)&&(e<=33))||((a>=33)&&(b<33)&&(c>=3
3)&&(d>=33)&&(e<=33))||((a<33)&&(b>=33)&&(c>=33)&&(d>=33)&&(e<=33)))
      {
            printf("\n\n\t\tYou got Compartment!!!");
      }
      else
```

```
	{
		printf("\n\n\t\tYou are failed this time, Better Luck Next Time...");
	}
	getch();
	printf("\n\nPress any key to QUIT");
	getch();
	printf("\a");
}
```

9. LOOPING STATEMENTS

It is defined as repetition. Loop is used when certain set of statement are to be executed again and again in a program. The statements to be repeted are written in pair of { }.

Looping statement allow the program to repeat a section of code any number of times or until some condition occurs. For example, loops are used to count the number of words in a document or to count the number of accounts that have past-due balances. There are three types of loops, *for* loop, *while* loop and *do while* loop.

1. The *for* Statement: The for statement allows the programmer to execute a block of code for a specified number of times. The general format of the *for* statement is:

```
for(initial statement; condition; iteration-statement)

{

      body-statement;

}
```

Simple example program of *for* loop:-

```
int i;
for(i=1;i<=5;i=i++)
{
      print(“Hello...”);
}
```

The Infinite loop – Although you can use any loop statement to create an infinite loop, *for* is traditionally used for this purpose. Since none of the three expressions that form the *for* loop are required, you can make an endless loop by leaving the expression empty:

```
for( ; ; )
{
      printf(“This loop will run forever.\n”);
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the for(; ;) construct to signify an infinite loop.

Nested loop – If a *for* loop is defined in another *for* loop, it is called nested for loop. The outer loop is executed first of all. The general format of nested for loop is:

```
for(initial statement; condition; iteration-statement)
{
     for(initial statement; condition; iteration-statement)
     {
          body-statement;
     }
     body statement;
}
```

2. The *while* statement: The *while* statement is used when the program needs to perform repetitive tasks. It is also called *pre-test* loop, it is choice of uses to select *for* 'or' *while* loop for a particular program. When initialization is done outside the loop, we will use *while* loop. The general form of a *while* statement is:

```
initialisation
while(condition)
{
     statement;


     increment/decrement;
}
```

*The infinite **while** loop –* The general format of infinite while statement is:

```
while(1)
{


}
```

3. The *do while* statement: It is similar to *while* loop but in *do while* loop, condition is checked after execution of loop. It is used when number of times loop will be executed is known in advance. Whether condition is true/false, loop will be executed atleast once without checking the condition. For this, two keywords *do&while* are used. The general format of *do while* loop is:

```
initialization
```

```
do
{
	statement;
	increment/decrement
}	while(condition)
```

Jump Statements

C has four statements that perform an unconditional branch: *return, goto, break*, and *continue*. Of these, we may use *return* and *goto* anywhere in our program. We may use the *break* and *continue* statements in conjunction with nay of the loop statements.

1. The ***return*** Statement: – The *return* statement is used to return a function. It is categorized as a jump statement because it causes execution to return (jump back) to the point at which the call to the function was made. A return may or may not have a value associated with it.

If *return* has a value associated with it, that value becomes the return value of the function. In C, a non-void function does not technically have to return a value. If no return value is specified, a garbage value is returned. The general format of the *return* statement is:

```
return expression;
```

The expression is present only if the function is declared as returning a value. In this case, the value of expression will become the return value of the function.

We can use as many *return* statements as we like within a function. However, the function will stop executing as soon as it encounters the first return. The } that ends a function also causes the function to return. It is the same as a return without any specified value. If this occurs within a non-void function, then the return value of the function is undefined.

2. The ***goto*** Statement: – Since C has a rich set of a control structures and allows additional control using *break* and *continue*, there is little need for *goto*. Most programmer's chief concern about the *goto* is its tendency to render programs unreadable. Nevertheless, although the *goto* statement fell out of favor some years ago, it occasionally has its uses.

There are no programming situations that require *goto.* Rather, it is a convenience, which, if used wisely, can be benefit in a narrow set of programming situations, such as jumping out of a set of deeply nested loops.

The *goto* statement requires a label for operation. A label is a valid identifier followed by a colon. Furthermore, the label must be in the same function as the goto that uses it–you cannot jump between functions. The general format of *goto* statement is:

```
goto label;

label:
```

where label is any valid label either before or after goto.

*Note:*In C, the names of variables, functions, labels, and various other user-defined objects are called identifiers.

3. The ***break*** Statement: – Loops can be exited at any point through the use of a *break* statement. Break statement is a jump statement in C and is generally used for breaking from a loop or breaking from a case as discussed in switch statement.

Break statement when encountered within a loop immediately terminates the loop by passing condition check and execution transfer to the first statment after the loop. In case of switch it terminates the flow of control from one case to another.

Also one important thing to remember about break is that it terminates only the innermost switch or loop construct in case of nested switch and loop variations.

The C source code below shows an simple application of break statement-

```
#include <stdio.h>
void main ()
{
        for (int i = 0; i<100; i++)
        {
                printf ("\n%d", i);
                if (i == 10); //This code prints value only upto 10.
                break;
        }
}
```

4. The ***continue*** Statement: – Continue is a jump statement provided by C. It is analogues to break statement. Unlike break which breaks the loop, continue statement forces the next execution of loop bypassing any code in between.

For *for* statements it causes the conditional check and increment of loop variable, for while and do-while it passes the control to the condition check jumping all the statements in between. Continue plays an important role in efficiently applying certain algorithms.

Below is a C source code showing the application of continue statement –

```
#include<stdio.h>
void main()
{
        for(int i = 0; i<100; i++)
        {
                if(i == 10); // This code prints value only upto 9 even though loop
executes 100 times.
                continue ;
                printf ("\n%d", i);
        }
}
```

Ques. WAP to check whether the number entered is a prime number or not.

```
#include"stdio.h"
#include"conio.h"
void main()
{
        int num,i,j;
        clrscr();
        printf("\nEnter number:\n");
        scanf("%d",&num);
        i=2;
        while(i<num)
        {
                j=num%i;
                if(j==0)
                {
                        printf("\nThe number is not prime");
                        break;
                }
                i++;
        }
        if(i==num)
        {
                printf("\nThe number is prime");
        }
        getch();
}
```

Ques. WAP to print prime number between 100 to 200.

```
#include"stdio.h"
#include"conio.h"
void main()
{
        int i,j,num,count=0,n,n1;
        clrscr();
        printf("\nEnter the start number:");
        scanf("%d",&n);
        printf("\nEnter the end number:");
        scanf("%d",&n1);
        i=n;
        while(i<n1)
        {
                for(j=2;j<i;j++)
                {
                        num=i%j;
                        if(num==0)
                        {
                                break;
                        }
                }
                if(j==i)
                {       printf("\n%d",i);
```

```
                                                count++;
                                }
                                i++;
                }
                printf("\nThe total prime number is: %d",count);
                getch();
}
```

Ques. Write a program to find the sum of *n* integers.

```
/*This program is to find the sum of n integers**
**Program is made by Pankaj Gill**
**This project is started on 27/02/2012 22:42*/
#include<stdio.h>
#include<conio.h>
void main()
{
        int a,b,c,d=0;
        clrscr();
        printf("\t\t\t\tTitle : Sum of n integers\n\n");
        printf("Enter the number of integers = ");
        scanf("%d",&a);
        for(b=1;b<=a;b++)
        {
                printf("\nEnter %d Integer = ",b);
                scanf("%d",&c);
                d=c+d;
        }
        printf("\n\n\t\tSum of %d integers is %d",a,d);
        printf("\n\n\t\tAverage is %d",(d/a));
        getch();
        printf("\n\n\nPress any key to QUIT");
        getch();
}
```

Ques. Write a program to find the table of any integerentered by the user.

```
//Program is made for print the table of any number//
//Program is made by Pankaj Gill//
#include<stdio.h>
#include<conio.h>
void main()
{
        int n,i,j;
        clrscr();
        printf("Enter the number = ");
        scanf("%d",&n);


                for(j=1;j<=10;j++)
                printf("\n%d*%d=%d ",n,j,n*j);

        getch();
        printf("Press any key to QUIT");
}
```

Ques. Make a program to reverse the table of any integerentered by the user.

```
//Program is made for reversing an integer's table.//
//Program is made on 28/02/2012 at 12:45//
#include<stdio.h>
```

```
#include<conio.h>
void main()
{
        int a,b;
        clrscr();
        printf("Enter an Integer to reverse its table = ");
        scanf("%d",&a);
        for(b=10;b>=1;b--)
        {
               printf("\n%d * %d = %d",a,b,a*b);
        }
        getch();
        printf("\n\nPress any key to QUIT");
}
```

Ques. Make a program to draw a right angle triangle by the help of * sign.

```
/*Aim - To draw a Triangle with the help of * sign ***
**Program is compiled by Pankaj Singh Gill ***********
**Date & Time of Development - 28/02/2012 21:06 *****/
#include<stdio.h>
#include<conio.h>
void main()
{
        int a,i,j;
        clrscr();
        printf("\t\t\t\tRight Angled Triangle");
        printf("\n\nEnter the maximum dot size = ");
        scanf("%d",&a);
        printf("\n\n");
        for(i=1;i<=a;i++)
        {
            printf("\n");
            for(j=1;j<=i;j++)
            {
               printf(" * ");
            }
        }
        getch();
}
```

Ques. Make a program to draw a Triangle with the help of integers.

```
/*Aim - To draw a Triangle with the help of integers *
**Program is compiled by Pankaj Singh Gill ***********
**Date & Time of Development - 28/02/2012 21:06 *****/
#include<stdio.h>
#include<conio.h>
void main()
{
        int a,i,j;
        clrscr();
        printf("\t\t\t\tRight Angled Triangle");
        printf("\n\nMaximum limit for an accurate triangle is 9\nEnter the maximum
            dot size = ");
        scanf("%d",&a);
        printf("\n\n");
        for(i=1;i<=a;i++)
        {
            printf("\n");
            for(j=1;j<=i;j++)
            {
               printf(" %d ",i);
            }
            printf("\n");
        }
        getch();
}
```

Ques. **Make a program to draw a triangle with the help of triangle.**

```
/*Aim - To draw a Triangle with the help of integers *
**Program is compiled by Pankaj Singh Gill ***********
**Date & Time of Development - 28/02/2012 21:06 *****/
#include<stdio.h>
#include<conio.h>
void main()
{
        int a,i,j;
        clrscr();
        printf("\t\t\t\tRight Angled Triangle");
        printf("\n\nMaximum limit for an accurate triangle is 9\nEnter the maximum
            dot size = ");
        scanf("%d",&a);
        printf("\n\n");
        for(i=1;i<=a;i++)
        {
            printf("\n");
            for(j=1;j<=i;j++)
            {
               printf(" %d ",i);
            }
            printf("\n");
        }
        getch();
}
```

Ques. **Make a program to draw triangle, square & rectangle by taking input from the user.**

```
/********************Three Shapes***************************
** Aim - To draw a selected shape with the help of '*'sign *
** Program is compiled by Pankaj Singh Gill ****************
** Date & Time of Development - 28/02/2012 22:48 ***********
***********************************************************/
#include<stdio.h>
#include<conio.h>
void main()
{
        int a,b,i,j;       //variable declaration a=length or max. size, b=bredth
            of rectangle, 'i' for vertical loop, 'j' for horizontal loop
        char c;            //variable declaration
        clrscr();
        printf("\t\t\t\tThree Shapes");
        printf("\n");
        start:
        printf("\n\nEnter the desired shape Right Angle Triangle(T), Square(S),
            Rectangle(R) = ");
        c=getch();
```

```
if((c=='T')||(c=='t'))        //Condition 1
{
    printf("\n\nEnter the maximum dot size of a Right Angle triangle = ");
    scanf("%d",&a);
    for(i=1;i<=a;i++)
    {
       printf("\n");
       for(j=1;j<=i;j++)
       {
              printf(" * ");
       }
       printf("\n");
    }
    getch();
    goto end;
}
else if((c=='S')||(c=='s'))   //Condition 2
{
    printf("\n\nEnter a side of a square = ");
    scanf("%d",&a);
    for(i=1;i<=a;i++)
    {
       printf("\n");
       for(j=1;j<=a;j++)
       {
              printf(" * ");
       }
       printf("\n");
    }
    getch();
    goto end;
}
else if((c=='R')||(c=='r'))   //Condition 3
{
    printf("\n\nEnter the length of Rectangle = ");
    scanf("%d",&a);
    printf("\nEnter the bredth of Rectangle =  ");
    scanf("%d",&b);
    for(i=1;i<=a;i++)
    {
       printf("\n");
       for(j=1;j<=b;j++)
       {
              printf(" * ");
       }
       printf("\n");
    }
    getch();
    goto end;
}
else                         //default
{
    goto start;
}
end:
printf("\nDo you want to CONTINUE Y/N");
c=getch();
if((c=='Y')||(c=='y'))
{
    clrscr();
```

```
            goto start;
        }
        else if((c=='N')||(c=='n'))
        {
            exit();
        }
        else
        {
            printf("\a\nINVALID INPUT \nPlease Enter a valid Input");
            goto end;
        }
}
```

Ques. Make a program to draw the mirror image of a right angle triangle.

```
/* This is a simple mirror-image of a right angle triangle */

#include<stdio.h>
#include<conio.h>
void main()
{
        int i, j,n=4,s;
        for(i=1;i<=10;i++)
        {
            printf("\n");
            for(j=1;j<=i;j++)
            {
               printf(" *");
            }
            for (s=n;s>=1;s--)
            {  // Spacing factor
               printf("  ");
            }
            for (j=1;j<=i;j++)
            {
               printf(" *");
            }
        printf("\n");
        --n;   // Controls the spacing factor
        }
        getch();
}
```

Ques. **Make a program to make a pattern given below with your name between the pattern.**

```
* * * * * * * * *
* * * *   * * * *
* * *       * * *
* *           * *
*               *
* *           * *
* * *       * * *
* * * *   * * * *
* * * * * * * * *
```

```
/*****************************************************
** Aim - To Print Pattern & Name between the pattern **
** Compiled by - Pankaj Singh Gill                    **
** Date & Time of Developement - 01/03/2012 21:50    **
*****************************************************/
#include<stdio.h>
#include<conio.h>
void main()
{
        int i,j,k;              //variable declaration
        clrscr();
        for(i=19;i>=1;i--)       //loop 1 - upper triangles
        {
                printf("\n");
                for(j=1;j<=i;j++)
                {
                        printf("* ");

                }
                for(k=18;k>=i;k--)
                {
                        printf("    ");
                }
                for(j=1;j<=i;j++)
                {
                        printf("* ");
                }
        }
        printf("\n*                           P A N K A J   G I L L
*");
        for(i=2;i<=19;i++)       //loop 2 - lower triangles
        {
                printf("\n");
                for(j=1;j<=i;j++)
                {
                        printf("* ");

                }
                for(k=18;k>=i;k--)
                {
                        printf("    ");
                }
                for(j=1;j<=i;j++)
                {
                        printf("* ");
                }
        }
        getch();
}
```

Ques. Make a program to find any integer’s table using while loop.

```
/*******************************************************
** Aim - To print a table of user entered integer    **
** This program is made for demonstrating while loop **
** Date & Time of Compilation - 04/03/2012 21:17     **
*******************************************************/
#include<stdio.h>
#include<conio.h>
void main()
{
        int a,b;
        clrscr();
        printf("Enter an integer = ");
        scanf("%d",&a);
        b=1;
        while(b<=10)
        {
                printf("\n%d * %d = %d",a,b,a*b);
                b++;
        }
        getch();
}
```

Ques. Write a program to find the Armstrong number.

```
#include<stdio.h>
#include<conio.h>
void main()
{
        long int a,n,t,r,s;
        clrscr();
        printf("\t\t\tArmstrong numbers");
        printf("\n\nEnter the maximum range to find Armstrong Numbers = ");
        scanf("%ld",&a);
        for(n=1;n<=a;n++)
        {
                t=n;
                s=0;
                while(t>0)
                {
                        r=t%10;
                        s=s+(r*r*r);
                        t=t/10;
                }
                if(n==s)
                printf("\n\n%ld",n);
        }
        printf("\n\nPress any key to QUIT");
        getch();
}
```

Ques. WAP to enter a no. and its power and show the result

```
#include<stdio.h>
#include<conio.h>
void main()
{
      int i=1,e,x,p=1;        //variable declaration
      clrscr();
      printf("Enter a number : ");
      scanf("%d",&e);         //scanning variables
      printf("Enter its power : ");
      scanf("%d",&x);
      while(i<=x)              //loop for power
      {
            p=p*e;
            i++;
      }
      printf("\n%d raised to power %d is %d",e,x,p);
      printf("\n\n\t\t\tPress any key to QUIT");
      getch();
}
```

Ques. WAP to enter a decimal no. and convert into binary no.

```
#include<stdio.h>
#include<conio.h>
void main()
{
      inta,b;                              //variable declaration
      clrscr();
      printf("Enter the number : ");
      scanf("%d",&a);
      printf("\t");
      while(a>=1)                   //loop – binary
      {
            b=a%2;
            printf("%d\b\b",b);
            a=a/2;
      }
      printf("\n\n\t\t\tPress any key to QUIT");
      getch();
}
```

Ques. WAP to show the Fibonacci series upto 100

```
#include<stdio.h>
#include<conio.h>
void main()
{
      int a=0,b=1,c;                       //variable declaration
      clrscr();
      printf("The fibonacci series upto 100 : \n");
      printf("%d %d ",a,b);
```

```
	for(c=0;c<=89;c++)			//loop – Fibonacci
	{
		c=a+b;
		a=b;
		b=c;
		printf("%d ",c);
	}
	printf("\n\n\t\t\tPress any key to QUIT");
	getch();
}
```

Ques. **WAP to convert binary number into decimal number.**

```
#include"stdio.h"
#include"conio.h"
void main()
{
	int binary,decimal=0,pow=1,num;
	clrscr();
	printf("\nEnter Binary number:");
	scanf("%d",&binary);
	while(binary!=0)
	{
		num=binary%10;
		binary=binary/10;
		decimal+=num*pow;
		pow=pow*2;
	}
	printf("\nThe decimal number is:%d",decimal);
	getch();
}
```

Ques. **WAP to convert any number into binary number.**

```
#include"stdio.h"
#include"conio.h"
void main()
{
	int num,rem;
	char ch;
	clrscr();
	printf("Press d for decimal to binary");
	printf("\nPress O for octal to binary");
	printf("\nPress H for hexadecimal to binary");
	printf("\nEnter your choice:");
	ch=getch();
	switch(ch)
	{
		case 'D':
		case 'd':
			printf("\nEnter number:");
			scanf("%d",&num);
			printf("Binary number is:\n\t\t\t");
```

```
                        while(num!=0)
                        {
                                rem=num%2;
                                printf("%d\b\b",rem);
                                num=num/2;
                        }
                        break;
                case 'O':
                case 'o':
                        printf("\nEnter number:");
                        scanf("%d",&num);
                        printf("Octal to binary is:\t\t\t");
                        while(num!=0)
                        {
                                rem=num%10;
                                num=num/10;
                                switch(rem)
                                {
                                        case 0:
                                                printf("000\b\b\b\b\b\b");
                                                break;
                                        case 1:
                                                printf("001\b\b\b\b\b\b");
                                                break;
                                        case 2:
                                                printf("010\b\b\b\b\b\b");
                                                break;
                                        case 3:
                                                printf("011\b\b\b\b\b\b");
                                                break;
                                        case 4:
                                                printf("100\b\b\b\b\b\b");
                                                break;
                                        case 5:
                                                printf("101\b\b\b\b\b\b");
                                                break;
                                        case 6:
                                                printf("110\b\b\b\b\b\b");
                                                break;
                                        case 7:
                                                printf("111\b\b\b\b\b\b");
                                                break;
                                        default:
                                                printf("It is not octal number");
                                                break;
                                }
                        }
        }
        getch();
}
```

