Overview to C

Why use C?

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

1. the portability of the compiler;
2. the standard library concept;
3. a powerful and varied repertoire of operators;
4. an elegant syntax;
5. ready access to the hardware when needed;
6. and the ease with which applications can be optimized by hand-coding isolated procedures

C is often called a "Middle Level" programming language. This is not a reflection on its lack of programming power but more a reflection on its capability to access the system's low level functions. Most high-level languages (e.g. FORTRAN) provide everything the programmer might want to do already build into the language. A low level language (e.g. assembler) provides nothing other than access to the machines basic instruction set. A middle level language, such as C, probably doesn't supply all the constructs found in high-languages - but it provides you with all the building blocks that you will need to produce the results you want!

Uses of C

C was initially used for system development work, in particular the programs that make-up the operating system. Why use C? Mainly because it produces

code that runs nearly as fast as code written in assembly language. Some examples of the use of C might be:

1. Operating Systems
2. Language Compilers
3. Assemblers
4. Text Editors
5. Print Spoolers
6. Network Drivers
7. Modern Programs
8. Data Bases
9. Language Interpreters
10. Utilities

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

A Brief History of C

C is a general-purpose language which has been closely associated with the **UNIX** operating system for which it was developed - since the system and most of the programs that run it are written in C. Many of the important ideas of C stem from the language **BCPL**, developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language **B**, which was written by Ken Thompson in 1970 at Bell Labs, for the first UNIX system on a **DEC** PDP-7. **BCPL** and **B** are "type less" languages whereas C provides a variety of data types. In 1972 Dennis Ritchie at Bell Labs writes C and in 1978 the publication of The C Programming Language by Kernighan & Ritchie caused a revolution in the computing world. In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or "ANSI C", was completed late 1988.

Keywords and Identifiers

Character set

Character set are the set of alphabets, letters and some special characters that are valid in C language.

Alphabets:

Uppercase: A B C ..... X Y Z
Lowercase: a b c ..... x y z

Digits:

0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 .....

Special Characters:

Special Characters in C language

,	<	>	.	_	(	)	;	$	:	%	[	]	#	?
'	&	{	}	"	^	!	*	/	|	-	\	~	+	

White space Characters:

blank space, new line, horizontal tab, carriage return and form feed

Keywords

Keywords are the reserved words used in programming. Each keywords has fixed meaning and that cannot be changed by user. For example: int money;

Here, int is a keyword that indicates, 'money' is of type integer. As, C programming is case sensitive, all keywords must be written in lowercase.

Keywords in C Language:

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

Besides these keywords, there are some additional keywords supported by Turbo C. Additional Keywords for Borland C asm far interrupt pascal near huge cdecl All these keywords, their syntax and application will be discussed in their respective topics.

Identifiers

In C programming, identifiers are names given to C entities, such as variables, functions, structures etc. Identifier are created to give unique name to C entities to identify it during the execution of program. For example:

int money;

int mango_tree;

Here, money is a identifier which denotes a variable of type integer. Similarly, mango_tree is another identifier, which denotes another variable of type integer.

Rules for writing identifier:

1. An identifier can be composed of letters (both uppercase and lowercase letters), digits and underscore '_' only.
2. The first letter of identifier should be either a letter or an underscore. But, it is discouraged to start an identifier name with an underscore though it is legal. It is because, identifier that starts with underscore can conflict with system names. In such cases, compiler will complain about it. Some system names that start with underscore are _fileno, _iob, _wfopen etc.
3. There is no rule for the length of an identifier. However, the first 31 characters of an identifier are discriminated by the compiler. So, the first 31 letters of two identifiers in a program should be different.

Tips for Good Programming Practice:

Programmer can choose the name of identifier whatever they want. However, if the programmer choose meaningful name for an identifier, it will be easy to understand and work on, particularly in case of large program.

Variables and Constants

Variables

Variables are memory location in computer's memory to store data. To indicate the memory location, each variable should be given a unique name called identifier. Variable names are just the symbolic representation of a memory location. Examples of variable name: sum, car_no, count etc.
int num;
Here, num is a variable of integer type.

Rules for writing variable name in C

1. Variable name can be composed of letters (both uppercase and lowercase letters), digits and underscore '_' only.
2. The first letter of a variable should be either a letter or an underscore. But, it is discouraged to start variable name with an underscore though it is legal. It is because, variable name that starts with underscore can conflict with system names and compiler may complain.
3. There is no rule for the length of length of a variable. However, the first 31 characters of a variable are discriminated by the compiler. So, the first 31 letters of two variables in a program should be different.
   In C programming, you have to declare variable before using it in the program.

Constants

Constants are the terms that can't be changed during the execution of a program. For example: 1, 2.5, "Programming is easy." etc. In C, constants can be classified as:

Integer constants

Integer constants are the numeric constants (constant associated with number) without any fractional part or exponential part. There are three types of integer constants in C language: decimal constant(base 10), octal constant(base 8) and hexadecimal constant(base 16) .

Decimal digits: 0 1 2 3 4 5 6 7 8 9

Octal digits: 0 1 2 3 4 5 6 7

Hexadecimal digits: 0 1 2 3 4 5 6 7 8 9 A B C D E F.

For example:

Decimal constants: 0, -9, 22 etc
Octal constants: 021, 077, 033 etc
Hexadecimal constants: 0x7f, 0x2a, 0x521 etc
Notes:

1. You can use small caps a, b, c, d, e, f instead of uppercase letters while writing a hexadecimal constant.
2. Every octal constant starts with 0 and hexadecimal constant starts with 0x in C programming.

Floating-point constants

Floating point constants are the numeric constants that has either fractional form or exponent form. For example:

-2.0
0.0000234
-0.22E-5
Note: Here, E-5 represents 10-5. Thus, -0.22E-5 = -0.0000022.

Character constants

Character constants are the constant which use single quotation around characters. For example: 'a', 'l', 'm', 'F' etc.

Escape Sequences

Sometimes, it is necessary to use newline(enter), tab, quotation mark etc. in the program which either cannot be typed or has special meaning in C programming. In such cases, escape sequence are used. For example: \n is used for newline. The backslash(\ ) causes "escape" from the normal way the characters are interpreted by the compiler.

Escape Sequences	Character
\b	Backspace
\f	Form feed
\n	Newline
\r	Return
\t	Horizontal tab
\v	Vertical tab
\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
\?	Question mark
\0	Null character

String constants

String constants are the constants which are enclosed in a pair of double-quote marks. For example:

"good"	//string constant
""	//null string constant
" "	//string constant of six white space
"x"	//string constant having single character.
"Earth is round\n"	//prints string with newline

Enumeration constants

Keyword enum is used to declare enumeration types. For example:

enum color {yellow, green, black, white};

Here, the variable name is color and yellow, green, black and white are the enumeration constants having value 0, 1, 2 and 3 respectively by default.

Data Types

In C, variable (data) should be declared before it can be used in program. Data types are the keywords, which are used for assigning a type to a variable.

Data types in C

1. Fundamental Data Types
    - Integer types
    - Floating Type
    - Character types
2. Derived Data Types
    - Arrays
    - Pointers
    - Structures
    - Enumeration

Syntax for declaration of a variable

data_type variable_name;

Integer data types

Keyword int is used for declaring the variable with integer type. For example:

int var1;

Here, var1 is a variable of type integer.

The size of int is either 2 bytes (In older PC's) or 4 bytes. If you consider an integer having size of 4 byte (equal to 32 bits), it can take 2^{32} distinct states

as: -2^{31},$-2^{31}+1$, ...,-2, -1, 0, 1, 2, ..., $2^{31}-2$, $2^{31}-1$

Similarly, int of 2 bytes, it can take 216 distinct states from -2^{15} to $2^{15}-1$. If you try to store larger number than $2^{31}-1$, i.e. 2147483647 and smaller number than -2^{31}, i.e, -2147483648, program will not run correctly.

Floating types

Variables of floating types can hold real values (numbers) such as: 2.34, -9.382 etc. Keywords either float or double is used for declaring floating type variable. For example:

```
float var2;
double var3;
```

Here, both var2 and var3 are floating type variables.

In C, floating values can be represented in exponential form as well. For example:

```
float var3=22.442e2
```

Difference between float and double

Generally the size of float (Single precision float data type) is 4 bytes and that of double (Double precision float data type) is 8 bytes. Floating point variables has a precision of 6 digits whereas the precision of double is 14 digits.

Note: Precision describes the number of significant decimal places that a floating values carries.

Character types

Keyword char is used for declaring the variable of character type. For example:

char var4='h';

Here, var4 is a variable of type character which is storing a character 'h'.

The size of char is 1 byte. The character data type consists of ASCII characters. Each character is given a specific value. For example:

For, 'a', value =97
For, 'b', value=98
For, 'A', value=65
For, '&', value=33
For, '2', value=49

Qualifiers

Qualifiers alters the meaning of base data types to yield a new data type.

Size qualifiers:

Size qualifiers alters the size of basic data type. The keywords long and short are two size qualifiers. For example:

long int i;

The size of int is either 2 bytes or 4 bytes but, when long keyword is used, that variable will be either 4 bytes of 8 bytes. If the larger size of variable is not needed then, short keyword can be used in similar manner as long keyword.

Sign qualifiers:

Whether a variable can hold only positive value or both values is specified by sign qualifiers. Keywords signed and unsigned are used for sign qualifiers.

unsigned int a;
// unsigned variable can hold zero and positive values only

It is not necessary to define variable using keyword signed because, a variable is signed by default. Sign qualifiers can be applied to only int and char data types. For a int variable of size 4 bytes it can hold data from -2^{31} to $2^{31}-1$ but, if that variable is defined unsigned, it can hold data from 0 to $2^{32} - 1$.

Constant qualifiers

Constant qualifiers can be declared with keyword const. An object declared by const cannot be modified.

const int p=20;

The value of p cannot be changed in the program.

Volatile qualifiers:

A variable should be declared volatile whenever its value can be changed by some external sources outside program. Keyword volatile is used to indicate volatile variable

Input Output

ANSI standard has defined many library functions for input and output in C language. Functions printf() and scanf() are the most commonly used to display out and take input respectively. Let us consider an example:

```
#include <stdio.h>              //This is needed to run printf() function.
int main()
{
printf("C Programming"); //displays the content inside quotation
return 0;
}
```

Output : C Programming

Explanation of How this program works

1. Every program starts from main() function.
2. *printf()* is a library function to display output which only works if *#include<stdio.h>* is included at the beginning.
3. Here, *stdio.h* is a header file (standard input output header file) and *#include* is command to paste the code from the header file when necessary. When compiler encounters *printf()* function and doesn't find *stdio.h* header file, compiler shows error.
4. Code *return 0*; indicates the end of program. You can ignore this statement but, it is good programming practice to use *return 0*;.

I/O of integers in C

```
#include<stdio.h>
int main(){
int c=5;
printf("Number=%d",c);
return 0;}
```

Output : Number=5
Inside quotation of *printf()* there, is a conversion format string "%d" (for integer). If this conversion format string matches with remaining argument, i.e., c in this case, value of c is displayed.

```
#include<stdio.h>
int main()
{
int c;
printf()("Enter a number\n");
scanf()("%d",&c);
printf()("Number=%d",c);
return 0;
}
```

Output

```
Enter a number
4
Number=4
```

The scanf() function is used to take input from user. In this program, the user is asked a input and value is stored in variable c. Note the '&' sign before c. &c denotes the address of c and value is stored in that address.

I/O of floats in C

```
#include <stdio.h>
int main()
{
float a;
printf("Enter value: ");
scanf("%f",&a);
printf("Value=%f",a); //%f is used for floats instead of %d
```

```
return 0;
}
```

Output

```
Enter value: 23.45
Value=23.450000
```

Conversion format string "%f" is used for floats to take input and to display floating value of a variable.

I/O of characters and ASCII code

```
#include<stdio.h>
int main()
{
char var1;
printf("Enter character: ");
scanf("%c",&var1);
printf("You entered %c.",var1);
return 0;
}
```

Output

```
Enter character: g
You entered g.
```

Conversion format string "%c" is used in case of characters.

ASCII code

When character is typed in the above program, the character itself is not recorded a numeric value(ASCII value) is stored. And when we displayed that value by using "%c", that character is displayed.

```
#include <stdio.h>
int main()
{
char var1;
printf("Enter character: ");
scanf("%c",&var1);
printf("You entered %c.\n",var1);
/* \n prints the next line(performs work of enter). */
printf("ASCII value of %d",var1);
return 0;
}
```

Output

```
Enter character:
g
103
```

When, 'g' is entered, ASCII value 103 is stored instead of g.

You can display character if you know ASCII code only. This is shown by following example.

```
#include<stdio.h>
int main()
{
int var1=69;
printf("Character of ASCII value 69: %c",var1);
return 0;
}
```

Output

```
Character of ASCII value 69: E
```

The ASCII value of 'A' is 65, 'B' is 66 and so on to 'Z' is 90. Similarly ASCII value of 'a' is 97, 'b' is 98 and so on to 'z' is 122.

More about Input/Output of floats and Integer

Variations in Output for integer an floats

Integer and floating-points can be displayed in different formats in C programming as:

```
#include<stdio.h>
int main()
{
printf("Case 1:%6d\n",9876);
/* Prints the number right justified within 6 columns */
printf("Case 2:%3d\n",9876);
/* Prints the number to be right justified to 3 columns but, there are 4 digits so number is not right justified */
printf("Case 3:%.2f\n",987.6543);
/* Prints the number rounded to two decimal places */
printf("Case 4:%.f\n",987.6543);
/* Prints the number rounded to 0 decimal place, i.e, rounded to integer */
printf("Case 5:%e\n",987.6543);
/* Prints the number in exponential notation(scientific notation) */
return 0;
}
```

Output

```
Case 1: 9876
Case 2:9876
Case 3:987.65
Case 4:988
Case 5:9.876543e+002
```

Variations in Input for integer and floats

```
#include<stdio.h>
int main()
{
int a,b;
float c,d;
printf("Enter two intgers: ");
/* Two integers can be taken from user at once as below */
scanf("%d%d",&a,&b);
printf("Enter intger and floating point numbers: ");
/* Integer and floating point number can be taken at once from user as
below */
scanf("%d%f",&a,&c);
return 0;
}
```

Similarly, any number of input can be taken at once from user.

Operators

Operators are the symbol which operates on value or a variable. For example: + is a operator to perform addition.
C programming language has wide range of operators to perform various operations. For better understanding of operators, these operators can be classified as:

Operators in C programming
Arithmetic Operators
Increment and Decrement Operators
Assignment Operators
Relational Operators
Logical Operators
Conditional Operators
Bitwise Operators
Special Operators

Arithmetic Operators

Operator	**Meaning of Operator**
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
%	remainder after division(modulo division)
/	division
%	remainder after division(modulo division)

Example of working of arithmetic operators

```
/* Program to demonstrate the working of arithmetic operators in C. */
#include<stdio.h>
int main()
{
int a=9,b=4,c;
c=a+b;
printf("a+b=%d\n",c);
c=a-b;
printf("a-b=%d\n",c);
c=a*b;
printf("a*b=%d\n",c);
c=a/b;
printf("a/b=%d\n",c);
c=a%b;
printf("Remainder when a divided by b=%d\n",c);
return 0;
}
```

Output

```
a+b=13
a-b=5
a*b=36
a/b=2
Remainder when a divided by b=1
```

Explanation

Here, the operators +, - and * performed normally as you expected. In normal calculation, 9/4 equals to 2.25. But, the output is 2 in this program. It is because, a and b are both integers. So, the output is also integer and the compiler neglects the term after decimal point and shows answer 2 instead of 2.25. And, finally a%b is 1,i.e. ,when a=9 is divided by b=4, remainder is 1.