Ques. **WAP to convert Decimal number into any number.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
      int a,b,num;
      clrscr();
      printf("Enter the number : ");
      scanf("%d",&a);
      num=a;
      printf("\nThe binary conversion of %d is = \t\t",a);
      while(a>=1)
      {
            b=a%2;
            printf("%d\b\b",b);
            a=a/2;
      }
      a=num;
      printf("\nThe Octal conversion of %d is = \t\t",a);
      while(a>=1)
      {
            b=a%8;
            printf("%d\b\b",b);
            a=a/8;
      }
      a=num;
      printf("\nThe Hexadecimal conversion of %d is = \t\t",a);
      while(a>=1)
      {
            b=a%16;
            switch(b)
            {
                  case 10:
                        printf("A\b\b");
                        break;
                  case 11:
                        printf("B\b\b");
                        break;
                  case 12:
                        printf("C\b\b");
                        break;
                  case 13:
                        printf("D\b\b");
                        break;
                  case 14:
                        printf("E\b\b");
                        break;
                  case 15:
                        printf("F\b\b");
                        break;
                  default:
                        printf("%d\b\b",b);
            }
            a=a/16;
      }
      printf("\n\n\t\t\tPress any key to QUIT");
      getch();
}
```

10. ARRAY

Array is defined as a collection of related items. All the items are similar data types. All items or elements are assigned a common name.

In constructing our building, we have identified each brick (variable) by name. That process is fine for a small number of bricks, but what happens when we want to construct something larger? We would like to point to a stack of Plates. That's plate 1, plate 2, plate 3….

Arrays allow us to do something similar with variables. An array is a set of homogenous elements of consecutive memory locations used to store data. Each item in the array is called an element. The number of elements in an array is called the dimension of the array.

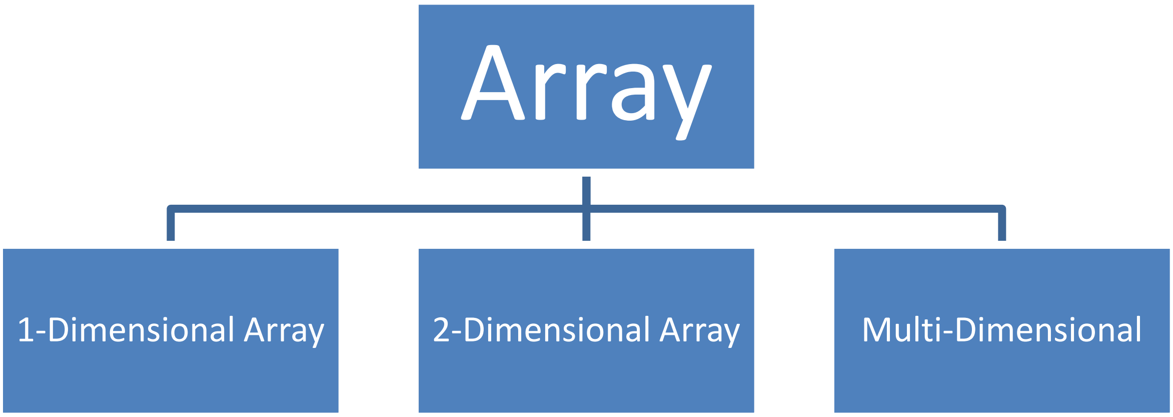
Arrays are allocated in continuous manner.

Lower bound : the smallest element of an array's index.

Upper bound : the highest element of an array's index.

Range : the number of element in the array.

Range : upper bound – lower bound+1



Declaration of Arrays

Standard Array declaration is as follows: -

```
Data_typename[array length];
```

We can declare the array of any type as show below

```
double height[10];
```

```
float width[20];

int min[9];
```

Note: In C, arrays start at position 0. Therefore, when you writechar p[10];

You are declaring a character array that has ten elements. p[0] through p[9].

Initialization of Arrays

So far we have used arrays that did not have any values in them to begin with. We managed to store values in them during program execution. Let us now see how to initialize an array while declaring it. Following are a few examples that demonstrate this.

```
intnum[6]={2,4,5,12,3,45};

int n[]={1,2,3,4,5,6,7,8,9};

float press[]={12.3,45.2,123.2,34.4};
```

Note:-

- Till the array elements are not given any specific value, they are supposed to contain garbage values.
- If the array is initialized where it is declared, mentioning the dimension of the array is optional as in the 2nd example above.

Array Operation

- Insertion : - A process to adding a new element to the array.
- Deletion : - Process to deleting single or more than one element from an array.
- Sorting : - Process of arrangement of elements in any particular order.
- Traversal : - It is visiting each and every element of an array from start to endexactly to end exactly ones.
- Merging : - Process to combining two arrays into single one.
- Searching : - To finding particular element in array
  i.) Linear Search II.) Binary Search

An array of integers

The next program is a short example of using an array of integers.

```
#include "stdio.h"

void main( )

{

      int values[12];
```

```
        int index;

        for (index = 0;index < 12; index++)

        values[index] = 2 * (index + 4);

        for (index = 0;index < 12; index++)

        printf("The value at index = %2d is %3d\n", index, values[index]);

}
```

An array of floating point data

Now for an example of a program with an array of float type data. This program also has an extra feature to illustrate how strings can be initialized.

```
#include "stdio.h"
#include "string.h"
char name1[ ] = "First Program Title";
void main( )
{
        int index;
        int stuff[12];
        float weird[12];
        static char name2[] = "Second Program Title";
        for (index = 0; index < 12; index++)
        {
                stuff[index] = index + 10;
                weird[index] = 12.0 * (index + 7);
        }
        printf("%s\n", name1);
        printf("%s\n\n", name2);
        for (index = 0; index < 12; index++)
        printf("%5d %5d %10.3f\n", index, stuff[index], weird[index]);
}
```

The first line of the program illustrates how to initialize a string of characters. Notice that the square brackets are empty, leaving it up to the compiler to count the characters and allocate enough space for our string including the terminating NULL. Another string is initialized in the body of the program but it must be declared static here. This prevents it from being allocated as an automatic variable and allows it to retain the string once the program is started. You can think of a static declaration as a local constant.

There is nothing else new here, the variables are assigned nonsense data and the results of all the nonsense are printed out along with a header. This program should also be easy for you to follow, so study it until you are sure of what it is doing before going on to the next topic.

Notice that the array is defined in much the same way we defined an array of char in order to do the string manipulations in the last section. We have 12 integer variables to work with not counting the one named index. The names of the variables are values[0], values[1], ... , and values[11]. Next we have a loop to assign nonsense, but well defined, data to each of the 12 variables, then print all 12 out. Note carefully that each element of the array is simply an int type variable capable of storing an integer. The only difference between the variables index and values[2], for example, is in the way that you address them. You should have no trouble following this program, but be sure you understand it. Compile and run it to see if it does what you expect it to do.

Bound Checking

In C, there is no check to see if the subscript used for an array exceeds the size of the array. Data entered with a subscript exceeding the array size will simply be placed in memory outside the array; probably on top of other data, or on the program itself. This will lead to unpredictable results, to say the least, and there will be no error message to warn you that you are going beyond the array size. In some cases, the computer may just hang. Thus, the following program may turn out to be suicidal. Thus to see to it we do not reach beyond the array size, is entirely the programmer's botheration and not the compilers. For example,

```
#include<stdio.h>
void main()
{
      intnum[40],i;
      for(i=0;i<=100;i++)
      {
            num[i] = I;
      }
}
```

Storage Capacity of an array

The amount of storage required to hold an array is directly related to its type and size. For a single-dimension array, the total size in bytes is computed as shown below:

Total bytes = sizeof(data type) * size of array

C has no bounds checking on arrays. You could overwrite either end of an array and write into some other variable's data or even into the program's code. As the programmer, it is your job to provide bounds checking where needed. For example, the code below will compile without error, but is incorrect because the for loop will cause the array count to be overrun.

```
int count[10],i;

for(i=0;i<100;i++)  \\this will cause count to be overrun
```

One Dimensional Arrays

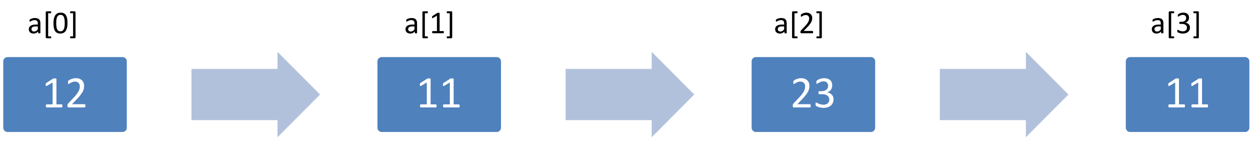
One Dimensional array has only a user defined block size. It doesn’t have any column defined in it. Let us look at the declaration of 1-D array:

```
    Data_Typearray_name[size];
```

Example: int a [100];

It is stored in a way similar to storing a string. Let us have a look how an array is stored in the memory when the array is

```
int a[4]={12,11,23,11};
```



Implementing 1-dimensional array

In C an array variable is implemented as a pointer variable, so the bracketsautomatically imply that variable is a pointer.

The address of the first location in array is called base address of array, denotedby base(arr_name)

*Example:*int b [100];

the type of b is "pointer to integer" or int* .

Implementing an array of varying-sized elements

If not the all elements have the same size, a different implementation must be used:

- Reserve a contiguous set of memory location. Each of which hold an address. The contents of each memory location are the address of the varying-length array element in some other portion of memory.
- Keep all fixed length portion of the elements in the contiguous array.

Arrays as Parameters

Since array variable is a pointer, array parameters are passed by reference.Passing an array by reference rather than by value is more efficient in both timeand space and we will discuss it later in functions.

Character strings in C

We know that a string is an array of characters and each string terminated by NULL character.The NULL character automatically appended to the end of string when it is storedwithin a program.The NULL character is denoted by escape sequence \0.

A string represents an array with lower bound = 0, and upper bound = the numberof character in the string.

Example:

"HELLO THERE" is an array of 12 characters.

(Blank(Space) and '\0'(NULL) each counts as a character)

Note:

How to calculate the address of an element in array?
Suppose we have array b, int b[100];
base(b) : The address of the first location in array b.
esize : The size of each element of the array.
The reference of the element b[0]= base(b)
The reference of the element b[1]= base(b)+esize
Then the reference of the element b[i]= base(b)+i*esize

Ques. WAP to implement the operation of searching in 1-D array.

```
#include<stdio.h>
#include<conio.h>
void main()
{
      inti,f=0,a[10],n,item;
      clrscr();
      printf("\nEnter the size of 1-D Array between 1-10 ");
      scanf("%d",&n);
      for(i=0;i<n;i++)
      {
            printf("\nEnter %d element ",i+1);
            scanf("%d",&a[i]);
      }
      printf("\nEnter element You want to Search ");
      scanf("%d",&item);
      for(i=0;i<=n;i++)
      {
            if(a[i]==item)
            {
                  f=i+1;
            }
      }
      if(f!=0)
```

```
	{
		printf("\nElement found in array at position %d",f);
	}
	else
	{
		printf("\nElement not found");
	}
	getch();
}
```

Ques. **WAP to implement the operation of sorting.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
	int x[10],n,i,j,t,p,flag;	//variable declaration
	clrscr();
	start:				//start label
	printf("How much numbers are to be sorted(<10) : ");
	scanf("%d",&n);
	if(n>10)			//case – array range
	{
		printf("\n\n\t\tArray out of limit!!! Please enter valid Input...");
		getch();
		clrscr();
		goto start;
	}
	for(i=0;i<=n-1;i++)		//loop – scanning
	{
		printf("Enter %d Element : ",i+1);
		scanf("%d",&x[i]);
	}
	for(p=1;p<n-1;p++)		//loop – sorting
	{
		flag=0;
		for(i=0;i<n-1-p;i++)
		{
			if(x[i]>x[i+1])
			{
				t=x[i];
				x[i]=x[i+1];
				x[i+1]=t;
				flag=1;
			}
		}
		if(flag==0)
		{
			break;
		}
	}
	printf("\n\nArray after sorting : \n");
```

```
      for(i=0;i<n;i++)              //loop – printing
      {
            printf("Element %d : %d\n",i+1,x[i]);
      }
      printf("\n\n\t\t\tPress any key to QUIT");
      getch();
}
```

Ques. WAP to implement the operation of Insertion.

```
#include<stdio.h>
#include<conio.h>
void main()
{
      inti,a[10],n,item,pos;
      clrscr();
      printf("\nEnter the size of 1-D Array between 1-10 ");
      scanf("%d",&n);
      for(i=0;i<n;i++)
      {
            printf("\nEnter %d element ",i+1);
            scanf("%d",&a[i]);
      }
      printf("\nEnter position of element u want to insert ");
      scanf("%d",&pos);
      printf("\nEnter element You want to Insert ");
      scanf("%d",&item);
      if(pos<=n)
      {
            for(i=n;i>=pos-1;i--)
            {
                  if(i==pos-1)
                  {
                        a[i]=item;
                        n++;
                  }
                  else
                  {
                        a[i]=a[i-1];
                  }
            }
      }
      for(i=0;i<n;i++)
      {
                  printf("\n%d ",a[i]);
      }
      getch();
}
```

Ques. WAP to implement the operation of deletion.

```
#include<stdio.h>
#include<conio.h>
void main()
{
      int i,a[10],n,item,pos;
      clrscr();
      printf("\nEnter the size of 1-D Array between 1-10 ");
      scanf("%d",&n);
      for(i=0;i<n;i++)
      {
            printf("\nEnter %d element ",i+1);
            scanf("%d",&a[i]);
      }
      printf("\nEnter position of element u want to insert ");
      scanf("%d",&pos);
      pos--;
      printf("\nEnter element You want to Insert ");
      scanf("%d",&item);
      if(pos<n)
      {
            for(i=n;i>=pos;i--)
            {
                  if(i==pos)
                  {
                        a[i]=item;
                        n++;
                  }
                  else
                  {
                        a[i]=a[i-1];
                  }
            }
      }
      for(i=0;i<n;i++)
      {
            printf("\n%d ",a[i]);
      }
      getch();
}
```

Ques. WAP to implement the operation of merging.

```
#include<stdio.h>
#include<conio.h>
void main()
{
      int a[50],b[50],n1,n2,i,j,merge,s;
      clrscr();
      printf("Enter the size of first array=");
      scanf("%d",&n1);
      printf("\nEnter the size of second array=");
      scanf("%d",&n2);
      for(i=0;i<n1;i++)
      {
            printf("\nEnter %d element of first array=",i+1);
            scanf("%d",&a[i]);
      }
      for(i=0;i<n2;i++)
      {
            printf("\nEnter %d element of second array=",i+1);
            scanf("%d",&b[i]);
      }
      merge=n1+n2;
      for(i=n1;i<merge;i++)
      {
            a[i]=b[i-n1];
      }
      printf("\nthe now MERGING of array is:");
      for(i=0;i<merge;i++)
      {
            printf("\n%d",a[i]);
      }
      printf("\nfor sorting element press any key");
      getch();
      for(j=0;j<merge;j++)
      {
            if(j==merge-1)
            {
                  clrscr();
                  printf("the sorting element is:");
            }
            for(i=0;i<merge;i++)
            {
                  if(a[i]<a[i+1])
                  {
                        printf("\n%d",a[i]);
                  }
                  else
                  {
                        s=a[i+1];
                        a[i+1]=a[i];
                        a[i]=s;
                        printf("\n%d",a[i]);
                  }
            }
      }
      getch();
}
```

Two-Dimensional Arrays

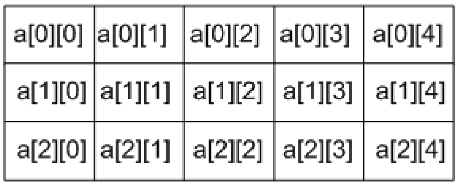
In C we can have arrays of two or more dimensions. The two dimensional array is also known as a matrix. A two dimensional array is a grid having rows and columns in which each element is specified by two subscripts. It is the simplest of multidimensional arrays. For example, An array a[m][n] is an m by n table having m rows and n columns containing m x n elements. The size of the array (total number of elements) is obtained by calculating m x n.

Declaration

```
Datatypearr_name[rows][cols];
```

Example: `inta[3][5];`

The array will be stored in the form given below:



a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]

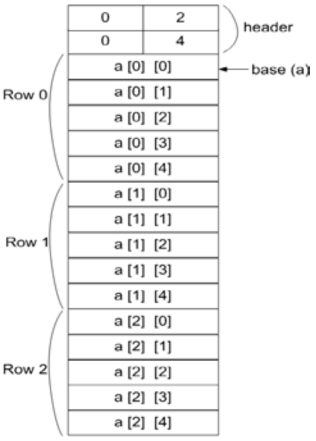
The number of rows or columns is called the range of the dimension.

a to w – dimensional array is a logical data structure that is useful in programming and problem solving.

Implementing two dimensional array

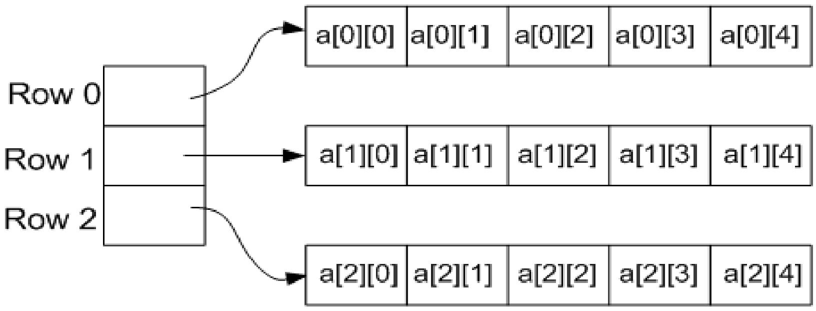
Computer memory is usually linear (one-dimensional array).We need to transform to w dimensional reference to linear representation.Two methods to representing a to w dimensional array in memory: Row-major & array of pointers.