Suppose a=5.0, b=2.0, c=5 and d=2
In C programming,
a/b=2.5
a/d=2.5
c/b=2.5
c/d=2

Note: % operator can only be used with integers.

Increment and decrement operators

In C, ++ and -- are called increment and decrement operators respectively. Both of these operators are unary operators, i.e, used on single operand. ++ adds 1 to operand and -- subtracts 1 to operand respectively. For example:

Let a=5 and b=10
a++; //a becomes 6
a--; //a becomes 5
++a; //a becomes 6
--a; //a becomes 5

When i++ is used as prefix(like: ++var), ++var will increment the value of var and then return it but, if ++ is used as postfix(like: var++), operator will return the value of operand first and then only increment it. This can be demonstrated by an example:

```
#include<stdio.h>
int main()
{
int c=2,d=2;
printf("%d\n",c++); //this statement displays 2 then, only c incremented by 1 to 3.
printf("%d",++c); //this statement increments 1 to c then, only c is displayed.
```

```
return 0;
}
```

Output

```
2
4
```

Assignment Operators

The most common assignment operator is =. This operator assigns the value in right side to the left side. For example:

```
var=5 //5 is assigned to var
a=c; //value of c is assigned to a
5=c; // Error! 5 is a constant.
```

Operator	Example	Same as
=	a=b	a=b
+=	a+=b	a=a+b
-=	a-=b	a=a-b
=	a=b	a=a*b
/=	a/=b	a=a/b
%=	a%=b	a=a%b

Relational Operator

Relational operators checks relationship between two operands. If the relation is true, it returns value 1 and if the relation is false, it returns value 0. For example:

```
a>b
```

Here, > is a relational operator. If a is greater than b, a>b returns 1 if not then, it returns 0.

Relational operators are used in decision making and loops in C programming.

Operator	Meaning of Operator	Example
==	Equal to	5==3 returns false (0)
>	Greater than	5>3 returns true (1)
<	Less than	5<3 returns false (0)
!=	Not equal to	5!=3 returns true(1)
>=	Greater than or equal to	5>=3 returns true (1)
<=	Less than or equal to	5<=3 return false (0)

Logical Operators

Logical operators are used to combine expressions containing relation operators. In C, there are 3 logical operators:

Operator	Meaning of Operator	Example
&&	Logial AND	If c=5 and d=2 then,((c==5) && (d>5)) returns false.
||	Logical OR	If c=5 and d=2 then, ((c==5) || (d>5)) returns true.
!	Logical NOT	If c=5 then, !(c==5) returns false.

Explanation

For expression, ((c==5) && (d>5)) to be true, both c==5 and d>5 should be true but, (d>5) is false in the given example. So, the expression is false. For expression ((c==5) || (d>5)) to be true, either the expression should be true. Since, (c==5) is true. So, the expression is true. Since, expression (c==5) is true, !(c==5) is false.

Conditional Operator

Conditional operator takes three operands and consists of two symbols ? and : . Conditional operators are used for decision making in C. For example:

c=(c>0)?10:-10;

If c is greater than 0, value of c will be 10 but, if c is less than 0, value of c will be -10.

Bitwise Operators

A bitwise operator works on each bit of data. Bitwise operators are used in bit level programming.

Operators	Meaning of operators
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right

Bitwise operator is advance topic in programming . .

Other Operators

Comma Operator

Comma operators are used to link related expressions together. For example:

int a,c=5,d;

The sizeof operator

It is a unary operator which is used in finding the size of data type, constant, arrays, structure etc. For example:

```
#include<stdio.h>
int main()
{
int a;
float b;
double c;
char d;
printf("Size of int=%d bytes\n",sizeof(a));
printf("Size of float=%d bytes\n",sizeof(b));
printf("Size of double=%d bytes\n",sizeof(c));
printf("Size of char=%d byte\n",sizeof(d));
return 0;
}
```

Output

```
Size of int=4 bytes
Size of float=4 bytes
Size of double=8 bytes
Size of char=1 byte
```

Conditional operators (?:)

Conditional operators are used in decision making in C programming, i.e, executes different statements according to test condition whether it is either true or false.

Syntax of conditional operators

conditional_expression?expression1:expression2

If the test condition is true, *expression1* is returned and if false *expression2* is returned.

Example of conditional operator

```
#include<stdio.h>
int main()
{
char feb;
int days;
printf("Enter l if the year is leap year otherwise enter 0: ");
scanf("%c",&feb);
days=(feb=='l')?29:28;
/*If test condition (feb=='l') is true, days will be equal to 29. */
/*If test condition (feb=='l') is false, days will be equal to 28. */
printf("Number of days in February = %d",days);
return 0;
}
```

Output

```
Enter l if the year is leap year otherwise enter n: l
Number of days in February = 29
```

Other operators such as &(reference operator), *(dereference operator) and ->(member selection) operator will be discussed in pointer chapter.

Bitwise Operators

Bitwise operators are special types of operators that are used in programming the processor. In processor, mathematical operations like: addition, subtraction, addition and division are done using the bitwise operators which makes processing faster and saves power.

Operators	Meaning of operators
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right

Bitwise AND operator in C programming.

The output of logical AND is 1 if both the corresponding bits of operand is 1. If either of bit is 0 or both bits are 0, the output will be 0. It is a binary operator (works on two operands) and indicated in C programming by & symbol. Let us suppose the bitwise AND operation of two integers 12 and 25.

12 = 00001100 (In Binary)
25 = 00011001 (In Binary)
Bit Operation of 12 and 25
  00001100
& 00011001
  00001000 = 8 (In decimal)

As, every bitwise operator works on each bit of data. The corresponding bits of two inputs are check and if both bits are 1 then only the output will be 1. In this case, both bits are 1 at only one position,i.e, fourth position from the right, hence the output bit of that position is 1 and all other bits are 0.

```
#include<stdio.h>
int main()
{
int a=12,b=39;
printf("Output=%d",a&b);
return 0;
}
```

Output

```
Output=4
```

Bitwise OR operator in C

The output of bitwise OR is 1 if either of the bit is 1 or both the bits are 1. In C Programming, bitwise OR operator is denoted by |.

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25
  00001100
| 00011001
  00011101 = 29 (In decimal)
```

```
#include<stdio.h>
int main()
{
int a=12,b=25;
```

```
printf("Output=%d",a&b);
return 0;
}
```

Output

Output=29

C Programming Bitwise XOR(exclusive OR) operator

The output of bitwise XOR operator is 1 if the corresponding bits of two operators are opposite(i.e., To get corresponding output bit 1; if corresponding bit of first operand is 0 then, corresponding bit of second operand should be 1 and vice-versa.). It is denoted by ^.

12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bitwise XOR Operation of 12 and 25
  00001100
| 00011001
 00010101 = 21 (In decimal)

```
#include<stdio.h>
int main()
{
int a=12,b=25;
printf("Output=%d",a^b);
return 0;
}
```

Output
Output=21

Bitwise compliment operator

Bitwise compliment operator is an unary operator(works on one operand only). It changes the corresponding bit of the operand to opposite bit, i.e., 0 to 1 and 1 to 0. It is denoted by ~.

35=00100011 (In Binary)

Bitwise complement Operation of 35

~ 00100011
11011100 = 220 (In decimal)

Twist in bitwise complement operator in C Programming

Output of ~35 shown by compiler won't be 220, instead it shows -36. For any integer n, bitwise complement of n will be -(n+1). To understand this, you should understand the concept of 2's complement.

2's Complement

Two's complement is the operation on binary numbers which allows number to write it in different form. The 2's complement of number is equal to the complement of number plus 1. For example:

Decimal	Binary	2's complement
0	0000000	-(11111111+1) = -00000000 = -0(decimal)
1	00000001	-(11111110+1) = -11111111 = -256(decimal)
12	00001100	-(11110011+1) = -11110100 = -244(decimal)
220	11011100	-(00100011+1) = -00100100 = -36(decimal)

Note: Overflow is ignored while computing 2's complement.

If we consider the bitwise complement of 35, 220(in decimal) is converted into 2's complement which is -36. Thus, the output shown by computer will be -36 instead of 220.

How is bitwise complement of any number N=-(N+1)?

bitwise complement of N= ~N (represented in 2's complement form)
2'complement of N= -((~N)+1) = -(N+1)

```
#include<stdio.h>
int main()
{
printf("complement=%d\n",~35);
printf("complement=%d\n",~-12);
return 0;
}
```

Output

```
complement=-36
complement=11
```

Shift Operator in C programming

There are two shift operators in C programming:

1. Right shift operator
2. Left shift operator.

Right Shift Operator

Right shift operator moves the all bits towards the right by certain number of bits which can be specified. It is denoted by >>.

212 = 11010100 (In binary)
212>>2 = 00110101 (In binary) [Right shift by two bits]
212>>7 = 00000001 (In binary)
212>>8 = 00000000
212>>0 = 11010100 (No Shift)

Left Shift Operator

Left shift operator moves the all bits towards the left by certain number of bits which can be specified. It is denoted by <<.

212 = 11010100 (In binary)
212<<1 = 110101000 (In binary) [Left shift by one bit]
212<<0 =11010100 (Shift by 0)
212<<4 = 110101000000 (In binary) =3392(In decimal)

```
#include<stdio.h>
int main()
{
int num=212,i;
for (i=0;i<=2;++i)
printf("Right shift by %d: %d\n",i,num>>i);
printf("\n");
for (i=0;i<=2;++i)
printf("Left shift by %d: %d\n",i,num<<i);
return 0;
}
```

Output

Right Shift by 0: 212
Right Shift by 1: 106
Right Shift by 2: 53

Left Shift by 0: 212
Left Shift by 1: 424
Left Shift by 2: 848

Interesting thing to note in Left and Right Shift

For any positive number, right shift is equal to integer division of that number by (shift bit plus one) and for any integer left shift is equal to the multiplication of that number by (shift bit plus one).

Precedence and Associativity of Operators

Precedence of operators

If more than one operators are involved in an expression then, C language has predefined rule of priority of operators. This rule of priority of operators is called operator precedence.

In C, precedence of arithmetic operators(*,%,/,+,-) is higher than relational operators(==,!=,>,<,>=,<=) and precedence of relational operator is higher than logical operators(&&, || and !). Suppose an expression:
(a>b+c&&d)
This expression is equivalent to:
((a>(b+c))&&d)
i.e, (b+c) executes first
then, (a>(b+c)) executes
then, (a>(b+c))&&d) executes

Associativity of operators

Associativity indicates in which order two operators of same precedence(priority) executes. Let us suppose an expression:
a==b!=c
Here, operators == and != have same precedence. The associativity of both == and != is left to right, i.e, the expression in left is executed first and execution take pale towards right. Thus, a==b!=c equivalent to :
(a==b)!=c

The table below shows all the operators in C with precedence and associativity.

Operator	Meaning of operator	Associativity
() [] -> .	Functional call Array element reference Indirect member selection Direct member selection	Left to right
! ~ + - ++ -- & * sizeof (type)	Logical negation Bitwise(1 's) complement Unary plus Unary minus Increment Decrement Dereference Operator(Address) Pointer reference Returns the size of an object Type cast(conversion)	Right to left
* / %	Multiply Divide Remainder	Left to right
+ -	Binary plus(Addition) Binary minus(subtraction)	Left to right
<< >>	Left shift Right shift	Left to right
< <= > >=	Less than Less than or equal Greater than Greater than or equal	Left to right
== !=	Equal to Not equal to	Left to right
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise OR	Left to right
&&	Logical AND	Left to right

||	Logical OR	Left to right
?:	Conditional Operator	Left to right
= *= /= %= -= &= ^= |= <<= >>=	Simple assignment Assign product Assign quotient Assign remainder Assign sum Assign difference Assign bitwise AND Assign bitwise XOR Assign bitwise OR Assign left shift Assign right shift	Right to left
,	Separator of expressions	Left to right

Note: Precedence of operators decreases from top to bottom in the given table.

Loop

Loops causes program to execute the certain block of code repeatedly until some conditions are satisfied, i.e.,loops are used in performing repetitive work in programming.

Suppose you want to execute some code/s 100 times. You can perform it by writing that code/s only one time and repeat the execution 100 times using loop.

There are 3 types of loops in C programming:

1. for loop
2. while loop
3. do...while loop

for Loop Syntax

```
for(initial expression; test expression; update expression)
{
code/s to be executed;
}
```

How for loop works in C programming?

The initial expression is initialized only once at the beginning of the loop. Then, the test expression is checked by the program. If the test expression is false, for loop is terminated. But, if test expression is true then, the codes are executed and update expression is updated. Again, the test expression is checked. If it is false, loop is terminated and if it is true, the same process repeats until test expression is false.

This flowchart describes the working of for loop in C programming.

for loop example

Write a program to find the sum of first n natural numbers where n is entered by user. Note: 1,2,3... are called natural numbers.

```
#include<stdio.h>
int main()
{
int n, count, sum=0;
printf("\n Enter the value of n. : ");
scanf("%d",&n);
for(count=1;count<=n;++count) //for loop terminates if count>n
{
sum+=count; /* this statement is equivalent to sum=sum+count */
}
printf("Sum=%d",sum);
return 0;
}
```

Output

```
Enter the value of n. : 19
Sum=190
```

In this program, the user is asked to enter the value of n. Suppose you entered 19 then, count is inatialized to 1 at first. Then, the test expression in the for loop,i.e., (count<= n) becomes true. So, the code in the body of for loop is executed which makes sum to 1. Then, the expression ++count is executed and again the test expression is checked, which becomes true. Again, the body of for loop is executed which makes sum to 3 and this process continues. When count is 20, the test condition becomes false and the for loop is terminated.
Note: Initial, test and update expression are separated by semicolon(;).

IF-ELSE

if, if..else and Nested if...else Statement

Decision making are needed when, the program encounters the situation to choose a particular statement among many statements. In C, decision making can be performed with following two statements.

1. if...else statement
2. switch statement

if statement syntax

```
if (test expression0)
{
statement/s to be executed if test expression is true;
}
```

If the test expression is true then, statements for the body if, i.e, statements inside parenthesis are executed. But, if the test expression is false, the execution of the statements for the body of if statements are skipped.

Example of if statement

Write a C program to print the number entered by user only if the number entered is negative.