Row-major



Representing a tow dimensional array

Array of pointers



Calculatetheaddressofanelementin2Dimensional-array(using row-major method):

Suppose we have:

`intar[r1][r2];`
r1=number of rows,
r2=number of columns.
base(ar) : the address of first element in array
esize : the size of each element in ar.
Now we want to calculate the address of the element ar[i1][i2]:
The address of the first element in rowi1=base(ar)+i1*r2*esize.
The n the address of ar[i1][i2]=base(ar)+(i1*r2+i2)*esize

Ques. WAP to print a Matrix.

```
#include<stdio.h>
#include<conio.h>
void main()
{
      int a[3][3],b,c,i,j;
      clrscr();
      //printf("Enter the size of Matrix(<10) i,j : ");
      //scanf("%d""%d",&b,&c);
      for(i=0;i<3;i++)
```

```
	{
		for(j=0;j<3;j++)
		{
			printf("Enter element (%d,%d) : ",i,j);
			scanf("%d",&a[i][j]);
		}
	}
	printf("\n\nPrintable Matrix is : \n");
	for(i=0;i<3;i++)
	{
		for(j=0;j<3;j++)
		{

			printf("%d\t",a[i][j]);
		}
		printf("\n");
	}
	getch();
}
```

Ques. WAP to add two matrixes.

```
#include<stdio.h>
#include<conio.h>
void main()
{
	int a[10][10],b[10][10],d[10][10],r,c,r1,c1,i,j;
	clrscr();
	printf("\nenter the number of row of first matrix=");
	scanf("%d",&r);
	printf("\nenter the number of column of first matrix=");
	scanf("%d",&c);
	printf("\nenter the number of rows of second matrix=");
	scanf("%d",&r1);
	printf("\nenter the number of columns of second matrix=");
	scanf("%d",&c1);
	if(r==r1&&c==c1)
	{
		for(i=0;i<r;i++)
		{
			for(j=0;j<c;j++)
			{
				printf("\nenter the %d row %d colmn of first
matrix=",i,j);
				scanf("%d",&a[i][j]);
			}
		}
		printf("\nthe first matrix is:");
		for(i=0;i<r;i++)
		{
			printf("\n");
			for(j=0;j<c;j++)
```

```
                  {
                        printf("\t");
                        printf("%d",a[i][j]);
                  }
            }
            for(i=0;i<r1;i++)
            {
                  for(j=0;j<c1;j++)
                  {
                        printf("\nenter the %d row %d colmn of second 
matrix=",i,j);
                        scanf("%d",&b[i][j]);
                  }
            }
            printf("\n\nthe second matrix is:");
            for(i=0;i<r1;i++)
            {
                  printf("\n");
                  for(j=0;j<c1;j++)
                  {
                        printf("\t");
                        printf("%d",b[i][j]);
                  }
            }
            printf("\n\n\n\n\t\tthe addition of two matrix is:");
            for(i=0;i<r1;i++)
            {
                  printf("\n");
                  for(j=0;j<c1;j++)
                  {
                        d[i][j]=0;
                        d[i][j]=a[i][j]+b[i][j];
                        printf("\t");
                        printf("%d",d[i][j]);
                  }
            }
      }
      else
      {
            printf("\nmatrix is no possible");
      }
      getch();
}
```

**Ques.	WAP to subtract a matrix from another matrix.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
      int a[10][10],b[10][10],d[10][10],r,c,r1,c1,i,j;
      clrscr();
      printf("\nenter the number of row of first matrix=");
      scanf("%d",&r);
      printf("\nenter the number of column of first matrix=");
      scanf("%d",&c);
      printf("\nenter the number of rows of second matrix=");
      scanf("%d",&r1);
      printf("\nenter the number of columns of second matrix=");  scanf("%d",&c1);
      if(r==r1&&c==c1)
      {
            for(i=0;i<r;i++)
            {
                  for(j=0;j<c;j++)
                  {
                        printf("\nenter the %d row %d colmn of first 
matrix=",i,j);
                        scanf("%d",&a[i][j]);
                  }
            }
            printf("\nthe first matrix is:");
            for(i=0;i<r;i++)
            {
                  printf("\n");
                  for(j=0;j<c;j++)
                  {
                        printf("\t");
                        printf("%d",a[i][j]);
                  }
            }
            for(i=0;i<r1;i++)
            {
                  for(j=0;j<c1;j++)
                  {
                        printf("\nenter the %d row %d colmn of second 
matrix=",i,j);
                        scanf("%d",&b[i][j]);
                  }
            }
            printf("\n\nthe second matrix is:");
            for(i=0;i<r1;i++)
            {
                  printf("\n");
                  for(j=0;j<c1;j++)
                  {
                        printf("\t");
                        printf("%d",b[i][j]);
                  }
            }
            printf("\n\n\n\n\t\tthe subtraction of two matrix is:\n\n");
            for(i=0;i<r1;i++)
            {
                  printf("\n");
                  for(j=0;j<c1;j++)
                  {
                        d[i][j]=0;
                        d[i][j]=a[i][j]-b[i][j];
                        printf("\t");
                        printf("%d",d[i][j]);
                  }
            }
      }
      else
      {
            printf("\nmatrix is no possible");
      }
      getch();
}
```

Ques. WAP to multiply two matrices.

```
#include<stdio.h>
#include<conio.h>
void main()
{
      int x[5][5], y[5][5], z[5][5], i, j, r, c, R, C,m,n,p;      //variable
declaration
      clrscr();
      start:                              //start : label
      printf("Enter Row of first Matrix : ");
      scanf("%d",&r);
      printf("Enter column of first Matrix : ");
      scanf("%d",&c);
      if((r>5)||(c>5))              //case – Matrix range
      {
            printf("\n\n\t\tMatrix out of Range! Enter Valid range...");
            getch();
            clrscr();
            goto start;
      }
      printf("\nEnter first Matrix \n\n");
      for(i=0;i<r;i++)              //loop – Mat1 scan
      {
            for(j=0;j<c;j++)
            {
                  printf("Enter element(%d,%d) : ",i+1,j+1);
                  scanf("%d", &x[i][j]);
            }
      }
      start2:                             //start2 : label
      clrscr();
      printf("Enter Row of second Matrix : ");
      scanf("%d",&R);
      printf("Enter column of second Matrix : ");
      scanf("%d",&C);
      if((R>5)||(C>5))              //case – Matrix Range
```

```
	{
		printf("\n\n\t\tMatrix out of Range! Enter Valid range...");
		getch();
		goto start2;
	}
	if(c!=R)
	{
		printf("\n\n\t\tCan't Multiply... Order Mismatch\nEnter Correct
Order!!!");
		getch();
		goto start2;
	}
	printf("\nEnter second Matrix \n\n");
	m=c;
	n=R;
	for(i=0;i<R;i++)		//loop – Mat2 scan
	{
		for(j=0;j<C;j++)
		{
			printf("Enter element(%d,%d) : ",i+1,j+1);
			scanf("%d", &y[i][j]);
		}
	}
	for(i=0;i<r;i++)		//Mat3 = Mat1*Mat2
	{
		for(j=0;j<C;j++)
		{
			z[i][j]=0;
			for(p=0;p<c;p++)
			{
				z[i][j]=z[i][j]+(x[i][p])*(y[p][j]);
			}
		}
	}
	clrscr();
	printf("Resultant Matrix is : \n");
	for(i=0;i<m;i++)		//Loop – Mat3 print
	{
		for(j=0;j<n;j++)
		{
			printf("%d\t", z[i][j]);
		}
		printf("\n");
	}
	printf("\n\n\t\t\tPress any key to QUIT");
	getch();
}
```

Array of Strings

A String is an array of char objects. An array of string can be declared and handled like a 2d(two dimensional) arrays.A String is an array of char objects. An array of string can be declared and handled like a 2d (two dimensional) arrays. You can see in the given example that we have declared a 2 dimensional character array consisting of three 'rows' and twelve 'columns'. The array is initialized with three character strings. In C, a format specifier %s is used with the printf to print the string.

Declaration

```
charvariable_name[no. of string][length of string];
```

Example Program of Array of Strings :

```
#include <stdio.h>
#include <conio.h>
void main()
{
      clrscr();
      chararr[3][12]= { "Rose", "India", "technologies" };
      printf("Array of String is = %s,%s,%s\n", arr[0], arr[1], arr[2]);
      getch();
}
```

We can also use the array of stings by using a loop.

Ques. WAP to implement print the list of student name using string.

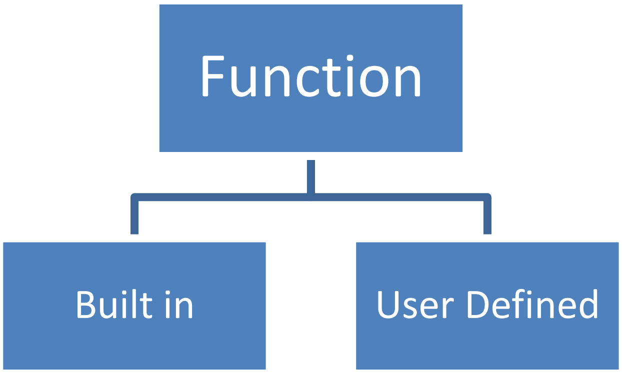
```
#include"stdio.h"
#include"conio.h"
void main()
{
      char name[10][10];
      int i,n;
      clrscr();
      printf("\nenter number of student:");
      scanf("%d",&n);
      for(i=0;i<n;i++)
      {
            printf("\n%dname:",i+1);
            scanf("%s",&name[i][0]);
      }
      printf("\nName:");
      for(i=0;i<n;i++)
      {
            printf("\n%s",name[i]);
      }
      getch();
}
```

11. FUNCTION

A function is a module or block of program code which deals with a particular task. Making functions is a way of isolating one block of code from other independent blocks of code.

Functions serve two purposes, They allow a programmer to say: ‘this piece of code does a specific job which stands by itself and should not be mixed up with anything else',Second they make a block of code reusable since a function can be reused in many different contexts without repeating parts of the program text.

A function can take a number of parameters, do required processing and then return a value. There may be a function which does not return any value.



You already have seen couple of built-in functions like printf() those are built in function; Similar way you can define your own functions in C language, and those were User Defined Functions.

Consider the following chunk of code

```
int total = 10;

printf("Hello World");

total = total + l;
```

To turn it into a function you simply wrap the code in a pair of curly brackets to convert it into a single compound statement and write the name that you want to give it in front of the brackets:

```
int Demo()
{
      int total = 10;
      printf("Hello World");
      total = total + l;
}
```

Curved brackets after the function's name are required. You can pass one or more parameters to a function as follows:

```
int Demo(int par1, int par2)
{
      int total = 10;
      printf("Hello World");
      total = total + l;
}
```

By default function does not return anything. But you can make a function to return any value as follows:

```
int Demo( int par1, int par2)
{
      int total = 10;
      printf("Hello World");
      total = total + l;
      return total;
}
```

A return keyword is used to return a value and data type of the returned value is specified before the name of function. In this case function returns total which is int type. If a function does not return a value then void keyword can be used as return value.Once you have defined your function you can use it within a program:

```
main()
{
      Demo();
}
```

Ques. WAP to add two nos. by using return statement.

```
#include<stdio.h>
#include<conio.h>
void main()
{
      int a,b;
      clrscr();
      printf("Enter first no. : ");
      scanf("%d",&a);
      printf("Enter second no. : ");
      scanf("%d",&b);
      printf("\na + b = %d",add(a,b));
      getch();
}
int add(int x, int y)
{
      return x+y;
}
```

Functions and Variables

Each function behaves the same way as C language standard function main(). So a function will have its own local variables defined. In the above example total variable is local to the function Demo.

A global variable can be accessed in any function in similar way it is accessed in main() function.

Declaration and Definition

When a function is defined at any place in the program then it is called function definition. At the time of definition of a function actual logic is implemented with-in the function.A function declaration does not have any body and they just have their interfaces.A function declaration is usually declared at the top of a C source file, or in a separate header file.

A function declaration is sometime called function prototype or function signature. For the above Demo() function which returns an integer, and takes two parameters a function declaration will be as follows:

```
int Demo( int par1, int par2);
```

Passing Parameters to a Function

There are two ways to pass parameters to a function:

Call by Value: In call by value, actual argument are sent to function, its value is copied in formal argument.So, duplicate copies of variable name, if there is change in formal argument in function it will not be reflected in actual argument in main program.

Call by Reference: In Call by reference, address of actual argument is sent to function. The address is sent to function call using address operator(&).The address of actual argument is received by pointers in function whenever there is change it will be same in main as well in the function.

Here are two programs to understand the difference: First example is for Call by value:

Ques. WAP to implement Call by Value for swapping two integers.

```
#include <stdio.h>
void swap( int p1, int p2 );
int main()
{
      int a = 10;
      int b = 20;
      printf("Before: Value of a = %d and value of b = %d\n", a, b );
      swap( a, b );
      printf("After: Value of a = %d and value of b = %d\n", a, b );
}
void swap( int p1, int p2 )
{
      int t;
      t = p2;
      p2 = p1;
      p1 = t;
      printf("Value of a (p1) = %d and value of b(p2) = %d\n", p1, p2 );
}
```

Here is the result produced by the above example. Here the values of a and b remain unchanged before calling swap function and after calling swap function.

```
Before: Value of a = 10 and value of b = 20
Value of a (p1) = 20 and value of b (p2) = 10
After: Value of a = 10 and value of b = 20
```

Following is the example which demonstrate the concept of call by reference

Ques. WAP to implement Call by Reference for swapping two integers.

```
#include <stdio.h>
/* function declaration goes here.*/
void swap( int *p1, int *p2 );
int main()
{
      int a = 10;
      int b = 20;
      printf("Before: Value of a = %d and value of b = %d\n", a, b );
      swap(&a, &b );
      printf("After: Value of a = %d and value of b = %d\n", a, b );
}
void swap( int *p1, int *p2 )
{
      int t;
      t = *p2;
      *p2 = *p1;
      *p1 = t;
      printf("Value of a (p1) = %d and value of b(p2) = %d\n", *p1, *p2 );
}
```

Here is the result produced by the above example. Here the values of a and b are changes after calling swap function.

```
Before: Value of a = 10 and value of b = 20
Value of a (p1) = 20 and value of b (p2) = 10
After: Value of a = 20 and value of b = 10
```

Recursion

The meaning of recursion is defining something terms of itself. In functions, recursion is used. Recursion is used when function calls itself again & again.

Ques. Program to print factorial of any no. entered by user using recursion

```
#include<stdio.h>
#include<conio.h>
void main()
{
      int n;
      clrscr();
      printf("enter no ");
      scanf("%d",&n);
      printf("the result is %d",fact(n));
      getch();
}
int fact(num)
{
      int f;
      if(num==1)
      {
            return 1;
      }
      else
      {
            f=num*fact(num-1);
      }
      return f;
}
```

Passing Array Elements to a Function

Array elements can be passed to a function by calling the function by value, or by reference. In the call by value, we pass value of array elements to the function, whereas in the call by reference, we pass addresses of array elements to the function. These two calls are illustrated below:

```
/*Demonstration of passing array to a Function by call by value */
#include<stdio.h>
void main()
{
      int i;
      int marks[] = {55, 56, 43, 35, 45, 32, 89, 99};
      for(i=0;i<=7;i++)
      {
            display(marks[i]);
      }
      getch();
}
int display(int m)
{
      printf(“%d”,m);
}
```

Here we are passing an individual array element at a time to the function display() and getting it printed in the function display(). Note that, since at a time only one element is passed, this element is collected in an ordinary integer variable m, in the function display().

And now the call by reference.

```
/*Demonstration of passing array to a function by call by reference */
#include<stdio.h>
void main()
{
      int i;
      int marks[] = {55, 56, 43, 35, 45, 32, 89, 99};
      for(i=0;i<=7;i++)
      {
            display(marks[i]);
      }
      getch();
}
int display(int*m)
{
      printf(“%d”,*m);
}
```

Here we are passing addresses of individual array elements to the function display(). Hence, the variable in which this address is collected (m), is declared as a pointer variable. And since m contains the address of array element, to print out the array element, we are using the value at address operator(*).

Command Line Argument

In C it is possible to accept command line arguments. Command-line arguments are given after the name of a program in command-line operating systems like DOS or Linux, and are passed in to the program from the operating system. To use command line arguments in your program, you must first understand the full declaration of the main function, which previously has accepted no arguments. In fact, main can actually accept two arguments: one argument is number of command line arguments, and the other argument is a full list of all of the command line arguments.

Every program must have a function main. Till now, we know that main function takes no argument but the empty () in the main() may contain special arguments that allows parameters to be passed to the main() from operating system. Two arguments are passed to the main function.

The full declaration of main looks like this:

```
int main ( intargc, char *argv[] )
```

The integer, argc is the argument count. It is the number of arguments passed into the program from the command line, including the name of the program.

The array of character pointers is the listing of all the arguments. argv[0] is the name of the program, or an empty string if the name is not available. After that, every element number less than argc is a command line argument. You can use each argv element just like a string, or use argv as a two dimensional array. argv[argc] is a null pointer.

In short we can say :

1. argc : Integer type argument indicate the no. of parameters passed.
2. agrv : Array of strings. Each string in this array will represent a parameter thatis passed to the main function.

How could this be used? Almost any program that wants its parameters to be set when it is executed would use this. One common use is to write a function that takes the name of a file and outputs the entire text of it onto the screen. Here's is an example program for passing arguments to the main function.

Ques. WAP to implement Command Line Argument.

```
#include <stdio.h>
void main ( intargc, char *argv[] )
{
      if ( argc != 2 ) /* argc should be 2 for correct execution */
      {     /* We print argv[0] assuming it is the program name */
            printf( "usage: %s filename", argv[0] );
      }
      else
      {     // We assume argv[1] is a filename to open
      FILE *file = fopen(argv[1], "r" );
      /* fopen returns 0, the NULL pointer, on failure */
            if ( file == 0 )
      {
                  printf( "Could not open file\n" );
      }
            else
      {
                  int x;
/* read one character at a time from file, stopping at EOF, which indicates the
end of the file.  Note that the idiom of "assign to a variable, check the value"
used below works because the assignment statement evaluates to the value assigned.
*/
      while  ( ( x = fgetc( file ) ) != EOF )
            {
                        printf( "%c", x );
      }
                  fclose( file );
      }
      }
}
```

This program is fairly short, but it incorporates the full version of main and even performs a useful function. It first checks to ensure the user added the second argument, theoretically a file name. The program then checks to see if the file is valid by trying to open it. This is a standard operation, and if it results in the file being opened, then the return value of fopen will be a valid FILE*; otherwise, it will be 0, the NULL pointer. After that, we just execute a loop to print out one character at a time from the file. The code is self-explanatory, but is littered with comments; you should have no trouble understanding its operation this far.

Now Let us take a simple example of the Command Line Argument.

```
void main(int a, char *b[])
{
      int i;
      clrscr();
      printf(“\nNo. Of arguments %d”,a);
      printf(“\nName of Program %s”,b[0]);
      for(i=1;i<a;i++)
      {
            printf(“\n%s”,b[i]);
      }
      getch();
}
```

Ques. WAP to implement print Command Line Argument.

```
#include"stdio.h"
#include"conio.h"
void main(int a,char *b[])
{
      int i;
      clrscr();
      printf("\nTotal no of arg : %d",a-1);
      for(i=1;i<a;i++)
      {
            printf("\n%s",b[i]);
      }
      getch();
}
```

Storage Class Specifier for Variable (Scope of Variable)

There are two different ways to characteristics variables in a function.

1. By Data type
2. By Storage Class

Data type referred to the type of information while storage class refers to the lifetime of variable & its scope within the program.

A variable in C can have anyone of the four storage class:

1. Automatic Variable
2. External Variable
3. Static Variable
4. Register Variable

1. **Automatic or Internal Variable:** The scope of an automatic variable is confined to that function in which it is declared. It is created when function is called & destroyed and deployed automatically when function is exited. Hence the name is automatic. For example,

```
#include<stdio.h>
void main()
{
      int m = 1000;
      f2();
      printf("%d",m);
}
f1()
{
      int m = 10;
      printf("%d,m);
}
f2()
{
      int m = 100;
      f1();
      printf("%d",m);
}
```

Advantage:

1. Tell us the scope
2. Tell us the lifetime

2. **External Variable:** An external variable is also known as global variable. It is not confined to a single function. Its scope extends from the point of definition through the remainder of the program.

   External Variables can be accessed from any function that falls within there scope. They are declared outside a function. If local variable & a global variable have a same name local variable will have residence over global in the function where it is declared. For example,

```
int m;
main()
{
      m=10;
      printf("%d",m);
      printf("%d",f1());
      printf("%d",f2());
      printf("%d",f3());
}
f1()
{
      m = m + 10;
      return m;
}
f2()
{
      int m = 1;
      return m;
}
f3()
{
      m = m + 10;
      return m;
}
```

External Declaration: In this program the main cannot excess the variable if as it has been declared after the main function. This function can be solved by declaring the variable with the storage class extern. For example,

```
main()
{
      externint y;
      ...;
      ...;
      ...;
}
function1()
{
      externint y;
      ...;
}
int y;
function2()
{
      y = y + 10;
}
```

Here the variable declared is either external variable and declared in the function with extern keyword or the variable is declared outside the function without any keyword. Those are external variable.

3. **Static Variable:** Static Variable is defined within a function in a same manner as a automatic variable except that the variable declaration must beyond with the static storage class destination. For example,

```
      static int a;
      static float y;
```

A static variable maybe either an internal type or external type depending upon the place of declaration.

```
main()
{
           int i;
           for(i=1;i<3;i++)
           stat();
}
stat()
{
           staticint x = 0;
           x=x+1;
           printf("%d",x);
}
```

4. **Register Variable:** We can tell the compiler that a variable should be kept in one of the machine's registers instead of keeping in the memory (where normal variable are stored).

Since, a register excess & keeping a frequently accessed variable in the register will lead to faster execution of program. For example,

```
    register int count;
```

Since, only a general variables can be placed in a register. It is important to carefully select the variable for this purpose. However C will automatically convert register variables into non-register variable once when the limit is reached.

12. POINTERS

A pointer is a special kind of variable. Pointers are designed for storing memory address i.e. the address of another variable. Declaring a pointer is the same as declaring a normal variable except you stick an asterisk '*' in front of the variables identifier.

A pointer is a special type variable which can hold the address of another variable. A pointer can also be used for handling user defined data type like structure, union, array etc.

- There are two new operators you will need to know to work with pointers. The "address of" operator '&' and the "dereferencing" operator '*'. Both are prefix unary operators.
- When you place an ampersand in front of a variable you will get its address, this can be stored in a pointer variable.
- When you place an asterisk in front of a pointer you will get the value at the memory address pointed to.

Declaring & Initializing Pointer

For declaring pointer * operator is used, it is also called pointer operator or indirection pointer. The declaration of a pointer is :

```
datatype *variable_name;
```

1. A pointer manages the memory more efficiently.
2. A pointer can be used for dynamic memory allocation and de-allocation of memory
3. Pointer saves the memory space.
4. Pointer can be used to handle function more efficiently by using concept of call by reference.
5. Pointer can handle character, strings, efficiently by using their address.
6. If pointer is integer, it will store int value.
7. If pointer is float, it will store float value.

Here is an example to understand what I have stated above.

```
#include <stdio.h>
void main()
{
        intmy_variable = 6, other_variable = 10;
        int *my_pointer;
        printf("the address of my_variable is    : %p\n", &my_variable);
        printf("the address of other_variable is : %p\n", &other_variable);
        my_pointer = &my_variable;
        printf("\nafter \"my_pointer = &my_variable\":\n");
        printf("\tthe value of my_pointer is %p\n", my_pointer);
        printf("\tthe value at that address is %d\n", *my_pointer);
        my_pointer = &other_variable;
        printf("\nafter \"my_pointer = &other_variable\":\n");
        printf("\tthe value of my_pointer is %p\n", my_pointer);
        printf("\tthe value at that address is %d\n", *my_pointer);
}
```

This will produce following result.

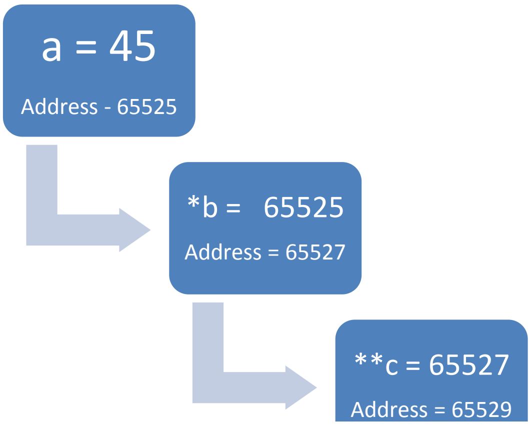
```
the address of my_variable is    : 0xbfffdac4
the address of other_variable is : 0xbfffdac0

after "my_pointer = &my_variable":
the value of my_pointer is 0xbfffdac4
the value at that address is 6

after "my_pointer = &other_variable":
the value of my_pointer is 0xbfffdac0

the value at that address is 10
```

Ques. WAP to implement pointers.

```
#include<stdio.h>
#include<conio.h>
void main()
{
      int a=45;
      int *b,**c;
      b=&a;
      c=&b;
      clrscr();
      printf("Address of a = %u",&a);
      printf("\nAddress of b = %u",b);
      printf("\nAddress of c = %u\n",*c);
      printf("\nValue of a = %u",&b);
      printf("\nValue of b = %u",c);
      printf("\nValue of c = %u\n",&c);
      printf("\nValue of b = %u",a);
      printf("\nValue of c = %u",c);
      printf("\nValue of a = %d\n",a);
      printf("\nValue of a = %d",*b);
      printf("\nValue of b = %d",*(&a));
      printf("\nValue of c = %d",**c);
      getch();
}
```

Chain Pointer

If a pointer contains the address of another pointer variable, then it is called chain pointer or pointer to pointer.



Here integer a contains the value and the pointer b is holding the address of a, in meanwhile the address of b is getting stored by another pointer c and this is known as Chain Pointer or Pointer of Pointer.

Ques. WAP to implement pointer to pointer(chain pointer).

```
#include"stdio.h"
#include"conio.h"
void main()
{
	int a,*p,**cp;
	clrscr();
	p=&a;
	cp=&p;
	printf("\n\t\t\tValue Print To The Chain Pointer");
	printf("\n\nEnter value of a:");
	scanf("%d",p);
	printf("\nAddress ofa:%u\tValue of a:%d",&a,a);
	printf("\nAddress ofa:%u\tValue of a:%d",p,*p);
	printf("\nAddress ofa:%u\tValue of a:%d",*cp,**cp);
	getch();
}
```

Pointers & Strings

String is defined as collection of character like array. Strings are represented by char data type. A pointer *s is also declared to be character data type. This pointer is assigned base address of string using argument operator. An example program to count vowels in a name by using pointers is as follows:

Ques. WAP to implement Poitners & Strings.

```
#include<stdio.h>
#include<conio.h>
void main()
{
	char name[30];
```

```
      char *p;
      int count;
      clrscr();
      printf(“Enter a name : “);
      scanf(“%s”,&name);
      p=name;
      while(*p!=”\0”)
      {
            switch(*p)
            {
                  case'A’;
                  case'a’;
                  case'E’;
                  case'e’;
                  case'I’;
                  case'i’;
                  case'O’;
                  case'o’;
                  case'U’;
                  case'u’;
                  count++;
                  break;
            }
            p++;
      }
      printf(“Total No. of vowels present in the name : %d“,count+1);
      getch();
}
```

Ques. WAP to implement print the list of student name using string pointer.

```
#include"stdio.h"
#include"conio.h"
void main()
{
      int i,n;
      char *name[5];
      clrscr();
      printf("\nenter the number of name:");
      scanf("%d",&n);
      printf("\nEnter the string\n");
      for(i=0;i<n;i++)
      {
            scanf("%s",name[i]);
      }
      printf("\nName:");
      for(i=0;i<n;i++)
      {
            printf("\n%s",name[i]);
      }
      getch();
}
```

Pointer & Function

It is a kind of operation where we passes a pointer to a function, it is kind of same to the call by reference method.

```
#include<stdio.h>
void main()
{
      int *p;
      p = ABC();
      printf(“Value is %d”,*p);
      printf(“Address is %u”,p);
}
int ABC()
{
      int a=10;
      a=a*2;
      return(&a);
}
```

Ques. WAP to implement return pointers from function.

```
#include"stdio.h"
#include"conio.h"
int *add();
void main()
{
      int a,b,*p;
      clrscr();
      printf("\n\t\t\tUsing Integer Pointer Function");
      printf("\nEnter a:");
      scanf("%d",&a);
      printf("\nEnter b:");
      scanf("%d",&b);
      p=add(&a,&b);
      printf("\nThe address of c:%u\tValue of c:%d",p,*p);
      getch();
}
int *add(int *x,int *y)
{
      int c;
      printf("\nThe address of x:%u\tValue of x:%d",&x,*x);
      printf("\nThe address of y:%u\tValue of y:%d",&y,*y);
      printf("\nThe address of a:%u\tValue of a:%d",x,*x);
      printf("\nThe address of b:%u\tValue of b:%d",y,*y);
      c=*x+*y;
      return (&c);
}
```

Pointers and Arrays

The most frequent use of pointers in C is for walking efficiently along arrays. In fact, in the implementation of an array, the array name represents the address of the zeroth element of the array, so you can't use it on the left side of an expression. For example:

```
      char                                                      *y;
      char x[100];
```

y is of type pointer to character (although it doesn't yet point anywhere). We can make y point to an element of x by either of

```
y = &x[0];
y = x;
```

Since x is the address of x[0] this is legal and consistent. Now `*y' gives x[0]. More importantly notice the following:

```
*(y+1)  gives x[1]
*(y+i)  gives x[i]
```

and the sequence

```
y = &x[0];
y++;
```

leaves y pointing at x[1].

Ques. WAP to implement the sum of array elements using pointers & show there addresses

```
#include"stdio.h"
#include"conio.h"
void main()
{
      int a[20],*p,s,sum=0,i;
      clrscr();
      printf("\nEnter size of array:\n");
      scanf("%d",&s);
      //p=&a[0];
      p=a;
      for(i=0;i<s;i++)
      {
            printf("\nEnter %d element: ",i+1);
            scanf("%d",p+i);
      }
      for(i=0;i<s;i++)
      {
            printf("\nAddres %d element is %u ",i+1,p+i);
            printf("\tValue is %d ",*(p+i));
      }
      for(i=0;i<s;i++)
      {
            sum=sum+*(p+i);
      }
      printf("\nThe sum of elements are: %d",sum);
      getch();
}
```

Ques. WAP to print the sum and average of arrays element by using pointerand the size of array element entered by user.

```
#include"stdio.h"
#include"conio.h"
void main()
{
      int a[20],*p,s,i,sum=0;
      clrscr();
      p=a;
      printf("\t\t\tSum And Average of Element");
      printf("\n\nEnter the size of element:");
      scanf("%d",&s);
      for(i=0;i<s;i++)
      {
            printf("\nEnter %d element:",i+1);
            scanf("%d",p+i);
            sum=sum+*(p+i);
      }
      printf("\n\nThe sum of elements:%d",sum);
      printf("\nThe average of elements:%d",sum/i);
      getch();
}
```

Ques. WAP to find the smallest element in an array of n elements byusing pointer.

```
#include"stdio.h"
#include"conio.h"
void main()
{
      int a[20],*p,n,i,smallest;
      clrscr();
      printf("\n\t\t\tFind Smallest Element In Array");
      printf("\n\nEnter size of elements:");
      scanf("%d",&n);
      p=a;
      for(i=0;i<n;i++)
      {
            printf("\nEnter %d element:",i+1);
            scanf("%d",p+i);
      }
      smallest=*p;
      for(i=1;i<n;i++)
      {
            if(smallest>*(p+i))
            {
                  smallest=*(p+i);
            }
      }
      printf("\nThe smallest element:%d",smallest);
      getch();
}
```

Ques. WAP to print the value using array of pointer

```
#include"stdio.h"
#include"conio.h"
void main()
{
      int a[20],*p[20],i,v;
      clrscr();
      printf("\n\t\t\tPrint Array Value Using Array Of Pointer");
      printf("\n\nHow many value u want enter:");
      scanf("%d",&v);
      for(i=0;i<v;i++)
      {
            p[i]=&a[i];
      }
      for(i=0;i<v;i++)
      {
            printf("\nEnter %d value:",i+1);
            scanf("%d",p[i]);
      }
      printf("\n\nThe print of value using array of pointer:\n\n");
      for(i=0;i<v;i++)
      {
            printf("\nThe %d value:%d",i+1,*p[i]);
      }
      getch();
}
```

Pointer Arithmetic:

C is one of the few languages that allow pointer arithmetic. In other words, you actually move the pointer reference by an arithmetic operation. For example:

```
      int x = 5, *ip = &x;
      ip++;
```

On a typical 32-bit machine, *ip would be pointing to 5 after initialization. But ip++; increments the pointer 32-bits or 4-bytes. So whatever was in the next 4-bytes, *ip would be pointing at it.

Pointer arithmetic is very useful when dealing with arrays, because arrays and pointers share a special relationship in C.

Array of Pointer to string

A pointer value always contains an address, therefore if we construct an array of pointer, which would contain a no. of address. Here is a program to define the statement:

```
char *[] = {“RAM”, “SHYAM”, “GYAN”, “RAN”, “GAN”};

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

{
      printf("%s”,name[i]);

}
```

Using Pointer Arithmetic with Arrays

Arrays occupy consecutive memory slots in the computer's memory. This is where pointer arithmetic comes in handy - if you create a pointer to the first element, incrementing it one step will make it point to the next element.

```
#include <stdio.h>
void main()
{
        int *ptr;
        intarrayInts[10] = {1,2,3,4,5,6,7,8,9,10};
        ptr = arrayInts;       /* ptr = &arrayInts[0]; is also fine */
        printf("The pointer is pointing to the first ");
        printf("array element, which is %d.\n", *ptr);
        printf("Let's increment it.....\n");
        ptr++;
        printf("Now it should point to the next element,");
        printf(" which is %d.\n", *ptr);
        printf("But suppose we point to the 3rd and 4th: %d %d.\n",
*(ptr+1),*(ptr+2));
        ptr+=2;
        printf("Now skip the next 4 to point to the 8th: %d.\n", *(ptr+=4));
        ptr--;
        printf("Did I miss out my lucky number %d?!\n", *(ptr++));
        printf("Back to the 8th it is then..... %d.\n", *ptr);
}
```

This will produce following result:

```
The pointer is pointing to the first array element, which is 1.
Let's increment it.....
Now it should point to the next element, which is 2.
But suppose we point to the 3rd and 4th: 3 4.
Now skip the next 4 to point to the 8th: 8.
Did I miss out my lucky number 7?!
Back to the 8th it is then..... 8.
```

Pointers and const Type Qualifier

- The **const** type qualifier can make things a little confusing when it is used with pointer declarations.
- The below example:

```
constint *constip;      /* The pointer *ip is constand it points at is also cont
*/
int *constip;           /* The pointer *ip is const                 */
constint *ip;           /* What *ip is pointing at is const         */

int *ip;                /* Nothing is const                         */
```

As you can see, you must be careful when specifying the **const qualifier** when using pointers.

Modifying Variables Using Pointers

You know how to access the value pointed to using the dereference operator, but you can also modify the content of variables. To achieve this, put the de-referenced pointer on the left of the assignment operator, as shown in this example, which uses an array:

```
#include <stdio.h>
void main()
{
        char *ptr;
        chararrayChars[8] = {'F','r','i','e','n','d','s','\0'};
        ptr = arrayChars;
        printf("The array reads %s.\n", arrayChars);
        printf("Let's change it..... ");
        *ptr = 'f'; /* ptr points to the first element */
        printf(" now it reads %s.\n", arrayChars);
        printf("The 3rd character of the array is %c.\n", *(ptr+=2));
        printf("Let's change it again..... ");
        *(ptr - 1) = ' ';
        printf("Now it reads %s.\n", arrayChars);
}
```

This will produce following result:

```
The array reads Friends.
Let's change it..... now it reads friends.
The 3rd character of the array is i.

Let's change it again..... Now it reads f iends.
```

Generic Pointers (void Pointer)

When a variable is declared as being a pointer to type void it is known as a generic pointer. Since you cannot have a variable of type void, the pointer will not point to any data and therefore cannot be de-referenced. It is still a pointer though, to use it you just have to cast it to another kind of pointer first. Hence the term Generic pointer. This is very useful when you want a pointer to point to data of different types at different times.

Try the following code to understand Generic Pointers.

```
#include <stdio.h>
void main()
{
        int i;
        char c;
        void *the_data;
        i = 6;
        c = 'a';
        the_data = &i;
        printf("the_data points to the integer value %d\n",*(int*) the_data);
        the_data = &c;
        printf("the_data now points to the character %c\n",*(char*) the_data);
}
```

NOTE-1 : Here in first print statement, the_data is prefixed by ***(int*)**. This is called type casting in C language.Type is used to caste a variable from one data type to another datatype to make it compatible to the lvalue.

NOTE-2 : lvalue is something which is used to left side of a statement and in which we can assign some value. A constant can't be an lvalue because we can not assign any value in contact. For example x = y, here x is lvalue and y is rvalue.

However, above example will produce following result:

```
the_data points to the integer value 6

the_data now points to the character a
```

Primary Memory Allocation

It is a process of allocating and de-allocating memory during the running of program. C allocates desired amount of memory to variables and arrays defined in a program on the memory stack. However, in some cases the exact amount of memory is not known at the compile time. The memory must be allocated only at the run time. Runtime memory allocation is known as dynamic memory allocation.

A block of memory can be accessed using a pointer; following functions are used in dynamic memory allocation. These are defined in header file “alloc.h”.

1. **malloc()** : It is used to allocate a single block of memory to store data, the data can be any data type. It can be used to assign address of 1st memory location of block to pointer. Declaration of the malloc() is :

```
    p=(data type*)malloc(size of array);
```

Where p is a pointer variable, data type is data type stored in memory and size refer to no. of bytes allocating. Eg.

```
int *p;
p=(int*)malloc(10);
or
p=(int*)malloc(n*sizeof(int));
```

2. **calloc()** : It is used to allocate memory in a no. of blocks of same size during the execution of the program. Address of 1st element of data structure is assigned to pointer. Declaration of the calloc() is :

```
    p=(datatype*)calloc(a,b);
```

Where p is a pointer variable, data type is data type stored in memory, a refers to the no. of blocks to be allocated and b is the no. of bytes each block will be allocated. Eg.