```
#include<stdio.h>
int main()
{
int num;
printf("Enter a number to check.\n");
scanf("%d",&num);
```

```
if(num<0) /* checking whether number is less than 0 or not. */
printf("Number=%d\n",num);
/*If test condition is true, statement above will be executed, otherwise
it will not be executed */
printf("The if statement in C programming is easy.");
return 0;
}
```

Output 1

```
Enter a number to check.
-2
Number=-2
The if statement in C programming is easy.
```

When user enters -2 then, the test expression (num<0) becomes true. Hence, Number=-2 is displayed in the screen.

Output 2

```
Enter a number to check.
5
The if statement in C programming is easy.
```

When the user enters 5 then, the test expression (num<0) becomes false. So, the statement for body of if is skipped and only the statement below it is executed.

if...else statement

The if...else statement is used, if the programmer wants to execute some code, if the test expression is true and execute some other code if the test expression is false.

Syntax of if...else

```
if (test expression)
{
statements to be executed if test expression is true;
}
else
{
statements to be executed if test expression is false;
}
```

Example of if...else statement

Write a C program to check whether a number entered by user is even or odd

```
#include<stdio.h>
int main()
{
int num;
printf("Enter a number you want to check.\n");
scanf("%d",&num);
if((num%2)==0) //checking whether remainder is 0 or not.
printf("%d is even.",num);
else
printf("%d is odd.",num);
return 0;
}
```

Output 1

```
Enter a number you want to check.
25
25 is odd.
```

Output 2

```
Enter a number you want to check.
```

2
2 is even.

Nested if...else statement (if...elseif....else Statement)

The if...else statement can be used in nested form when a serious decision are involved.

Syntax of nested if...else statement.

```
if (test expression)
{
statements to be executed if test expression is true;
}
else
{
if(test expression 1)
statements to be executed if test expressions 1 is true;
else
if (test expression 2)
.
.
.
else
statements to be executed if all test expressions are false;
```

How nested if...else works?

If the test expression is true, it will execute the code before else part but, if it is false, the control of the program jumps to the else part and check test expression 1 and the process continues. If all the test expression are false then, the last statement is executed.

The ANSI standard specifies that 15 levels of nesting may be continued.

Example of nested if else statement

Write a C program to relate two integers entered by user using = or > or < sign.

```
#include<stdio.h>
int main()
{
int numb1, numb2;
printf("Enter two integers to check".\n);
scanf("%d %d",&numb1,&numb2);
if(numb1==numb2) //checking whether two integers are equal.
printf("Result: %d=%d",numb1,numb2);
else
if(numb1>numb2) //checking whether numb1 is greater than numb2.
printf("Result: %d>%d",numb1,numb2);
else
printf("Result: %d>%d",numb2,numb1);
return 0;
}
```

Output 1

```
Enter two integers to check.
5
3
Result: 5>3
```

Output 2

```
Enter two integers to check.
-4
-4
Result: -4=-4
```

WHILE Loop

Loops causes program to execute the certain block of code repeatedly until some conditions are satisfied, i.e., loops are used in performing repetitive work in programming.

Suppose you want to execute some code/s 10 times. You can perform it by writing that code/s only one time and repeat the execution 10 times using loop.

Syntax of while loop

```
while (test expression)
{
statements to be executed.
}
```

In the beginning of while loop, test expression is checked. If it is true, codes inside the body of while loop,i.e, code/s inside parentheses are executed and again the test expression is checked and process continues until the test expression becomes false.

Example of while loop

Write a C program to find the factorial of a number, where the number is entered by user.
(Hints: factorial of n = 1*2*3*...*n

```
/*C program to demonstrate the working of while loop*/
#include<stdio.h>
int main()
{
int number,factorial;
```

```
printf("Enter a number.\n");
scanf("%d",&number);
factorial=1;
while (number>0)
{
/* while loop continues util test condition number>0 is true */
factorial=factorial*number;
--number;
}
}
printf("Factorial=%d",factorial);
return 0;
}
```

Output

```
Enter a number.
5
Factorial=120
```

DO-WHILE Loop

Loops causes program to execute the certain block of code repeatedly until some conditions are satisfied, i.e., loops are used in performing repetitive work in programming.

Suppose you want to execute some code/s 10 times. You can perform it by writing that code/s only one time and repeat the execution 10 times using loop.

do...while loop

In C, do...while loop is very similar to while loop. Only difference between these two loops is that, in while loops, test expression is checked at first but, in do...while loop code is executed at first then the condition is checked. So, the code are executed at least once in do...while loops.

Syntax of do...while loops

```
do
{
some code/s;
}
while (test expression);
```

At first codes inside body of do is executed. Then, the test expression is checked. If it is true, code/s inside body of do are executed again and the process continues until test expression becomes false(zero).

Notice, there is semicolon in the end of while (); in do...while loop.

Example of do...while loop

Write a C program to add all the numbers entered by a user until user enters 0.

```
/*C program to demonstrate the working of do...while statement*/
#include<stdio.h>
int main()
{
int sum=0,num;
do /* Codes inside the body of do...while loops are at least executed
once. */
{
printf("Enter a number\n");
scanf("%d",&num);
sum+=num;
}
while(num!=0);
printf("sum=%d",sum);
return 0;
}
```

Output

```
Enter a number
3
Enter a number
-2
Enter a number
0
sum=1
```

In this C program, user is asked a number and it is added with sum. Then, only the test condition in the do...while loop is checked. If the test condition is true,i.e, num is not equal to 0, the body of do...while loop is again executed until num equals to zero

Break and Continue

There are two statement built in C, *break*; and *continue*; to interrupt the normal flow of control of a program. Loops performs a set of operation repeately until certain condition becomes false but, it is sometimes desirable to skip some statements inside loop and terminate the loop immediately without checking the test expression. In such cases, break and continue statements are used.

break Statement

In C programming, break is used in terminating the loop immediately after it is encountered. The break statement is used with conditional if statement.

Syntax of break statement

break;

The break statement can be used in terminating all three loops for, while and do...while loops.

The figure below explains the working of break statement in all three type of loops.

Example of break statement

Write a C program to find average of maximum of n positive numbers entered by user. But, if the input is negative, display the average(excluding the average of negative input) and end the program.

/* C program to demonstrate the working of break statement by terminating a loop, if user inputs negative number*/

```
# include<stdio.h>
int main()
{
float num,average,sum;
int i,n;
printf("Maximum no. of inputs\n");
scanf("%d",&n);
for(i=1;i<=n;++i)
{
printf("Enter n%d: ",i);
scanf("%f",&num);
if(num<0.0)
break; //for loop breaks if num<0.0
sum=sum+num;
}
average=sum/(i-1);
printf("Average=%.2f",average);
return 0;
}
```

Output

```
Maximum no. of inputs
4
Enter n1: 1.5
Enter n2: 12.5
Enter n3: 7.2
Enter n4: -1
Average=7.07
```

In this program, when the user inputs number less than zero, the loop is terminated using break statement with executing the statement below it i.e., without executing *sum=sum+num*.

In C, break statements are also used in switch...case statement.

continue Statement

It is sometimes desirable to skip some statements inside the loop. In such cases, continue statements are used.

Syntax of continue Statement

continue;

Just like break, continue is also used with conditional if statement.

For better understanding of how continue statements works in C programming. Analyze the figure below which bypasses some code/s inside loops using continue statement.

Example of continue statement

Write a C program to find the product of 4 integers entered by a user. If user enters 0 skip it.