```
int *p;
```

```
p=(int*)calloc(5,2*sizeof(int));
```

3. **realloc()** : With the function realloc, you can change the size of the allocated area once. Has the following form.

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

The first argument specifies the address of an area that is currently allocated to the size in bytes of the modified second argument. Change the size, the return value is returned in re-allocated address space. Otherwise it returns NULL.A Simple example program of realloc is:

```
#include<stdio.h>
void main ()
{
      int * p1, * p2;
      p1 = (int *) calloc (5, sizeof (int));    /* number of elements in an
array of type 5 int*/
      p2 = (int *) realloc (p1, sizeof (int));  /* re-acquire the space of
one type*/
      if (p2 == NULL)                     // check if successful
      {
            free (p1); /*if it fails to get realloc, the region remains
valid since the original*/
            return 0;
      }
      p1 = NULL; /*safety measure p1 is realloc () because it is released
inside, Keep assigning NULL to clear the other can not use To do
something*/
      free (p2);
      return 0;
}
```

4. **free()** : It is used to release the memory space allocated using calloc() or malloc() functions. The declaration of free() is:

```
    free(variable);
```

example

```
free(p);  //it will free memory space taken by p
```

Ques. WAP to implement allocate memory using malloc function to find greater element in array

```
#include"stdio.h"
#include"conio.h"
#include"alloc.h"
void main()
{
      int *p,i,n,largest;
      clrscr();
      printf("\n\t\t\tFind Largest Element In Array");
      printf("\n\nHow many element u want enter in array:");
      scanf("%d",&n);
      p=(int *)malloc(n*2);
      for(i=0;i<n;i++)
      {
            printf("\nEnter %d element in array:",i+1);
            scanf("%d",p+i);
      }
      largest=*p;
      for(i=1;i<n;i++)
      {
            if(largest<*(p+i))
            {
                  largest=*(p+i);
            }
      }
      printf("\nThe largest element in array:%d",largest);
      getch();
}
```

Ques. WAP to implement concept of dynamic memory allocation by using pointer & find out the Sum of n elements of array using dynamic memory allocation

```
#include"stdio.h"
#include"conio.h"
#include"alloc.h"
void main()
{
      int *p,n,i,sum=0;
      clrscr();
      printf("\n\t\t\tSum Of Element In Accurate Memory Space");
      printf("\n\nHow many element u want enter:");
      scanf("%d",&n);
      p=(int *)malloc(n*2);
      for(i=0;i<n;i++)
      {
            printf("\nEnter %d element:",i+1);
            scanf("%d",p+i);
            sum=sum+*(p+i);
      }
      for(i=0;i<n;i++)
      {
            printf("\nAddress of %d byte:%u\tValue of %d
byte:%d",i+1,p+i,i+1,*(p+i));
      }
      printf("\n\n\tAddition of elements:%d",sum);
      getch();
}
```

13. STRING MANIPULATION

A *string* is a group of characters, usually letters of the alphabet. In order to format your printout in such a way that it looks nice, has meaningful titles and names, and is aesthetically pleasing to you and the people using the output of your program, you need the ability to output text data. We have used strings extensively already, without actually defining them. A complete definition of a string is 'a sequence of char type data terminated by a NULL character,'

When C is going to use a string of data in some way, either to compare it with another, output it, copy it to another string, or whatever, the functions are set up to do what they are called to do until a NULL character (which is usually a character with a zero ASCII code number) is detected. You should also recall that the char type is really a special form of integer – one that stores the ASCII code numbers which represent characters and symbols.

An array (as we shall discover shortly) is a series of homogeneous pieces of data that are all identical in type. The data type can be quite complex as we will see when we get to the section of this module discussing structures. A string is simply a special case of an array, an array of char type data. The best way to see these principles is by use of an example.

```
#include "stdio.h"
void main( )
{
      char name[5];            //define a string of characters
      name[0] = 'D';
      name[1] = 'a';
      name[2] = 'v';
      name[3] = 'e';
      name[4] = 0;             //Null character - end of text
      printf("The name is %s\n",name);
      printf("One letter is %c\n",name[2]);
      printf("Part of the name is %s\n",&name[1]);
}
```

The data declaration for the string appears on line 4. We have used this declaration in previous examples but have not indicated its meaning. The data declaration defines a string called name which has, at most, 5 characters in it. Not only does it define the length of the string, but it also states, through implication, that the characters of the string will be numbered from 0 to 4. In the C language, all subscripts start at 0 and increase by 1 each step up to the maximum which in this case is 4. We have therefore named 5 char type variables: name[0], name[1], name[2], name[3], and name[4]. You must keep in mind that in C the subscripts actually go from 0 to one less than the number defined in the definition statement. This is a property of the original definition of C and the base limit of the string (or array), i.e. that it always starts at zero, cannot be changed or redefined by the programmer.

Using strings

The variable name is therefore a string which can hold up to 5 characters, but since we need room for the NULL terminating character, there are actually only four useful characters. To load something useful into the string, we have 5 statements, each of which assigns one alphabetical character to one of the string characters.

Finally, the last place in the string is filled with the numeral 0 as the end indicator and the string is complete. (A #define statement which sets NULL equal to zero would allow us to use NULL instead of an actual zero, and this would add greatly to the clarity of the program. It would be very obvious that this was a NULL and not simply a zero for some other purpose.) Now that we have the string, we will simply print it out with some other string data in the output statement.

You will, by now, be familiar with the %s is the output definition to output a string and the system will output characters starting with the first one in name until it comes to the NULL character; it will then quit. Notice that in the printf statement only the variable name needs to be given, with no subscript, since we are interested in starting at the beginning. (There is actually another reason that only the variable name is given without brackets. The discussion of that topic will be found in the next section.)

Outputting part of a string

The next printf illustrates that we can output any single character of the string by using the %c and naming the particular character of name we want by including the subscript. The last printf illustrates how we can output part of the string by stating the starting point by using a subscript. The & specifies the address of name[1].

This example may make you feel that strings are rather cumbersome to use since you have to set up each character one at a time. That is an incorrect conclusion because strings are very easy to use as we will see in the next example program.

Some string subroutines

The next example illustrates a few of the more common string handling functions. These functions are found in the C standard library and are defined in the header file *string.h*.

```
#include "stdio.h"
#include "string.h"
void main( )
{
      char name1[12], name2[12], mixed[25];
      char title[20];
      strcpy(name1, "Rosalinda");
      strcpy(name2, "Zeke");
      strcpy(title,"This is the title.");
      printf("    %s\n\n" ,title);
      printf("Name 1 is %s\n", name1);
      printf("Name 2 is %s\n", name2);
      if(strcmp(name1,name2)>0)                 /* returns 1 if name1 > name2 */
      {
            strcpy(mixed,name1);
      }
      else
      {
            strcpy(mixed,name2);
      }
      printf("The biggest name alphabetically is %s\n" ,mixed);
      strcpy(mixed, name1);
      strcat(mixed,"    ");
      strcat(mixed, name2);
      printf("Both names are %s\n", mixed);
}
```

First, four strings are defined. Next, a new function that is commonly found in C programs, the *strcpy* function, or string copy function is used. It copies from one string to another until it comes to the NULL character. Remember that the NULL is actually a 0 and is added to the character string by the system. It is easy to remember which one gets copied to which if you think of the function as an assignment statement. Thus if you were to say, for example, ‘x = 23;’, the data is copied from the right entity to the left one. In the strcpy function the data are also copied from the right entity to the left, so that after execution of the first statement, name1 will contain the string Rosalinda, but without the double quotes. The quotes define a *literal string* and are an indication to the compiler that the programmer is defining a string.

Similarly, Zeke is copied into name2 by the second statement, then the title is copied. Finally, the title and both names are printed out. Note that it is not necessary for the defined string to be exactly the same size as the string it will be called upon to store, only that it is **at least** as long as the string plus one more character for the NULL.

Alphabetical sorting of strings

The next function to be considered is the *strcmp* or the string compare function. It will return a 1 if the first string is lexicographically larger than the second, zero if they are the two strings are identical, and -1 if the first string is lexicographically smaller than the second. A lexicographical comparison uses the ASCII codes of the characters as the basis for comparison. Therefore, ‘A’ will be “smaller” than ‘Z’ because the ASCII code for ‘A’ is 65 whilst the code for ‘Z’ is 90. Sometimes however, strange results occur. A string that begins with the letter ‘a’ is larger than a string that begins with the letter ‘Z’ since the ASCII code for ‘a’ is 97 whilst the code for ‘Z’ is 90.

One of the strings, depending on the result of the compare, is copied into the variable mixed, and therefore the largest name alphabetically is printed out. It should come as no surprise to you that Zeke wins because it is alphabetically larger: length doesn’t matter, only the ASCII code values of the characters.

Combining strings

The last four statements in the [STRINGS.C] example have another new feature, the *strcat*, or string concatenation function. This function simply adds the characters from one string onto the end of another string taking care to adjust the NULL so a single string is produced. In this case, name1 is copied into mixed, then two blanks are concatenated to mixed, and finally name2 is concatenated to the combination. The result is printed out which shows both names stored in the one variable called mixed.

Ques. How is a string stored in memory? Is there any difference between string and charcter array? Write a C program to copy one string to another, using pointers and without using library functions.

Soln: A string variable has to be declared before it is used in any of the functions. Consider the following declaration:

```
char a[80];
```

Using this declaration, the compiler allocates 80 memory locations for the variable **a** ranging from 0 to 79 as shown below:

a	b	c	....	‘\0’
0	1	2	....	80

Since the size of character is 1 byte, each character in the array occupies 1 byte. The array thus declared can hold maximum of 80 characters including NULL character.

Basically a string is an array of characters(character array). However a string should be terminated with the '\0'(NULL) character. There is no such rule for character array.

```
#include<stdio.h>
void my_strcpy(char *dest, char *src)
{
      /*Copy the string */
      while(*src != '\0')
      *dest++ = *src++; /* Attach null character at end */
      *dest = '\0';
}
void main()
{
      char src[20] , dest[20];
      printf(“Enter some text\n”);
      scanf(“%s”,src);
      my_strcpy(dest,src);
      printf(“The copied string = %s\n”,dest);
}
```

Ques. WAP to reverse a string without using reverse function.

```
#include"stdio.h"
#include"conio.h"
#include"string.h"
void main()
```

```
{
      char str[30];
      int s,i;
      clrscr();
      puts("Enter string:");
      gets(str);
      s=strlen(str);
      for(i=s-1;i>=0;i--)
      {
            printf("%c",str[i]);
      }
      getch();
}
```

Ques. WAP to reverse a string using for loop.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
      char a[10],b[10];             //variable declaration
      int i,x,y;
      clrscr();
      printf("Enter a string to reverse : ");
      scanf("%s",a);
      x=(strlen(a)-1);
      for(i=0;i<=x;i++)             //loop – reversing str
      {
            b[i]=a[x-i];
      }
      b[x+1]='\0';
      y=strlen(b);                        //string lengh func.
      printf("\n\nOriginal String : %s \nReversed string : %s",a,b);
      printf("\n\nOriginal String length : %d\nReversed String Length :
%d",x,y);
      printf("\n\n\n\t\t\tPress any key to QUIT...");
      getch();
}
```

Ques. WAP to implement convert a string in lowercase character without using strlowr function

```
#include<stdio.h>
#include<conio.h>
void main()
{
      int s,i,str2[30];
      char str1[30];
      clrscr();
      puts("Enter string:");
      gets(str1);
      s=strlen(str1);
      for(i=0;i<=s-1;i++)
      {
             str2[i]=str1[i];
```

```
	}
	printf("\nThe lowercase of string is:\n\n\n");
	if(str2[0]<98)
	{
		for(i=0;i<=s-1;i++)
		{
			if(str2[i]==32||str2[i]==46)
			{
				printf("%c",str2[i]);
			}
			else
			{
				str2[i]=str2[i]+32;
				printf("%c",str2[i]);
			}
		}
	}
	else
	{
		printf("Please enter uppercase!");
	}
	getch();
}
```

Ques. **WAP to con-centate 2 strings of diff. length**

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
	char st1[15];			//string declaration 1
	char st2[15];			//string declaration 2
	clrscr();
	printf("Enter String 1 : ");
	gets(st1);
	printf("Enter String 2 : ");
	gets(st2);
	strcat(st1,st2);		//string con-centate
	printf("First string after appending second is = %s", st1);
	printf("\n\n\n\t\t\tPress any key to QUIT...");
	getch();
}
```

Ques. **WAP To check while string is palindrome or not.**

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
	char x[20];			//string declaration
	inti,j,k=0;			//variable declaration
	clrscr();
	printf("Enter word : ");
	gets(x);
	i=strlen(x);                        //string length func.
	j=i/2;
	while(k<j)              //loop – palindrome check
	{
		if(x[k]!=x[i-1-k])
		{
			break;
		}
		k++;
	}
	if(k==j)                            //case – Yes
	{
		printf("\nWord is Palindrome.");
	}
	Else                                //default case – No
	{
		printf("\nWord is not Palindrome.");
	}
	printf("\n\n\n\t\t\tPress any key to QUIT...");
	getch();
}
```

<ctype.h>

The header <ctype.h> declares several functions useful for testing, mapping and converting characters. In almost cases the argument is an int, the value of which shall be represent-able as an unsigned char or shall be equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

The behavior of these functions is effected by the current locale. Those functions that have implementation-defined aspects only when not in the "C" locale are noted below.

The term *printing character* refers to a member of an implementation-defined set of characters, each of which occupies one printing position on a display device: the term *control character* refers to a member of an implementation-defined set of characters that are not printing characters.

Implementation Notes

- All the character testing functions are defined as macros that test one or more bits in a constant array indexed by the given character, which must be in the range -1 to 255.
- On the ERC32, the size of the array is 257 bytes.
- On the M1750, the size of the array is 257, 16-bit words, (a total of 514 bytes).

Character Testing Functions

1. isalnum() - The *isalnum* function tests for any character for whichisalphaorisdigitistrue.

Synopsis

```
#include <ctype.h>
int isalnum (int c );
```

2. isalpha() - The *isalpha* function tests for any character for whichisupperorisloweris true, or any character that is one of an implementation-defined set of characters for which non of iscntrl, isdigit, ispunct, orisspace is true. In the "C" locale, isalpha returns true for only the characters for which isupper or islower is true.

Synopsis

```
#include <ctype.h>
int isalpha (int c );
```

3. iscntrl() - The *iscntrl* function tests for any control character.

Synopsis

```
#include <ctype.h>
int iscntrl (int c );
```

4. isdigit() - The *isdigit* function tests for any decimal-digit character

Synopsis

```
#include <ctype.h>
int isdigit (int c );
```

5. isgraph() - The *isgraph* function tests for any printing character except space(' ')

Synopsis

```
#include <ctype.h>
int isgraph (int c );
```

6. islower() - The *islower* function tests for any character that is an lowercase letter or is one of an implementation-defined set of characters for which none of iscntrl,isdigit,ispunct or isspace is true. In the "C" locale, islower returns true only for the characters defined as lower case letters

Synopsis

```
#include <ctype.h>
int islower (int c );
```

7. **isprint()** - The *isprint* function tests for any of the printing characters including space (' ').

Synopsis

```
#include <ctype.h>
int isprint (int c );
```

8. **ispunct()** - The *ispunct* function tests for any printing character that is neither a space (' ') nor a character for which isalnum is true.

Synopsis

```
#include <ctype.h>
int ispunct (int c );
```

9. **isspace()** - The *isspace* function tests for any character that is a standard white space character or is one of an implementation-defined set of characters for whichisalnumis false. The standard white space characters are the following: space (' '), form feed ('\f'), new_line ('\n'), carriage return ('\r'), horizontal tab ('\t'), and vertical tab ('\v'). In the "C" locale, isspace returns true only for standard white space characters.

Synopsis

```
#include <ctype.h>
int isspace (int c );
```

10. **isupper()** - The *isupper* function tests for any character that is an uppercase letter or is one of an implementation-defined set of characters for which none ofiscntrl,isdigit,ispunctorisspace is true. In the "C" locale,isupper returns true only for the characters defined as upper case letters

Synopsis

```
#include <ctype.h>
int isupper (int c );
```

11. **isxdigit()** - The *isxdigit* functions tests for any hexadecimal-digit character.

Synopsis

```
#include <ctype.h>
int isxdigit (int c );
```

12. **isascii()** - The *isacii* functions tests for a 7-bit US-ASCII character.

Synopsis

```
#include <ctype.h>
int isascii (int c );
```

13. toupper() - The *toupper* function converts a character to upper case. It can be performed on a character type to a character type.

Synopsis

```
#include <ctype.h>
char a = toupper (char c );
```

14. tolower() - The *tolower* functions converts any upper case character to a lower case character. It can be performed by the same way the toupper() is performed.

Synopsis

```
#include <ctype.h>
char a = tolower (char c );
```

15. toascii() - The *toascii* converts any character to 7-bit US-ASCII character.

Synopsis

```
#include <ctype.h>
int toascii (char c );
```

Ques. WAP to implementing <ctype.h>

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
void main()
{
      char c;
      clrscr();
      printf("Hello\n");
      c=getch();
      printf("%d",('+'));
      getch();
}
```

Ques. WAP to implement <ctype.h> while using alnum() for testing for numerical entry.

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
void main()
{
      int a;
      char c;
      clrscr();
      printf("Enter any Character : ");
      scanf("%c",&c);
      a=isalnum(c);
      if(a==0)
      {
            printf("The Character is Numerical");
      }
      else
      {
            printf("The Character is Not Numerical");
      }
      getch();
}
```

<string.h>

The string header provides many functions useful for manipulating strings (character arrays).

1. strcat() - Appends the string pointed to by *str2* to the end of the string pointed to by *str1*. The terminating null character of *str1* is overwritten. Copying stops once the terminating null character of *str2* is copied. If overlapping occurs, the result is undefined. The argument *str1* is returned.

Declaration: `char *strcat(char *str1, const char *str2);`

2. strncat() - Appends the string pointed to by *str2* to the end of the string pointed to by *str1* up to *n* characters long. The terminating null character of *str1* is overwritten. Copying stops once *n* characters are copied or the terminating null character of *str2* is copied. A terminating null character is always appended to *str1*. If overlapping occurs, the result is undefined. The argument *str1* is returned.

Declaration: `char *strncat(char *str1, const char *str2, size_t n);`

3. strchr() - Searches for the first occurrence of the character *c* (an unsigned char) in the string pointed to by the argument *str*. The terminating null character is considered to be part of the string. Returns a pointer pointing to the first matching character, or null if no match was found.

Declaration: `char *strchr(const char *str, int c);`

4. strcmp() - Compares the string pointed to by *str1* to the string pointed to by *str2*. Returns zero if *str1* and *str2* are equal. Returns less than zero or greater than zero if *str1* is less than or greater than *str2* respectively.

Declaration: `int strcmp(const char *str1, const char *str2);`

5. strncmp() - Compares at most the first *n* bytes of *str1* and *str2*. Stops comparing after the null character. Returns zero if the first *n* bytes (or null terminated length) of *str1* and *str2* are equal. Returns less than zero or greater than zero if *str1* is less than or greater than *str2* respectively.

Declaration: `int strncmp(const char *str1, const char *str2, size_t n);`

6. strcpy() - Copies the string pointed to by *str2* to *str1*. Copies up to and including the null character of *str2*. If *str1* and *str2* overlap the behavior is undefined. Returns the argument *str1*.

Declaration: `char *strcpy(char *str1, const char *str2);`

7. strncpy() - Copies up to *n* characters from the string pointed to by *str2* to *str1*. Copying stops when *n* characters are copied or the terminating null character in *str2* is reached. If the null character is reached, the null characters are continually copied to *str1* until *n* characters have been copied. Returns the argument *str1*.

Declaration: `char *strncpy(char *str1, const char *str2, size_t n);`

8. strerror() - Searches an internal array for the error number *errnum* and returns a pointer to an error message string. Returns a pointer to an error message string.

Declaration: `char *strerror(int errnum);`

9. strlen() - Computes the length of the string *str* up to but not including the terminating null character. Returns the number of characters in the string.

Declaration: `size_t strlen(const char *str);`

10. strpbrk() - Finds the first character in the string *str1* that matches any character specified in *str2*. A pointer to the location of this character is returned. A null pointer is returned if no character in *str2* exists in *str1*.

Declaration: `char *strpbrk(const char *str1, const char *str2);`

Example:

```
#include<string.h>
#include<stdio.h>
void main()
{
      char string[]="Hi there, Chip!";
      char *string_ptr;
      while((string_ptr=strpbrk(string," "))!=NULL)
      *string_ptr='-';
      printf("New string is \"%s\".\n",string);
}
```

The output should result in every space in the string being converted to a dash (-).

11. strspn() - Finds the first sequence of characters in the string *str1* that contains any character specified in *str2*. Returns the length of this first sequence of characters found that match with *str2*.

Declaration: `size_t strspn(const char *str1, const char *str2);`

Example:

```
#include<string.h>
#include<stdio.h>
void main()
{
      char string[]="7803 Elm St.";
      printf("The number length is %d.\n",strspn(string,"1234567890"));
}
```

```
The output should be: The number length is 4.
```

12. strlwr() - Converts a string to lower case.

Declaration: `strlwr(char variable_name[size]);`

13. strupr() - Convers a string to upper case.

Declaration: `strupr(char variable_name[size]);`

14. stricmp() - Compare n characters of string to another.

Declaration: `strnicmp(str1,str2,size);`

15. strrev() - Reverse a string.

Declaration: `strrev(string_name);`

14. STRUCTURES & UNION

C Language provides a facility of creating certain data type. Structure is a custom data type. It can combine different data type and data type according to need of user is created. Structure is also defined as user defined data type. A Structure can contain int, char, float elements. These elements are called as a member of structure.

Structure is also defined as a heterogeneous collection of variables grouped under single logical entity. Structures are called “records” in some languages, notably Pascal. Structures help organize complicated data, particularly in large programs, because they permit a group of related variables to be treated as a unit instead of as separate entities.

Structure is also defined as a collection of different data types under a common name. The best way to understand a structure is to look at an example

```
#include "stdio.h"
void main( )
{
      struct
      {
            char initial;                  //last name initial
            int age;                       //childs age
            int grade;                     //childs grade in school
      }boy,girl;
      boy.initial = 'R';
      boy.age = 15;
      boy.grade = 75;
      girl.age = boy.age - 1;       /* she is one year younger    */
      girl.grade = 82;
      girl.initial = 'H';
      printf("%c is %d years old and got a grade of %d\n",
      girl.initial, girl.age, girl.grade);
      printf("%c is %d years old and got a grade of %d\n",
      boy.initial, boy.age, boy.grade);
}
```

The program begins with a structure definition. The key word struct is followed by some simple variables between the braces, which are the components of the structure. After the closing brace, you will find two variables listed namely boy, and girl. According to the definition of a structure, boy is now a variable composed of three elements: initial, age, and grade. Each of the three fields are associated with boy, and each can store a variable of its respective type. The variable girl is also a variable containing three fields with the same names as those of boy but is actually different variables. We have therefore defined 6 simple variables.

A single compound variable

Let's examine the variable boy more closely. As stated above, each of the three elements of boy are simple variables and can be used anywhere in a C program where a variable of their type can be used. For example,the age element is an integer variable and can therefore be used anywhere in a C program where it is legal to use an integer variable: in calculations, as a counter, in I/O operations, etc. The only problem we have is defining how to use the simple variable age which is a part of the compound variable boy. We use both names with a decimal point between them with the major name first. Thus boy.age is the complete variable name for the age field of boy. This construct can be used anywhere in a C program that it is desired to refer to this field. In fact, it is illegal to use the name boy or age alone because they are only partial definitions of the complete field. Alone, the names refer to nothing.

Assigning values to the variables

Using the above definition, we can assign a value to each of the three fields of boy and each of the three fields of girl. Note carefully that boy.initial is actually a char type variable, because it was assigned that in the structure, so it must be assigned a character of data. Notice that boy.initial is assigned the character R in agreement with the aboverules. The remaining two fields of boy are assigned values in accordance with their respective types. Finally the three fields of girl are assigned values but in a different order to illustrate that the order of assignmentis not critical.

Using structure data

Now that we have assigned values to the six simple variables, we can do anything we desire with them. In order to keep this first example simple, we will simply print out the values to see if they really do exist as assigned. If you carefully inspect the printf statements, you will see that there is nothing special about them. The compound name of each variable is specified because that is the only valid name by which we can refer to these variables.

Structures are a very useful method of grouping data together in order to make a program easier to write and understand. This first example is too simple to give you even a hint of the value of using structures, but continue on through these lessons and eventually you will see the value of using structures.

Ques. WAP to implement define structure type struct employee that would contain employee name,age,d.o.b,salary,to read this information for three employee from the keyboard and print the same on screen.

```
#include"stdio.h"
#include"conio.h"
struct employee
{
	char emp_name[10];
	int emp_age;
```

```
      int date;
      int month;
      int year;
      long int salary;
};
void main()
{
      struct employee e1,e2,e3;
      clrscr();
            //value assigning area start
      printf("\n1:Emp_name:");      //first employee record
      scanf("%s",&e1.emp_name);
      printf("\nAge:");
      scanf("%d",&e1.emp_age);
      printf("\nD.O.B:");
      printf("\ndate:");
      scanf("%d",&e1.date);
      printf("\nmonth:");
      scanf("%d",&e1.month);
      printf("\nyear:");
      scanf("%d",&e1.year);
      printf("\nSalary:");
      scanf("%ld",&e1.salary);
      printf("\n2:Emp_name:");      //second employee record
      scanf("%s",&e2.emp_name);
      printf("\nAge:");
      scanf("%d",&e2.emp_age);
      printf("\nD.O.B:");
      printf("\ndate:");
      scanf("%d",&e2.date);
      printf("\nmonth:");
      scanf("%d",&e2.month);
      printf("\nyear:");
      scanf("%d",&e2.year);
      printf("\nSalary:");
      scanf("%ld",&e2.salary);
      printf("\n3:Emp_name:");      //third employee record
      scanf("%s",&e3.emp_name);
      printf("\nAge:");
      scanf("%d",&e3.emp_age);
      printf("\nD.O.B:");
      printf("\ndate:");
      scanf("%d",&e3.date);
      printf("\nmonth:");
      scanf("%d",&e3.month);
      printf("\nyear:");
      scanf("%d",&e3.year);
      printf("\nSalary:");
      scanf("%ld",&e3.salary);      //value assigning area close
      clrscr();                     //printing area start
      printf("\n\t\t\tThe Employee Record");
      printf("\n\nEmp_name\tAge\tD.O.B\t\tsalary");
      printf("\n1:");          //first employee
      printf("%s",e1.emp_name);
      printf("\t\t%d",e1.emp_age);
      printf("\t%d %d %d",e1.date,e1.month,e1.year);
      printf("\t%ld",e1.salary);
      printf("\n2:");         //second employee
      printf("%s",e2.emp_name);
      printf("\t\t%d",e2.emp_age);
      printf("\t%d %d %d",e2.date,e2.month,e2.year);
      printf("\t%ld",e2.salary);
      printf("\n3:");         //third employee
      printf("%s",e3.emp_name);
      printf("\t\t%d",e3.emp_age);
      printf("\t%d %d %d",e3.date,e3.month,e3.year);
      printf("\t%ld",e3.salary);
      getch();
}
```

An array of structures

The next program contains the same structure definition as before but this time we define an array of 12 variables named kids. This program therefore contains 12 times 3 = 36 simple variables, each of which can store one item of data provided that it is of the correct type. We also define a simple variable named index for use in the for loops.

```
#include "stdio.h"
void main( )
{
      struct {
      char initial;
      int age;
      int grade;
      }kids[12];
      int index;
      for (index = 0; index < 12; index++)
      {
            kids[index].initial = 'A' + index;
            kids[index].age = 16;
            kids[index].grade = 84;
      }
      kids[3].age = kids[5].age = 17;
      kids[2].grade = kids[6].grade = 92;
      kids[4].grade = 57;
      kids[10] = kids[4];                       //Structure assignment
      for (index = 0; index < 12; index++)
      {
            printf("%c is %d years old and got a grade
of%d\n",kids[index].initial,kids[index].age,kids[index].grade);
      }
      getch();
}
```

To assign each of the fields a value we use a for loop and each pass through the loop results in assigning a value to three of the fields. One pass through the loop assigns all of the values for one of the kids. This would not be a very useful way to assign data in a real situation, but a loop could read the data in from a file and store it in the correct fields. You might consider this the crude beginning of a data base – which, of course, it is.

In the next few instructions of the program we assign new values to some of the fields to illustrate the method used to accomplish this. It should be self explanatory, so no additional comments will be given.

Ques. WAP to maintain record of file customer in a bank using array of structure

```
#include<stdio.h>
#include<conio.h>
struct bank
{
      int bal;
      int acc_no;
};
void main()
{
      struct bank b[10],an[10];
      int n,i,flag,ac;
      clrscr();
      printf("\nHow many record u want enter:");      //enter through customer
      scanf("%d",&n);
      for(i=0;i<n;i++)
      {
            printf("\nEnter   %d acount no. balance:",8201+i);
            scanf("%d",&b[i].bal);
            an[i].acc_no=8201+i;
      }
      clrscr();
      a:
      printf("\nWhat account u check balance:");
      scanf("%d",&ac);
      for(i=0;i<n;i++)  //check through bank accountant
      {
            if(ac==an[i].acc_no)
            {
                  flag=0;
                  break;
            }
            else
            {
                  flag=-1;
            }
      }
      if(flag==0)
      {
            printf("\nThe balance is:%d",b[i].bal);
      }
      else
      {
            printf("\nAccount no. is not found!");
            goto a;
      }
      getch();
}
```

Copying structures

C allows you to copy an entire structure with one statement. Line 17 is an example of using a structure assignment. In this statement, all 3 fields of kids[4] are copied into their respective fields of kids[10].

The last few statements contain a for loop in which all of the generated values are displayed in a formatted list. Compile and run the program to see if it does what you expect it to do.

Ques. WAP to implement copy and compare to structure variable

```
#include"stdio.h"
#include"conio.h"
#include"string.h"
void main()
{
      struct student
      {
            char name[20];
            int rollno;
            int marks;
      }s1,s2;
      int a;
      clrscr();
      printf("\nEnter name:");
      scanf("%s",&s1.name);
      printf("\nEnter rollno:");
      scanf("%d",&s1.rollno);
      printf("\nEnter marks:");
      scanf("%d",&s1.marks);
      s2=s1;              //copy structure in another struct variable
      printf("\nName:%s",s2.name);
      printf("\nRollno:%d",s2.rollno);
      printf("\nMarks:%d",s2.marks);
      if(s2.rollno==s1.rollno&&s2.marks==s1.marks)
      {
            printf("\n\tBoth are equal");
      }
      else
      {
            printf("\n\tBoth are not equal!");
      }
      a=strcmp(s2.name,s1.name);
      if(a==0)
      {
            printf("\n\tBoth str are also equal");
      }
      else
      {
                  printf("\n\tBoth str are not equal");
      }
      getch();
}
```

Using Pointers and Structures together

The next program is an example of using pointers with structures; it is identical to the last program except that it uses pointers for some of the operations.

```
#include "stdio.h"
void main( )
{
      struct
      {
            char initial;
            int age;
            int grade;
      } kids[12], *point, extra;
      int index;
      for (index = 0; index < 12; index++)
      {
            point = kids + index;
            point->initial = 'A' + index;
            point->age = 16;
            point->grade = 84;
      }
      kids[3].age = kids[5].age = 17;
      kids[2].grade = kids[6].grade = 92;
      kids[4].grade = 57;
      for (index = 0; index < 12; index++)
      {
            point = kids + index;
            printf("%c is %d years old and got a grade of %d\n",(*point).initial,
kids[index].age, point->grade);
      }
      extra = kids[2];  /* Structure assignment */
      extra = *point;   /* Structure assignment */
}
```

The first difference shows up in the definition of variables following the structure definition. In this program we define a pointer named point which is defined as a pointer that points to the structure. It would be illegal to try to use this pointer to point to any other variable type.

There is a very definite reason for this restriction in C as we have alluded to earlier and will review in the next few paragraphs.

The next difference is in the for loop where we use the pointer for accessing the data fields. Since kids is a pointer variable that points to the structure, we can define point in terms of kids. The variable kids is a constant so it cannot be changed in value, but point is a pointer variable and can be assigned any value consistent with its being required to point to the structure. If we assign the value of kids to point then it should be clear that it will point to the first element of the array, a structure containing three fields.

Ques. **WAP to implement pointer and structure together to store a student info.**

```
#include"stdio.h"
#include"conio.h"
void main()
{
      struct student
      {
            int rollno;
            char name[10];
            int fees;
      }s,*p;
      p=&s;
      clrscr();
      printf("enter student rollno,name,fees");
      scanf("%d %s %d",&p->rollno,&p->name,&p->fees);
      printf("You have enter following detail");
      printf("Roll no = %d\nName = %s\nFees = %d",p->rollno,p->name,p->fees);
      getch();
}
```

Pointer arithmetic

Adding 1 to point will now cause it to point to the second field of the array because of the way pointers are handled in C. The system knows that the structure contains three variables and it knows how many memory elements are required to store the complete structure. Therefore if we tell it to add one to the pointer, it will actually add the number of memory elements required to get to the next element of the array. If, for example, we were to add 4 to the pointer, it would advance the value of the pointer 4 times the size of the structure, resulting in it pointing 4 elements farther along the array. This is thereason a pointer cannot be used to point to any data type other than the one for which it was defined.

Now return to the program. It should be clear from the previous discussion that as we go through the loop, the pointer will point to the beginning of one of the array elements each time. We can therefore usethe pointer to reference the various elements of the structure. Referring to the elements of a structure with a pointer occurs so often in C that a special method of doing that was devised. Using point->initial is the same as using (*point).initial which is really the way we did it in thelast two programs. Remember that *point is the stored data to which the pointer points and the construct should be clear. The -> is made up of the minus sign and the greater than sign.

Since the pointer points to the structure, we must once again define which of the elements we wish to refer to each time we use one of the elements of the structure. There are, as we have seen, several different methods of referring to the members of the structure, and in the for loop used for output at the end of the program, we use three different methods. This would be considered very poor programming practice, but is done this way here to illustrate to you that they all lead to the same result. This program will probably require some study on your part to fully understand, but it will be worth your time and effort to grasp these principles.

Nested and named structures

The structures we have seen so far have been useful, but very simple. It is possible to define structures containing dozens and even hundreds or thousands of elements but it would be to the programmer’s advantage not to define all of the elements at one pass but rather to use a hierarchical structure of definition. This will be illustrated with the next program which shows a nested structure.

```
#include "stdio.h"
void main( )
{
      struct person
      {
            char name[25];
            int age;
            char status; /* M = married, S = single */
      };
      struct alldat
      {
            int grade;
            struct person descrip;
            char lunch[25];
      }student[53];
      struct alldat teacher,sub;
      teacher.grade = 94;
      teacher.descrip.age = 34;
      teacher.descrip.status = 'M';
      strcpy(teacher.descrip.name,"Mary Smith");
      strcpy(teacher.lunch,"Baloney sandwich");
      sub.descrip.age = 87;
      sub.descrip.status = 'M';
      strcpy(sub.descrip.name,"Old Lady Brown");
      sub.grade = 73;
      strcpy(sub.lunch,"Yogurt and toast");
      student[1].descrip.age = 15;
      student[1].descrip.status = 'S';
      strcpy(student[1].descrip.name,"Billy Boston");
      strcpy(student[1].lunch,"Peanut Butter");
      student[1].grade = 77;
      student[7].descrip.age = 14;
      student[12].grade = 87;
}
```

The first structure contains three elements but is followed by no variable name. We therefore have not defined any variables only a structure, but since we have included a name at the beginning of the structure, the structure is named person. The name person can be used to refer to the structure but not to any variable of this structure type. It is therefore a new type that we have defined, and we can use the new type in nearly the same way we use int, char, or any other types that exist in C. The only restriction is that this new name must always be associated with the reserved word struct.

The next structure definition contains three fields with the middle field being the previously defined structure which we named person. The variable which has the type of person is named descrip. So the new structure contains two simple variables, grade and a string named lunch[25], and the structure named descrip. Since descrip contains three variables, the new structure actually contains 5 variables. This structure is also given a name, alldat, which is another type definition. Finally we define an array of 53 variables each with the structure defined by alldat, and each with the name student. If that is clear, you will see that we have defined a total of 53 times 5 variables, each of which is capable of storing a value.

Since we have a new type definition we can use it to define two more variables. The variables teacher and sub are defined in line 13 to be variables of the type alldat, so that each of these two variables contain5 fields which can store data.

Ques. WAP to maintain record of student (name,d.o.b,address) using nesting structure.

```
#include"stdio.h"
#include"conio.h"
void main()
{
      struct d_o_b
      {
            int date;
            int month;
            int year;
      };
      struct address
      {
            char city[10];
            char state[10];
      };
      struct student
      {
            char name[10];
            struct d_o_b d;
            struct address a;
      }s[10];
      int i,n;
      clrscr();
      printf("\nEnter number of student:");
      scanf("%d",&n);
      for(i=0;i<n;i++)
      {
            printf("\n%d student name:",i+1);
            scanf("%s",&s[i].name);
            printf("\n%d student d.o.b:",i+1);
            printf("\nDate:");
            scanf("%d",&s[i].d.date);
            printf("\nMonth:");
            scanf("%d",&s[i].d.month);
            printf("\nYear:");
            scanf("%d",&s[i].d.year);
            printf("\n%d student address:",i+1);
            printf("\nCity:");
            scanf("%s",&s[i].a.city);
            printf("\nState:");
            scanf("%s",s[i].a.state);
      }
      clrscr();
      printf("\n\t\t\tThe Record Of Student");
      printf("\n\nName\td_o_b\t\tCity\tState");
      for(i=0;i<n;i++)
      {
            printf("\n%d:%s",i+1,s[i].name);
            printf("\t%d",s[i].d.date);
            printf(" %d",s[i].d.month);
            printf(" %d",s[i].d.year);
            printf("\t%s",s[i].a.city);
            printf("\t%s",s[i].a.state);
      }
      getch();
}
```

Using fields

In the next five lines of the program we assign values to each of the fields of teacher. The first field is the grade field and is handled just like the other structures we have studied because it is not part of the nested structure. Next we wish to assign a value to her age which is part of the nested structure. To address this field we start with the variable name teacher to which we append the name of the group descrip, and thenwe must define which field of the nested structure we are interested in, so we append the name age. The teacher's status field is handled in exactly the same manner as her age, but the last two fields are assigned strings using the string copy function strcpy, which must be used for string assignment. Notice that the variable names in the strcpy function are still variable names even though they are each made up of several parts.

The variable sub is assigned nonsense values in much the same way, but in a different order since they do not have to occur in any requiredorder. Finally, a few of the student variables are assigned values for illustrative purposes and the program ends. None of the values are printed for illustration since several were printed in the last examples.

It is possible to continue nesting structures until you get totally confused. If you define them properly, the computer will not get confused because there is no stated limit as to how many levels of nesting are allowed. There is probably a practical limit of three beyond which you will get confused, but the language has no limit. In addition to nesting, you can include as many structures as you desire in any level of structures, such as defining another structure prior to alldat and using it in alldat in addition to using person. The structure named person could be included in alldat two or more times if desired.