```
//program to demonstrate the working of continue statement in C programming
# include<stdio.h>
int main()
{
int i,num,product;
for(i=1,product=1;i<=4;++i)
{
printf("Enter num%d:",i);
scanf("%d",&num);
if(num==0)
continue; / In this program, when num equals to zero, it skips the statement product=num and continue the loop. */
```

```
product*=num;
}
printf("product=%d",product);
return 0;
}
```

Output

```
Enter num1:3
Enter num2:0
Enter num3:-5
Enter num4:2
product=-30
```

Switch Statement

Decision making are needed when, the program encounters the situation to choose a particular statement among many statements. If a programmar has to choose one among many alternatives if...else can be used but, this makes programming logic complex. This type of problem can be handled in C programming using switch...case statement.

Syntax of switch...case

```
switch (expression)
{
case constant1:
codes to be executed if expression equals to constant1; break;
case constant2:
codes to be executed if expression equals to constant3;
break;
.
.
.
default:
codes to be executed if expression doesn't match to any cases;
}
```

In switch...case, expression is either an integer or a character. If the value of switch expression matches any of the constant in case, the relevant codes are executed and control moves out of the switch...case statement. If the expression doesn't matches any of the constant in case, then the default statement is executed.

Example of switch...case statement

Write a program that asks user an arithmetic operator('+','-','*' or '/') and two operands and perform the corresponding calculation on the operands.

```
/* C program to demonstrate the working of switch...case statement */
/* Program to create a simple calculator for addition, subtraction,
multiplication and division */
#include<stdio.h>
int main()
{
char operator;
float num1,num2;
printf("Enter operator +, - , * or / :\n");
operator=getche();
printf("\nEnter two operands:\n");
scanf("%f%f",&num1,&num2);
switch(operator)v {
case '+':
printf("num1+num2=%.2f",num1+num2);
break;
case '-':
printf("num1-num2=%.2f",num1-num2);
break;
case '*':
printf("num1*num2=%.2f",num1*num2);
break;
case '/':
printf("num2/num1=%.2f",num1/num2);
break;
default:
/* if operator is other than +, -, * or /, error message is shown */
printf("Error! operator is not correct");
break;
}
return 0;
```

```
}
```

Output

```
Enter operator +, -, * or / :
/
Enter two operators:
34
3
num2/num1=11.33
```

Notice break statement at the end of each case, which cause switch...case statement to exit. If break statement are not used, all statements below that case statement are also executed.

goto

In C programming, goto statement is used for altering the normal sequence of program execution by transferring control to some other part of the program.

Syntax of goto statement

```
goto label;
.....
.....
.....
label:
statement;
```

In this syntax, label is an identifier. When, the control of program reaches to goto statement, the control of the program will jump to the label: and executes the code/s after it.

Example of goto statement

```
/* C program to demonstrate the working of goto statement.*/
# include<stdio.h>
int main()
{
float num,average,sum;
int i,n;
printf("Maximum no. of inputs: ");
scanf("%d",&n);
for(i=1;i<=n;++i)
{
printf("Enter n%d: ",i);
```

```
scanf("%f",&num);
if(num<0.0)
goto jump; /* control of the program jumps to label jump */
sum=sum+num;
}
jump:
average=sum/(i-1);
printf("Average: %.2f",average);
return 0;
}
```

Output

```
Maximum no. of inputs: 4
Enter n1: 1.5
Enter n2: 12.5
Enter n3: 7.2
Enter n4: -1
Average: 7.07
```

Though goto statement is included in ANSI standard of C, use of goto statement should be reduced as much as possible in a program.

Reasons to avoid goto statement

Though, using goto statement give power to jump to any part of program, using goto statement makes the logic of the program complex and tangled. In modern programming, goto statement is considered a harmful construct and a bad programming practice.

The goto statement can be replaced in most of C program with the use of break and continue statements. In fact, any program in C programming can be perfectly written without the use of goto statement. All programmer should try to avoid goto statement as possible as they can.

Function

Function in programming is a segment that groups a number of program statements to perform specific task.

A C program has at least one function main( ). Without main() function, there is technically no C program.

Types of C functions

Basically, there are two types of functions in C on basis of whether it is defined by user or not.

1. Library function
2. User defined function

Library function

Library functions are the in-built function in C programming system. For example:

main()
- The execution of every C program starts from this main() function.

printf()
- prinf() is used for displaying output in C.

scanf()
- scanf() is used for taking input in C.

User defined function

C provides programmer to define their own function according to their requirement known as user defined functions. Suppose, a

programmer wants to find factorial of a number and check whether it is prime or not in same program. Then, he/she can create two separate user-defined functions in that program: one for finding factorial and other for checking whether it is prime or not.

How user-defined function works in C Programming?

```
#include<stdio.h>
void function_name()
{
.....
}
int main()
{
.....
function_name();
.....
}
```

As mentioned earlier, every C program begins from main() and program starts executing the codes inside main() function. When the control of program reaches to function_name() inside main() function. The control of program jumps to void function_name() and executes the codes inside it. When, all the codes inside that user-defined function are executed, control of the program jumps to the statement just after function_name() from where it is called. Analyze the figure below for understanding the concept of function in C programming. .

Remember, the function name is an identifier and should be unique.

Advantages of user defined functions

1. User defined functions helps to decompose the large program into small segments which makes programmar easy to understand, maintain and debug.

2. If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
3. Programmer working on large project can divide the workload by making different functions.

Example of user-defined function

Write a C program to add two integers. Make a function add to add integers and display sum in main() function.

```
/*Program to demonstrate the working of user defined function*/
#include<stdio.h>
int add(int a, int b); //function prototype(declaration)
int main()
{
int num1,num2,sum;
printf("Enters two number to add\n");
scanf("%d %d",&num1,&num2);
sum=add(num1,num2); //function call
printf("sum=%d",sum);
return 0;
}
int add(int a,int b) //function declarator
{
/* Start of function definition. */
int add;
add=a+b;
return add; //return statement of function
/* End of function definition. */
}
```

Function prototype(declaration):

Every function in C programming should be declared before they are used. These type of declaration are also called function prototype.

Function prototype gives compiler information about function name, type of arguments to be passed and return type.

Syntax of function prototype

return_type function_name(type(1) argument(1),....,type(n) argument(n));

In the above example,int add(int a, int b); is a function prototype which provides following information to the compiler:

1. name of the function is add()
2. return type of the function is int.
3. two arguments of type int are passed to function.

Function prototype are not needed if user-definition function is written before main() function.

Function call

Control of the program cannot be transferred to user-defined function unless it is called invoked).

Syntax of function call

function_name(argument(1),....argument(n));

In the above example, function call is made using statement add(num1,num2); from main(). This make the control of program jump from that statement to function definition and executes the codes inside that function.

Function definition

Function definition contains programming codes to perform specific task.

Syntax of function definition

```
return_type function_name(type(1) argument(1),..,type(n)
argument(n))
{
//body of function
}
```

Function definition has two major components:

1. Function declarator
2. Function body

1. Function declarator

Function declarator is the first line of function definition. When a function is invoked from calling function, control of the program is transferred to function declarator or called function.

Syntax of function declarator

```
return_type function_name(type(1) argument(1),....,type(n)
argument(n))
```

Syntax of function declaration and declarator are almost same except, there is no semicolon at the end of declarator and function declarator is followed by function body.

In above example, int add(int a,int b) in line 12 is a function declarator.

2. Function body

Function declarator is followed by body of function which is composed of statements.

Passing arguments to functions

In programming, argument/parameter is a piece of data(constant or variable) passed from a program to the function.

In above example two variable, num1 and num2 are passed to function during function call and these arguments are accepted by arguments a and b in function definition.

Arguments that are passed in function call and arguments that are accepted in function definition should have same data type. For example:

If argument *num1* was of int type and *num2* was of float type then, argument variable a should be of type int and b should be of type float,i.e., type of argument during function call and function definition should be same.

A function can be called with or without an argument.

Return Statement

Return statement is used for returning a value from function definition to calling function.

Syntax of return statement

return (expression);
OR
return;

For example:

return;
return a;
return (a+b);

In above example, value of variable add in add() function is returned and that value is stored in variable sum in main() function. The data type of expression in return statement should also match the return type of function

Types of Functions

For better understanding of arguments and return in functions, user-defined functions can be categorised as:

1. Function with no arguments and no return value.
2. Function with no arguments and return value.
3. Function with arguments but no return value.
4. Function with arguments and return value.

Let's take an example to find whether a number is prime or not using above 4 cateogories of user defined functions.

Function with no arguments and no return value.

```
/*C program to check whether a number entered by user is prime or not using function with no arguments and no return value*/
#include<stdio.h>
void prime();
int main()
{
prime(); //No argument is passed to prime().
return 0;
}
void prime() {
/* There is no return value to calling function main(). Hence, return type of prime() is void */}
int num,i,flag=0;}
printf("Enter positive integer enter to check:\n");
scanf("%d",&num);
for(i=2;i<=num/2;++i)
{
if(num%i==0)
```

```
{
flag=1;
}
}
if (flag==1)
printf("%d is not prime",num);
else
printf("%d is prime",num);
}
```

Function *prime()* is used for asking user a input, check for whether it is prime of not and display it accordingly. No argument is passed and returned form *prime()* function.

Function with no arguments but return value

```
/*C program to check whether a number entered by user is prime or not using function with no arguments but having return value */
#include<stdio.h>
int input();
int main()
{
int num,i,flag;
num=input(); /* No argument is passed to input() */
for(i=2,flag=i;i<=num/2;++i,flag=i)
{
if(num%i==0)
{
printf("%d is not prime",num);
++flag;
break;
}
}
if(flag==i)
printf("%d is prime",num);
```

```
return 0;
}
int input() { /* Integer value is returned from input() to calling function */
int n;
printf("Enter positive enter to check:\n");
scanf("%d",&n);
return n;
}
```

There is no argument passed to *input()* function But, the value of n is returned from *input()* to *main()* function.

Function with arguments and no return value

```
/*Program to check whether a number entered by user is prime or not using function with arguments and no return value */
#include<stdio.h>
void check_display(int n);
int main()
{
int num;
printf("Enter positive enter to check:\n");
scanf("%d",&num);
check_display(num); /* Argument num is passed to function. */
return 0;
}
void check_display(int n) { /* There is no return value to calling function. Hence, return type of function is void. */
int i,flag;
for(i=2,flag=i;i<=n/2;++i,flag=i)
{
if(n%i==0)
{
printf("%d is not prime",n);
++flag;
```

```
break;
}
}
if(flag==i)
printf("%d is prime",n);
}
```

Here, check_display() function is used for check whether it is prime or not and display it accordingly. Here, argument is passed to user-defined function but, value is not returned from it to calling function.

Function with argument and a return value

```
/* Program to check whether a number entered by user is prime or not using function with argument and return value */
#include<stdio.h>
int check(int n);
int main()
{
int num,num_check=0;
printf("Enter positive enter to check:\n");
scanf("%d",&num);
num_check=check(num); /* Argument num is passed to check() function. */
if(num_check==1)
printf("%d in not prime",num);
else
printf("%d is prime",num);
return 0;
}
int check(int n)
{
/* Integer value is returned from function check() */
int i;
for(i=2;i<=n/2;++i)
```

```
{
if(n%i==0)
return 1;
}
return 0;
}
```

Here, *check()* function is used for checking whether a number is prime or not. In this program, input from user is passed to function *check()* and integer value is returned from it. If input the number is prime, 0 is returned and if number is not prime, 1 is returned.

Recursion

A function that calls itself is known as recursive function and the process of calling function itself is known as recursion in C programming.

Example of recursion in C programming

Write a C program to find sum of first n natural numbers using recursion. Note: Positive integers are known as natural number i.e. 1, 2, 3....n

```
#include<stdio.h>
int sum(int n);
int main()
{
int num,add;
printf("Enter a positive integer:\n");
scanf("%d",&num);
add=sum(num);
printf("sum=%d",add);
}
int sum(int n)
{
if(n==0)
return n;
else
return n+sum(n-1); /*self call to function sum() */
}
```

Output

```
Enter a positive integer:
5
15
```

In, this simple C program, sum() function is invoked from the same function. If n is not equal to 0 then, the function calls itself passing argument 1 less than the previous argument it was called with. Suppose, n is 5 initially. Then, during next function calls, 4 is passed to function and the value of argument decreases by 1 in each recursive call. When, n becomes equal to 0, the value of n is returned which is the sum numbers from 5 to 1.

For better visualization of recursion in this example:

sum(5)
=5+sum(4)
=5+4+sum(3)
=5+4+3+sum(2)
=5+4+3+2+sum(1)
=5+4+3+2+1+sum(0)
=5+4+3+2+1+0
=5+4+3+2+1
=5+4+3+3
=5+4+6
=5+10
=15
Every recursive function must be provided with a way to end the recursion. In this example when, n is equal to 0, there is no recursive call and recursion ends.

Advantages and Disadvantages of Recursion

Recursion is more elegant and requires few variables which make program clean. Recursion can be used to replace complex nesting code by dividing the problem into same problem of its sub-type.

In other hand, it is hard to think the logic of a recursive function. It is also difficult to debug the code containing recursion.

Storage Class

Every variable and function in C programming has two properties: type and storage class. Type refers to the data type of variable whether it is character or integer or floating-point value etc.

There are 4 types of storage class:

1. automatic
2. external
3. static
4. register

Automatic storage class

Keyword for automatic variable

auto
Variables declared inside the function body are automatic by default. These variable are also known as local variables as they are local to the function and doesn't have meaning outside that function
Since, variable inside a function is automatic by default, keyword auto are rarely used.

External storage class

External variable can be accessed by any function. They are also known as global variables. Variables declared outside every function are external variables.

In case of large program, containing more than one file, if the global variable is declared in file 1 and that variable is used in file 2 then, compiler will show error. To solve this problem, keyword extern is used in file 2 to indicate that, the variable specified is global variable and declared in another file.

Example to demonstrate working of external variable

```
#include<stdio.h>
void Check();
int a=5; /* a is global variable because it is outside every function */
int main()
{
a+=4;
Check();
return 0;
}v void Check()
{
++a;
/* ----- Variable a is not declared in this function but, works in any function as they are global variable ------- */
printf("a=%d\n",a);
}
```

Output: a=10

Register Storage Class

Keyword to declare register variable

register

Example of register variable

register int a;

Register variables are similar to automatic variable and exists inside that particular function only.
If the compiler encounters register variable, it tries to store variable in microprocessor's register rather than memory. Value stored in register are much faster than that of memory.

In case of larger program, variables that are used in loops and

function parameters are declared register variables.
Since, there are limited number of register in processor and if it couldn't store the variable in register, it will automatically store it in memory.

Static Storage Class

The value of static variable persists until the end of the program. A variable can be declared static using keyword: static. For example:
static int i;
Here, *i* is a static variable.

Example to demonstrate the static variable