Structures can contain arrays of other structures which in turn can contain arrays of simple types or other structures. It can go on and on until you lose all reason to continue. Be conservative at first, and get bolder as you gain experience.

Ques. WAP to enter the data of student using array of structure & nesting structure.

```
#include<stdio.h>
#include<conio.h>
struct dob                 //structure – DOB
{
     int date;
     int month;
     int year;
};
struct student                  //structure – student
{
     int roll;
     char name[20];
     struct dob birth;          //nesting structure
     int marks;
     char phn[11];
};
void main()
{
     struct student s[10]; //structure declaration
     int i,n;                   //variable declaration
     start:                     //label : start
     clrscr();
     printf("\t\t\t\tStudent Info\n\n");
     printf("Enter the no. of students : ");
     scanf("%d",&n);
     if(n>10)                   //case – Array check
     {
            printf("Out of Range... Enter Valid Input");
            getch();
            goto start;
      }
      for(i=0;i<n;i++)      //loop – struct array scan
      {
            clrscr();
            printf("\t\t\t\tStudent Info\n\n");
            printf("Student %d\n\n",i+1);
            printf("Roll no. \t\t: ");
            scanf("%d",&s[i].roll);
            printf("Name \t\t\t: ");
            scanf("%s",&s[i].name);
            printf("DOB(DDMMYYYY) \t\t: ");
      scanf("%2d""%2d""%4d",&s[i].birth.date,&s[i].birth.month,&s[i].birth.
year);

            printf("Marks \t\t\t: ");
            scanf("%d",&s[i].marks);
            printf("Phn No. \t\t: ");
            scanf("%s",&s[i].phn);
            if(i==n-1)
            {
                  printf("\n\n\t\tYour Data is stored. Press any key to
view.");
                  getch();
                  clrscr();
            }
      }
      printf("\t\t\t\tStudent Info\n\n");
      printf("Roll No.\tName\t\tDOB\t\tMarks\t\tPh. No.\n");
      for(i=0;i<n;i++)      //loop – struct array print
      {
            printf("%d\t\t%s\t\t%d-%d-
%d\t%d\t\t%s",s[i].roll,s[i].name,s[i].birth.date,s[i].birth.month,s[i].bi
rth.year,s[i].marks,s[i].phn);
            printf("\n");
      }
      printf("\n\n\n\t\tPress any key to QUIT");
      getch();
}
```

Unions

A union is a variable that may hold (at different times) data of different types and sizes. For example, a programmer may wish to define a structure that will record the citation details about a book, and another that records details about a journal. Since a library item can be neither a book nor a journal simultaneously, the programmer would declare a library item to be a union of book and journal structures. Thus, on one occasion item might be used to manipulate book details, and on another occasion, item might be used to manipulate journal details. It is up to the programmer, of course, to remember the type of item with which they are dealing.

Examine the next program for examples.

```
void main( )
{
      union
      {
            int value;  // This is the first part of the union
            struct
            {
                  char first;       /* These two values are the second part */
                  char second;
            }half;
      }number;
      long index;
      for (index = 12; index < 300000; index += 35231)
      {
            number.value = index;
            printf("%8x %6x %6x\n", number.value,
umber.half.first,number.half.second);
      }
}
```

In this example we have two elements to the union, the first part being the integer named value, which is stored as a two byte variable somewhere in the computer's memory. The second element is made up of two character variables named first and second. These two variables are stored in the same storage locations that value is stored in, because that is what a union does. A union allows you to store different types of data in the same physical storage locations. In this case, you could put an integer number in value, then retrieve it in its two halves by getting each half using the two names first and second. This technique is often used to pack data bytes together when you are, for example, combining bytes to be used in the registers of the microprocessor.

Accessing the fields of the union is very similar to accessing the fields of a structure and will be left to you to determine by studying the example. One additional note must be given here about the program. When it is run using the C compiler the data will be displayed with two leading f's,due to the hexadecimal output promoting the char type variables to int and extending the sign bit to the left. Converting the char type data fields to int type fields prior to display should remove the leading f's from your display. This will involve defining two new int type variables and assigning the char type variables to them. This will be left as an exercise for you. Note that the same problem will also come up in a few of the later examples.

Compile and run this program and observe that the data is read out as an int and as two char variables. The char variables are reversed in order because of the way an int variable is stored internally in your computer. Don't worry about this. It is not a problem but it can be a very interesting area of study if you are so inclined.

The next program contains another example of a union, one which is much more common. Suppose you wished to build a large database including information on many types of vehicles. It would be silly to include the number of propellers on a car, or the number of tires on a boat. In order to keep all pertinent data, however, you would need those data points for their proper types of vehicles. In order to build an efficient data base, you would need several different types of data for each vehicle, some of which would be common, and some of which would be different. That is exactly what we are doing in the example program, in which we define a complete structure, and then decide which of the various types can go into it.

```
#include "stdio.h"
#define AUTO 1
#define BOAT 2
#define PLANE 3
#define SHIP 4
void main( )
{
      struct automobile       /* structure for an automobile      */
      {
            int tires;
            int fenders;
            int doors;
      };
      typedef struct
      {                 /* structure for a boat or ship */
            int displacement;
            char length;
      } BOATDEF;
      struct
      {
            char vehicle;     /* what type of vehicle?      */
            int weight;       /* gross weight of vehicle    */
            union
            {           /* type-dependent data  */
                  struct automobile car;  /* part 1 of the union  */
                  BOATDEF boat;     /* part 2 of the union  */
                  struct
                  {
                        char engines;
                        int wingspan;
                  } airplane;       /* part 3 of the union  */
                  BOATDEF ship;     /* part 4 of the union  */
            } vehicle_type;
            int value;  /* value of vehicle in dollars      */
            char owner[32];   /* owners name    */
      } ford, sun_fish, piper_cub;  /* three variable structures  */
      /* define a few of the fields as an illustration      */
      ford.vehicle = AUTO;
      ford.weight = 2742;     /* with a full petrol tank    */
      ford.vehicle_type.car.tires = 5;    /* including the spare  */
      ford.vehicle_type.car.doors = 2;
      sun_fish.value = 3742;  /* trailer not included       */
      sun_fish.vehicle_type.boat.length = 20;
      piper_cub.vehicle = PLANE;
      piper_cub.vehicle_type.airplane.wingspan = 27;
      if (ford.vehicle == AUTO)     /* which it is in this case */
      printf("The ford has %d tires.\n", ford.vehicle_type.car.tires);
      if (piper_cub.vehicle == AUTO)      /* which it is not      */
      printf("The plane has %d tires.\n",
      piper_cub.vehicle_type.car.tires);
}
```

First, we define a few constants with the #defines, and begin the program itself. We define a structure named automobile containing several fields which you should have no trouble recognizing, but we define no variables at this time.

Ques. WAP to accessing of union member and use of size of operator.

```
#include"stdio.h"
#include"conio.h"
#include"string.h"
void main()
{
      union emp
      {
            char name[10];
            int age;
            int salary;
      }e;
      clrscr();
      printf("\nSize of Union : %d",sizeof(e));
      printf("\nEnter name:");
      scanf("%s",&e.name);
      printf("\nName:%s",e.name);
      printf("\nEnter Age:");
      scanf("%d",&e.age);
      printf("\nAge:%d",e.age);
      printf("\nEnter Salary:");
      scanf("%d",&e.salary);
      printf("\nSalary:%d",e.salary);
      printf("\nSize of Union : %d",sizeof(e));
      //    printf("\nSize of age:%d",sizeof(e.age));
//    printf("\nSize of salary:%d",sizeof(e.salary));
```

```
      //    printf("\nSize of name:%d",sizeof(e.name));
      getch();
}
```

The typedef

Next we define a new type of data with a typedef. This defines a complete new type that can be used in the same way that int or char can be used. Notice that the structure has no name, but at the end where there would normally be a variable name there is the name BOATDEF. We now have a new type, BOATDEF, that can be used to define a structure anywhere we would like to. Notice that this does not define any variables, only a new type definition. Capitalising the name is a common preference used by programmers and is not a C standard; it makes the typedef look different from a variable name.

We finally come to the big structure that defines our data using the building blocks already defined above. The structure is composed of 5 parts, two simple variables named vehicle and weight, followed by the union, and finally the last two simple variables named value and owner. Of course the union is what we need to look at carefully here, so focus on it for the moment. You will notice that it is composed of four parts, the first part being the variable car which is a structure that we defined previously. The second part is a variable named boat which is a structure of the type BOATDEF previously defined. The third part ofthe union is the variable airplane which is a structure defined in place in the union. Finally we come to the last part of the union, the variable named ship which is another structure of the type BOATDEF.

It should be stated that all four could have been defined in any of the three ways shown, but the three different methods were used to show you that any could be used. In practice, the clearest definition would probably have occurred by using the typedef for each of the parts.

We now have a structure that can be used to store any of four different kinds of data structures. The size of every record will be the size of that record containing the largest union. In this case part 1 is the largest union because it is composed of three integers, the others being composed of an integer and a character each. The first member of this union would therefore determine the size of all structures of this type. The resulting structure can be used to store any of the four types of data, but it is up to the programmer to keep track of what is stored in each variable of this type. The variable vehicle was designed into this structure to keep track of the type of vehicle stored here. The four defines at the top of the page were designed to be used as indicators to be stored in the variable vehicle. A few examples of how to use the resulting structure are given in the next few lines of the program. Some of the variables are defined and a few of them are printed out for illustrative purposes.

The union is not used frequently, and almost never by novice programmers. You will encounter it occasionally so it is worth your effort to at least know what it is.There is one more technique, which in some situations can help to clarify the source code of C program. This technique is to made use of typedef declaration. Its purpose is to define the name of existing variable type. For example,

```
typedef unsigned long int Pankaj;

Pankaj P1, P2;
```

Thus, typedef provide a short and meaningful way to call a data type.

Ques. **WAP to renaming the structure data type using typedef.**

```
#include"stdio.h"
#include"conio.h"
struct student
{
      char name[10];
      int rollno;
      int marks;
};
void main()
{
      typedef struct student rahul;
      rahul s;
      rahul *ptr;
      ptr=&s;
      clrscr();
      printf("\nEnter student name:");
      scanf("%s",&ptr->name);
      printf("\nEnter student rollno:");
      scanf("%d",&ptr->rollno);
      printf("\nEnter student marks:");
      scanf("%d",&ptr->marks);
      printf("\nName\tRollno\tMarks");
      printf("\n%s",ptr->name);
      printf("\t%d",ptr->rollno);
      printf("\t%d",ptr->marks);
      getch();
}
```

Bitfields

The bitfield is a relatively new addition to the C programming language. In the next program we have a union made up of a single int type variable in line 5 and the structure defined in lines 6 to 10. The structure is composed of three bitfields named x, y, and z. The variable named x is only one bit wide, the variable y is two bits wide and adjacent to the variable x, and the variable z is two bits wide and adjacent to y. Moreover, because the union causes the bits to be stored in the same memory location as the variable index, the variable x is the least significant bit of the variable index, y forms the next two bits, and z forms the two high-order bits.

```
#include "stdio.h"
void main( )
{
      union
      {
            int index;
            struct
            {
                  unsigned int x : 1;
                  unsigned int y : 2;
                  unsigned int z : 2;
            }bits;
      }number;
      for (number.index = 0; number.index < 20; number.index++)
      {
            printf("index = %3d, bits = %3d%3d%3d\n", number.index, number.bits.z,
number.bits.y, number.bits.x);
      }
}
```

Compile and run the program and you will see that as the variable indexis incremented by 1 each time through the loop, you will see the bitfields of the union counting due to their respective locations within the int definition. One thing must be pointed out: the bitfields must be defined as parts of an unsigned int or your compiler will issue an error message.

The bitfield is very useful if you have a lot of data to separate into separate bits or groups of bits. Many systems use some sort of a packed format to get lots of data stored in a few bytes. Your imagination is your only limitation to effective use of this feature of C.

Ques. WAP to implement bitfield in a record of employe.

```
#include"stdio.h"
#include"conio.h"
struct emp
{
      char name[10];
      unsigned age : 7;
      unsigned gender : 1;
};
union emp2
{
      int age2;
      int gender2;
      unsigned salary;
};
void main()
{
      struct emp e1;
      union emp2 e2;
      clrscr();
      printf("\nEnter name:");
```

```
	scanf("%s",&e1.name);
	printf("\nEnter age:");
	scanf("%d",&e2.age2);
	e1.age=e2.age2;
	printf("\nEnter Gender 0 for Male & 1 for Female:");
	scanf("%d",&e2.gender2);
	e1.gender=e2.gender2;
	printf("\nEnter salary:");
	scanf("%u",&e2.salary);
	printf("\nThe total size of structure is %d",sizeof(e1)+sizeof(e2));
	printf("\nName:%s",e1.name);
	printf("\nAge:%u",e1.age);
	printf("\nGender:%u",e1.gender);
	printf("\nSalary:%u",e2.salary);
	getch();
}
```

Enumerated Data Type(enum)

Enumerated data type give enum opportunity to invent your own data type and define what values the variable of data type can take. This can help in making the program listing more readable which can be an advantage when a program gets complicated or more than one programmer would be working on it. A simple example of enum data type is :

```
enum month { jan = 1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };

enum month this_month;

this_month = feb;
```

In the above declaration, month is declared as an enumerated data type. It consists of a set of values, jan to dec. Numerically, jan is given the value 1, feb the value 2, and so on. The variable this_month is declared to be of the same type as month, and then is assigned the value associated with feb. This_month cannot be assigned any values outside those specified in the initialization list for the declaration of month.

```
#include <stdio.h>
void main()
{
	char *pwest = "west",*pnorth = "north", *peast="east", *psouth = "south";
	enum location { east=1, west=2, south=3, north=4};
	enum location direction;
	direction = east;
	if( direction == east )
	printf("Cannot go %s\n", peast);
}
```

The variables defined in the enumerated variable location should be assigned initial values.

Ques. WAP to maintain the record of employee in a payroll system using enumerated data type department can have four possible values assembly, manufacturing,accounts,store.

```
#include"stdio.h"
#include"conio.h"
enum department
{
      assembly,manufacturing,accounts,store
};
struct employee
{
      char name[10];
      int age;
      int salary;
      enum department d;
};
void main()
{
      struct employee e;
      clrscr();
      printf("\nEnter employee name:");
      scanf("%s",&e.name);
      printf("\nEnter employee age:");
      scanf("%d",&e.age);
      printf("\nEnter employee salary:");
      scanf("%d",&e.salary);
      printf("\nEnter employee Department:");
      scanf("%d",&e.d);
       //   e.d=manufacturing;
      printf("\nemployee name:%s",e.name);
      printf("\nemployee age:%d",e.age);
      printf("\nemployee salary:%d",e.salary);
      printf("\nemployee department:%d",e.d);
      getch();
}
```

15. PREPROCESSOR & STANDARD LIBRARY

The preprocessing step happens to the C source before it is fed to the compiler. The two most common preprocessor directives are #define and #include... There are four types of pre-processor:

1. Macro expression
2. File Inclusion
3. Conditional Compilation
4. Miscellaneous Directives

1. Macro Expression (#define)

The #define directive can be used to set up symbolic replacements in the source. As with all preprocessor operations, #define is extremely unintelligent -- it just does textual replacement without understanding. #define statements are used as a crude way of establishing symbolic constants.

```
#define MAX 100
#define SEVEN_WORDS that_symbol_expands_to_all_these_words
```

Later code can use the symbols MAX or SEVEN_WORDS which will be replaced by the text to the right of each symbol in its #define.Example:

```
#include<stdio.h>
#define PI 3.145
void main( )
{
    float r = 6.25;
    float area;
    area = PI*r*r;
    printf("\n Area of Circle   = %f ", area);
}
```

UPPER and PI in the above programs are often called 'marco templates', whereas, 25 and 3.1415 are called their corresponding 'marco expansions'.When we compile the program, before the source code passes to the compiler, it is examined by the C preprocessor for any macro definitions.

When it sees the #define directive, it goes through the entire program in search of the marco templates.

*Note:*A marco template and its marco expansion are separated by blanks or tabs. A space between # and define is optional. Remember that a marco definition is never tobe terminated by a semicolon.

Thus, using #define can produce more efficient and more easily understand programs. This directive is used extensively by C Programmers. Following three examples show places where a #define directive is popularly used by C Programmers.

2. File Inclusion(#include)

The "#include" directive brings in text from different files during compilation. #include is a very unintelligent and unstructured -- it just pastes in the text from the given file and continues compiling. The #include directive is used in the .h/.c file convention below which is used to satisfy the various constraints necessary to get prototypes correct.

```
#include "stdio.h"      // refers to a "user" stdio.h file -// in the originating directory for the compile
#include <stdio.h>      // refers to a "system" stdio.h file --// in the compiler's directory somewhere
```

3. Conditional Compilation #if

At compile time, there is some space of names defined by the #defines. The #if test can be used at compile-time to look at those symbols and turn on and off which lines the compiler uses. The following example depends on the value of the FOO #define symbol. If it is true, then the "aaa" lines (whatever they are) are compiled, and the "bbb" lines are ignored. If FOO were 0, then the reverse would be true.

```
#define PANKAJ 1
...
#if PANKAJ
aaa
aaa
#else
bbb
bbb
#endif
```

You can use #if 0 ...#endifto effectively comment out areas of code you don't want to compile, but which you want to keep in the source file.

#if directive

- #if is followed by a integer constant expression.
- If the expression is not zero, the statement(s) following the #if are compiled, otherwise they are ignored.
- #if statements are bounded by a matching #endif, #else or #elif
- Macros, if any, are expanded, and any undefined tokens are replaced with 0 before the constant expression is evaluated.
- Relational operators and integer operators may be used.

Expression Examples

```
#if 1
#if 0
```

```
#if ABE == 3
#if ZOO < 12
#if ZIP == 'g'
#if (ABE + 2 - 3 * ZIP) > (ZIP - 2)
In most uses, expression is simple relational, often equality test
#if SPARKY == '7'
```

#else directive

- #else marks the beginning of statement(s) to be compiled if the preceding #if or #elif expression is zero (false)
- Statements following #else are bounded by matching #endif

Example:

```
#if OS = 'A'
     system( "clear" );
#else
     system( "cls" );
#endif
```

4. Miscellaneous Directives(Multiple #includes -- #pragma once)

There's a problem sometimes where a .h file is #included into a file more than one time resulting in compile errors. This can be a serious problem. Because of this, you want to avoid #including .h files in other .h files if at all possible. On the other hand, #including .h files in .c files is fine. If you are lucky, your compiler will support the #pragma oncefeature which automatically prevents a single file from being #included more than once inany one file. This largely solves multiple #include problems.

```
// foo.h
// The following line prevents problems in files which #include "foo.h"
#pragma once
<rest of foo.h ...>
```

Assert

Array out of bounds references are an extremely common form of C run-time error. Youcan use the assert() function to sprinkle your code with your own bounds checks. A fewseconds putting in assert statements can save you hours of debugging.Getting out all the bugs is the hardest and scariest part of writing a large piece ofsoftware. Assert statements are one of the easiest and most effective helpers for thatdifficult phase.

```
#include <assert.h>
#define MAX_INTS 100
{
      int ints[MAX_INTS];
      i = foo(<something complicated>); // i should be in bounds,
      // but is it really?
      assert(i>=0); // safety assertions
      assert(i<MAX_INTS);
      ints[i] = 0;
```

Depending on the options specified at compile time, the assert() expressions will be leftin the code for testing, or may be ignored. For that reason, it is important to only putexpressions in assert() tests which do not need to be evaluated for the proper functioningof the program...

```
int errCode = foo(); // yes
assert(errCode == 0);
assert(foo() == 0); // NO, foo() will not be called if
// the compiler removes the assert()
```

#undef

On some occasion, it may be desirable to cause a defined name to because ‘undefined’.This can be accomplished by means of the # undef directive.

```
#undef macro templates
```

Can be used .thus thestatement.

```
#undef  PENTIUM
```

Would cause the definition of PENTIUM tobe removed from the system. All subsequent # if def PENTIUM statements wouldevaluates to false.

#pragma

This directive is another special purpose directive that you can use to turn on or off certain features. Pragmas very from one compiler to another. There are certain pragmas available with Microsoft C compiler that deal with formatting source listing and placing comments in the object file generated by the compiler.

# pragma startup and #pragma exit

These directives allow us to specify function that are called upon program startup (before main( )) orprogram exit (just before the program terminates). For Example,

```
voidfun1( );
voidfun2( );
#pragmastartup fun1
#pragmaexit fun2
voidmain( ){printf(“\nInside main”);}
voidfun1( ){printf(“\nInside fun1”);}
voidfun2( ){printf(“\nInside fun2”);}
```

Output:

```
Insidefun1
Insidemain
Insidefun2
Inside main
```

Note: The functionfun1( ) and fun2( ) should neither receive nor return any value.

#pragma warn

On compilation the compiler reports errors and warnings in the programs, if any. Errors provide the programmer with no options, apart from correcting them. Warnings, on the other hand, offer the programmer a hint or suggestion that something may be wrong with a particular piece of code .Two most common situation when warning are displayed.

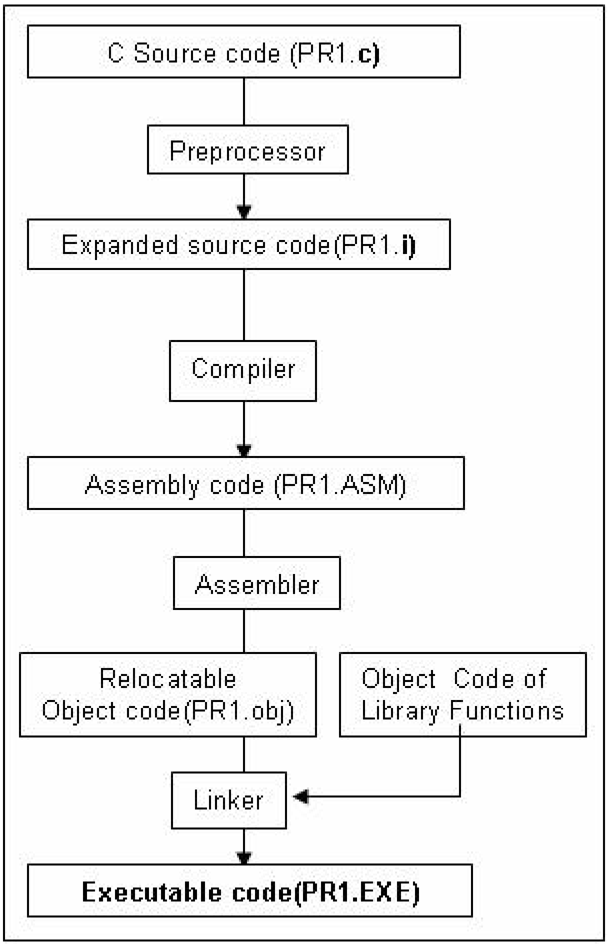
1. If you have written code that the compiler's designers (or the ANSI-C specification) consider bad C programming practice. For ex: if a function does not return a value then it should be declared as void.
2. If you have written code that night causes run-time errors, such as assigning a valueto an uninitialized pointer.

Example: 264, 265 enter here

The Build Process

There are many steps involved in converting a C Program into an executable form. These different steps along with the files created during each stage.

Processor	**Input**	**Output**
Editor	Program typed from Keyword	C source code containing program
Preprocessor	C source code file	Intermediate source code
Compiler	Intermediate source code	Assembly language code
Assembler	Assembly language code	Relocatable Object code in machine language
Linker	Object code of our program and object code of library functions	Executable code in machine Language
Loader	Executable file	



Block Diagram of Building Process in C

C Standard Library <stdlib.h>

The stdlib header defines several general operation functions and macros.

Macros

```
NULL
EXIT_FAILURE
EXIT_SUCCESS
RAND_MAX
MB_CUR_MAX
```

Variables

```
typedef size_t
typedef wchar_t
```

```
struct div_t
struct ldiv_t
```

Functions

```
abort();
abs();
atexit();
atof();
atoi();
atol();
bsearch();
calloc();
div();
exit();
free();
getenv();
labs();
ldiv();
malloc();
mblen();
mbstowcs();
mbtowc();
qsort();
rand();
realloc();
srand();
strtod();
strtol();
strtoul();
system();
wcstombs();
wctomb();
```

Variables and Definitions

`size_t` is the unsigned integer result of the sizeof keyword.
`wchar_t` is an integer type of the size of a wide character constant.
`div_t` is the structure returned by the div function.
`ldiv_t` is the structure returned by the ldiv function.

- `NULL` is the value of a null pointer constant.
- `EXIT_FAILURE` and `EXIT_SUCCESS` are values for the exit function to return termination status.

- RAND_MAX is the maximum value returned by the rand function.
- MB_CUR_MAX is the maximum number of bytes in a multibyte character set which cannot be larger than MB_LEN_MAX.

String Functions

1. atof() - The string pointed to by the argument *str* is converted to a floating-point number (type double). Any initial whitespace characters are skipped (space, tab, carriage return, new line, vertical tab, or formfeed). The number may consist of an optional sign, a string of digits with an optional decimal character, and an optional e or E followed by a optionally signed exponent. Conversion stops when the first unrecognized character is reached.

On success the converted number is returned. If no conversion can be made, zero is returned. If the value is out of range of the type double, then HUGE_VAL is returned with the appropriate sign and ERANGE is stored in the variable errno. If the value is too small to be returned in the type double, then zero is returned and ERANGE is stored in the variable errno.

Declaration: `double atof(const char *str);`

2. atoi() - The string pointed to by the argument *str* is converted to an integer (type int). Any initial whitespace characters are skipped (space, tab, carriage return, new line, vertical tab, or formfeed). The number may consist of an optional sign and a string of digits. Conversion stops when the first unrecognized character is reached.

On success the converted number is returned. If the number cannot be converted, then 0 is returned.

Declaration: `int atoi(const char *str);`

3. atol() - The string pointed to by the argument *str* is converted to a long integer (type long int). Any initial whitespace characters are skipped (space, tab, carriage return, new line, vertical tab, or formfeed). The number may consist of an optional sign and a string of digits. Conversion stops when the first unrecognized character is reached.

On success the converted number is returned. If the number cannot be converted, then 0 is returned.

Declaration: `long int atol(const char *str);`

4. strtod() - The string pointed to by the argument *str* is converted to a floating-point number (type double). Any initial whitespace characters are skipped (space, tab, carriage return, new line, vertical tab, or formfeed). The number may consist of an optional sign, a string of digits with an optional decimal character, and an optional e or E followed by a optionally signed exponent. Conversion stops when the first unrecognized character is reached.

The argument *endptr* is a pointer to a pointer. The address of the character that stopped the scan is stored in the pointer that *endptr* points to.

On success the converted number is returned. If no conversion can be made, zero is returned. If the value is out of range of the type double, then HUGE_VAL is returned with the appropriate sign and ERANGE is stored in the variable errno. If the value is too small to be returned in the type double, then zero is returned and ERANGE is stored in the variable errno.

Declaration: `double strtod(const char *str, char **endptr);`

5. strtol() - The string pointed to by the argument *str* is converted to a long integer (type `long int`). Any initial whitespace characters are skipped (space, tab, carriage return, new line, vertical tab, or formfeed). The number may consist of an optional sign and a string of digits. Conversion stops when the first unrecognized character is reached.

If the *base* (radix) argument is zero, then the conversion is dependent on the first two characters. If the first character is a digit from 1 to 9, then it is base 10. If the first digit is a zero and the second digit is a digit from 1 to 7, then it is base 8 (octal). If the first digit is a zero and the second character is an x or X, then it is base 16 (hexadecimal).

If the *base* argument is from 2 to 36, then that base (radix) is used and any characters that fall outside of that base definition are considered unconvertible. For base 11 to 36, the characters A to Z (or a to z) are used. If the base is 16, then the characters 0x or 0X may precede the number.

The argument *endptr* is a pointer to a pointer. The address of the character that stopped the scan is stored in the pointer that *endptr* points to.

On success the converted number is returned. If no conversion can be made, zero is returned. If the value is out of the range of the type long int, then LONG_MAX or LONG_MIN is returned with the sign of the correct value and ERANGE is stored in the variable errno.

Declaration: `long int strtol(const char *str, char **endptr, int base);`

6. strtoul() - The string pointed to by the argument *str* is converted to an unsigned long integer (type `unsigned long int`). Any initial whitespace characters are skipped (space, tab, carriage return, new line, vertical tab, or formfeed). The number may consist of an optional sign and a string of digits. Conversion stops when the first unrecognized character is reached.

If the *base* (radix) argument is zero, then the conversion is dependent on the first two characters. If the first character is a digit from 1 to 9, then it is base 10. If the first digit is a zero and the second digit is a digit from 1 to 7, then it is base 8 (octal). If the first digit is a zero and the second character is an x or X, then it is base 16 (hexadecimal).

If the *base* argument is from 2 to 36, then that base (radix) is used and any characters that fall outside of that base definition are considered unconvertible. For base 11 to 36, the characters A to Z (or a to z) are used. If the*base* is 16, then the characters 0x or 0X may precede the number.

The argument *endptr* is a pointer to a pointer. The address of the character that stopped the scan is stored in the pointer that *endptr* points to.

On success the converted number is returned. If no conversion can be made, zero is returned. If the value is out of the range of the type unsigned long int, then ULONG_MAX is returned and ERANGE is stored in the variable errno.

Declaration: `unsigned long int strtoul(const char *str, char **endptr, int base);`

Memory Functions

We have studied those functions before in Dynamic Memory Allocation.

1. calloc()

2. free()

3. malloc()

4. realloc()

Environment Functions

1. abort() - Causes an abnormal program termination. Raises the SIGABRT signal and an unsuccessful termination status is returned to the environment. Whether or not open streams are closed is implementation-defined.No return is possible.

Declaration: `void abort(void);`

2. atexit() - Causes the specified function to be called when the program terminates normally. At least 32 functions can be registered to be called when the program terminates. They are called in a last-in, first-out basis (the last function registered is called first).On success zero is returned. On failure a nonzero value is returned.

Declaration: `int atexit(void (*func)(void));`

3. exit() - Causes the program to terminate normally. First the functions registered by atexit are called, then all open streams are flushed and closed, and all temporary files opened with tmpfile are removed. The value of *status* is returned to the environment. If *status* is EXIT_SUCCESS, then this signifies a successful termination. If *status* is EXIT_FAILURE, then this signifies an unsuccessful termination. All other values are implementation-defined.No return is possible.

Declaration: `void exit(int status);`

4. getenv() - Searches for the environment string pointed to by *name* and returns the associated value to the string. This returned value should not be written to.If the string is found, then a pointer to the string's associated value is returned. If the string is not found, then a null pointer is returned.

Declaration: `char *getenv(const char *name);`

5. system() - The command specified by string is passed to the host environment to be executed by the command processor. A null pointer can be used to inquire whether or not the command processor exists.

If string is a null pointer and the command processor exists, then zero is returned. All other return values are implementation-defined.

Declaration: `int system(const char *string);`

Searching and Sorting Functions

1. bsearch() - Performs a binary search. The beginning of the array is pointed to by *base*. It searches for an element equal to that pointed to by *key*. The array is *nitems* long with each element in the array *size* bytes long.

The method of comparing is specified by the *compar* function. This function takes two arguments, the first is the key pointer and the second is the current element in the array being compared. This function must return less than zero if the compared value is less than the specified key. It must return zero if the compared value is equal to the specified key. It must return greater than zero if the compared value is greater than the specified key.

The array must be arranged so that elements that compare less than key are first, elements that equal key are next, and elements that are greater than key are last.

If a match is found, a pointer to this match is returned. Otherwise a null pointer is returned. If multiple matching keys are found, which key is returned is unspecified.

Declaration:

```
void *bsearch(const void *key, const void *base, size_t nitems, size_t size, int
(*compar)(const void *, const void *));
```

2. qsort() -Sorts an array. The beginning of the array is pointed to by *base*. The array is *nitems* long with each element in the array *size* bytes long.

The elements are sorted in ascending order according to the *compar* function. This function takes two arguments. These arguments are two elements being compared. This function must return less than zero if the first argument is less than the second. It must return zero if the first argument is equal to the second. It must return greater than zero if the first argument is greater than the second.

If multiple elements are equal, the order they are sorted in the array is unspecified.

No value is returned.

Declaration:

```
void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *,
const void*));
```

Example:

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

void main(void)
{
  char string_array[10][50]={"John",
                             "Jane",
                             "Mary",
                             "Rogery",
                             "Dave",
                             "Paul",
                             "Beavis",
                             "Astro",
                             "George",
                             "Elroy"};

  /* Sort the list */
  qsort(string_array,10,50,strcmp);

  /* Search for the item "Elroy" and print it */
  printf("%s",bsearch("Elroy",string_array,10,50,strcmp));
}
```

Math Functions

1. abs() - Returns the absolute value of *x*. Note that in two's compliment that the most maximum number cannot be represented as a positive number. The result in this case is undefined.The absolute value is returned.

Declaration: `int abs(int x);`

2. div() - Divides *numer* (numerator) by *denom* (denominator). The result is stored in the structure `div_t` which has two members:

```
int qout;
int rem;
```

Where *quot* is the quotient and *rem* is the remainder. In the case of inexact division, *quot* is rounded down to the nearest integer. The value *numer* is equal to `quot * denom + rem`.The value of the division is returned in the structure.

Declaration: `div_t div(int numer, int denom);`

3. labs() - Returns the absolute value of *x*. Not that in two's compliment that the most maximum number cannot be represented as a positive number. The result in this case is undefined.The absolute value is returned.

Declaration: `long int labs(long int x);`

4. ldiv() - Divides *numer* (numerator) by *denom* (denominator). The result is stored in the structure `ldiv_t` which has two members:

```
long int qout;
long int rem;
```

Where *quot* is the quotient and *rem* is the remainder. In the case of inexact division, *quot* is rounded down to the nearest integer. The value *numer* is equal to `quot * denom + rem`.The value of the division is returned in the structure.

Declaration: `ldiv_t ldiv(long int numer, long int denom);`

5. rand() -Returns a pseudo-random number in the range of `0` to RAND_MAX.The random number is returned.

Declaration: `int rand(void);`

6. srand() - This function seeds the random number generator used by the function `rand`. Seeding `srand` with the same seed will cause `rand` to return the same sequence of pseudo-random numbers. If `srand` is not called, `rand` acts as if `srand(1)` has been called.No value is returned.

Declaration: `void srand(unsigned int seed);`

16. File Handling

A file is a collection of logically related records and is better storage method than array. The data stored in array instead of file but an array stay only in memory & to store it in secondary memory there is no way other than to use file.

A file provides permanent data storage.

File Operation :

1. Creating a file
2. Opening an existing file
3. Reading from a file
4. Writing to a file
5. Moving to the specific location in a file
6. Closing a file

File I/O Functions :

1. fopen() : Creating a new file for use or open an existing file for use.
2. fclose() : Closes a file which had been opened for use.
3. getc() : Reading a character from a file. It can also called as fgetc().
4. putc() : Write a character to a file. It can also called as fputc().
5. fprintf() : Write a set of data values to a file.
6. fscanf() : Read a set of data values from a file.
7. getw() : Read an integer from a file.
8. putw() : Write an integer to a file.
9. fseek() : Sets the position to a desired point in a file.
10. ftell() : Gives the current position in a file(in terms of bytes from start).
11. rewind() : Set the position to the beginning of the file.

A Sample Program to Read from a file:

```
#include<stdio.h>
void main()
{
      FILE *fp;
      charch;
      fp=fopen("PR1.C","r");
      while(1)
      {
            ch=fgetc(fp);
            if(ch==EOF)
```

```
                break;
            printf("%c",ch);
        }
        fclose(fp);
}
```

- To open a file, we have called the function fopen().
- “r” is a string for read mode.
- fopen() perform three important tasks:
    1. Firstly search it on the disk, the file to be opened.
    2. Then load the file from the disk into the primary memory called buffer.
    3. Setup char pointer that points the first character of the buffer.
- To be able successfully read from a file information like mode of opening, size of file, place of file, where the next read operation could be performed has to be maintained. All information are interrelated is gathered by fopen() in a structure called file. fopen() return address of this structure which we have collected in the structure pointer called fp.
- To read a file content from memory, there exists a function called fgetc().

Note :- A special character whose ASCII value is 26 signifies EOF(End of File). This character is inserted behind the last character of a file character.

File Opening Mode

There are certain modes to open a file and those are described below:

1. “r” : Reading from the file.
2. “w” : Writing to the file.
3. “a” : Adding new contents at the end of the file.
4. “r+” : Pointer points the first character in it reading existing contents, writing new contents, modifying existing contents to the file.
5. “w+” : Writing new contents, reading them back and modifying existing.
6. “a+” : Reading existing contents, appending new contents to the end of the file. In this mode we are not been able to modify the existing content.

Random Access to Files

1. **ftell()** : Take a file pointer and return a number of type long that correspondence to the current position.

   ```
   long n;
   n=ftell(fp);
   ```

   n would be five the relative offset(in bytes) of the current position.
2. **rewind()** : Take a file pointer and reset the position to the start of the file. For example,

   ```
   rewind(fp);
   ```
3. **fseek()** : `fseek(file pointer, offset, position);`

   Position – can take one of the following 3 values.

Value	**-**	**Meaning**
0	-	Beginning of file
1	-	Current position of PTR
2	-	End of File

```
fseek(fp,5L,0);
fseek(fp,4L,1);
fseek(fp,-4L,2);
```

If seek operation is successful, then it will return a 0 otherwise it returns 1.

Ques. WAP to implement file handling.

```
#include<stdio.h>
#include<conio.h>
void main()
{
      FILE *fp;
      char c;
      printf("Data input \n\n");
      fp=fopen("Inputfile.txt","w");
      while((c=getch())!=EOF)
      {
            putc(c,fp);
      }
      fclose(fp);
      getch();
}
```

Ques. WAP to create a file and open that file.

```
#include<stdio.h>
#include<conio.h>
void main()
{
      FILE *f1,*f2;
      char c,fname[10];
      clrscr();
      f1=fopen("fname","r");
      if(f1==NULL)
      {
            printf("File Can't Open");
            exit();
      }
      f2=fopen("D:\fname2.txt","w");
      if(f2==NULL)
      {
            printf("File Can't Open");
            fclose(f1);
            exit();
      }
      while(1)
      {
            c=getc(f1);
            if(c==EOF)
                  break;
            else
                  fputc(c,f2);
      }
      fclose(f1);
      fclose(f2);
      getch();
}
```

PROGRAM INDEX