```
#include<stdio.h>
void Check();
int main()
{
Check();
Check();
Check();
}
void Check()
{
static int c=0;
printf("%d\t",c);
c+=5;
}
```

Output

0 5 10

During first function call, it will display 0. Then, during second function call, variable c will not be initialized to 0 again, as it is static variable. So, 5 is displayed in second function call and 10 in third call.
If variable c had been automatic variable, the output would have been:
0 0 0

Arrays

In C programming, one of the frequently arising problem is to handle similar types of data. For example: If the user want to store marks of 100 students. This can be done by creating 100 variable individually but, this process is rather tedious and impracticable. These type of problem can be handled in C programming using arrays.

An array is a sequence of data item of homogeneous value(same type).

Arrays are of two types:

1. One-dimensional arrays
2. Multidimensional arrays

Declaration of one-dimensional array

data_type array_name[array_size];

For example:
int age[5];
Here, the name of array is age. The size of array is 5,i.e., there are 5 items(elements) of array age. All element in an array are of the same type (int, in this case).

Array elements

Size of array defines the number of elements in an array. Each element of array can be accessed and used by user according to the need of program. For example:
int age[5];
Note that, the first element is numbered 0 and so on.

Here, the size of array age is 5 times the size of int because there are 5 elements.
Suppose, the starting addres of age[0] is 2120d and the size of int be 4 bytes. Then, the next address (address of a[1]) will be 2124d, address of a[2] will be 2128d and so on.

Initialization of one-dimensional array:

Arrays can be initialized at declaration time in this source code as:
int age[5]={2,4,34,3,4};

It is not necessary to define the size of arrays during initialization.
int age[]={2,4,34,3,4};

In this case, the compiler determines the size of array by calculating the number of elements of an array.

Accessing array elements

In C programming, arrays can be accessed and treated like variables in C.

For example:

```
scanf("%d",&age[2]);
/* statement to insert value in the third element of array age[]. */

scanf("%d",&age[i]);
/* Statement to insert value in (i+1)th element of array age[]. */
/* Because, the first element of array is age[0], second is age[1], ith is age[i-1] and (i+1)th is age[i]. */

printf("%d",age[0]);
/* statement to print first element of an array. */
```

```
printf("%d",age[i]);
/* statement to print (i+1)th element of an array. */
```

Example of array in C programming

```
/* C program to find the sum marks of n students using arrays */

#include<stdio.h>
int main()
{
int marks[10],i,n,sum=0;
printf("Enter number of students: ");
scanf("%d",&n);
for(i=0;i<n;++i)
{
printf("Enter marks of student%d: ",i+1);
scanf("%d",&marks[i]);
sum+=marks[i];
}
printf("Sum= %d",sum);
return 0;
}
```

Output

```
Enter number of students: 3
Enter marks of student1: 12
Enter marks of student2: 31
Enter marks of student3: 2
sum=45
```

Important thing to remember in C arrays

Suppose, you declared the array of 10 students. For example: arr[10]. You can use array members from arr[0] to arr[9]. But, what if you want to use element arr[10], arr[13] etc. Compiler may not show error using these elements but, may cause fatal error during program execution.

Multidimensional Arrays

C programming language allows to create arrays of arrays known as multidimensional arrays. For example:
float a[2][6];

Here, a is an array of two dimension, which is an example of multidimensional array. This array has 2 rows and 6 columns

For better understanding of multidimensional arrays, array elements of above example can be thinked of as below:

Initialization of Multidimensional Arrays

In C, multidimensional arrays can be initialized in different number of ways.

```
int c[2][3]={{1,3,0}, {-1,5,9}};
OR
int c[][3]={{1,3,0}, {-1,5,9}};
OR
int c[2][3]={1,3,0,-1,5,9};
```

Initialization Of three-dimensional Array

```
double cprogram[3][2][4]={
{{-0.1, 0.22, 0.3, 4.3}, {2.3, 4.7, -0.9, 2}},
{{0.9, 3.6, 4.5, 4}, {1.2, 2.4, 0.22, -1}},
{{8.2, 3.12, 34.2, 0.1}, {2.1, 3.2, 4.3, -2.0}}
};
```

Suppose there is a multidimensional array arr[i][j][k][m]. Then this array can hold i*j*k*m numbers of data.

Similarly, the array of any dimension can be initialized in C programming.

Example of Multidimensional Array In C

Write a C program to find sum of two matrix of order 2*2 using multidimensional arrays where, elements of matrix are entered by user.

```
#include<stdio.h>
int main()
{
float a[2][2], b[2][2], c[2][2];
int i,j;
printf("Enter the elements of 1st matrix\n");
/* Reading two dimensional Array with the help of two for loop. If there was an array of 'n' dimension, 'n' numbers of loops are needed for inserting data to array.*/
for(i=0;i<2;++i)
for(j=0;j<2;++j)
{
printf("Enter a%d%d: ",i+1,j+1);
scanf("%f",&a[i][j]);
}
printf("Enter the elements of 2nd matrix\n");
for(i=0;i<2;++i)
for(j=0;j<2;++j)
{
printf("Enter b%d%d: ",i+1,j+1);
scanf("%f",&b[i][j]);
}
for(i=0;i<2;++i)
for(j=0;j<2;++j)
{
/* Writing the elements of multidimensional array using loop. */
c[i][j]=a[i][j]+b[i][j]; /* Sum of corresponding elements of two arrays. */
}
```

```
printf("\nSum Of Matrix:");
for(i=0;i<2;++i)
for(j=0;j<2;++j)
{
printf("%.1f\t",c[i][j]);
if(j==1) /* To display matrix sum in order. */
printf("\n");
}
return 0;
}
```

Output

```
Enter the elements of 1st matrix
Enter a11: 2;
Enter a12: 0.5;
Enter a21: -1.1;
Enter a22: 2;
Enter the elements of 2nd matrix
Enter b11: 0.2;
Enter b12: 0;
Enter b21: 0.23;
Enter b22: 23;

Sum Of Matrix:
 2.2   0.5
-0.9   25.0
```

Pointers

Pointers are the powerful feature of C and (C++) programming, which differs it from other popular programming languages like: java and Visual Basic.

Pointers are used in C program to access the memory and manipulate the address.

Reference operator(&)

If var is a variable then, &var is the address in memory.

/* Example to demonstrate use of reference operator in C programming. */

```
#include<stdio.h>
int main()
{
int var=5;
printf("Value: %d\n",var);
printf("Address: %d",&var); //Notice, the ampersand(&) before var.
return 0;
}
```

Output

```
Value: 5
Address: 2686778
```

Note: You may obtain different value of address while using this code. In above source code, value 5 is stored in the memory location 2686778. var is just the name given to that location.
You, have already used reference operator in C program while using scanf() function.

scanf("%d",&var);

Reference operator(*) and Pointer variables

Pointers variables are used for taking addresses as values, i.e., a variable that holds address value is called a pointer variable or simply a pointer.

Declaration of Pointer

Dereference operator(*) are used to identify an operator as a pointer.

data_type * pointer_variable_name;v int *p;

Above statement defines, p as pointer variable of type int.

Example To Demonstrate Working of Pointers

```
/* Source code to demonstrate, handling of pointers in C program */
#include<stdio.h>
int main()
{
int *pc,c;
c=22;
printf("Address of c:%d\n",&c);
printf("Value of c:%d\n\n",c);
pc=&c;
printf("Address of pointer pc:%d\n",pc);
printf("Content of pointer pc:%d\n\n",*pc);
c=11;
printf("Address of pointer pc:%d\n",pc);
printf("Content of pointer pc:%d\n\n",*pc);
*pc=2;
printf("Address of c:%d\n",&c);
printf("Value of c:%d\n\n",c);
```

```
return 0;
}
```

Output

```
Address of c: 2686784
Value of c: 22

Address of pointer pc: 2686784
Content of pointer pc: 22

Address of pointer pc: 2686784
Content of pointer pc: 11

Address of c: 2686784
Value of c: 2
```

Explanation of program and figure

1. Code int *pc, p; creates a pointer pc and a variable c. Pointer pc points to some address and that address has garbage value. Similarly, variable c also has garbage value at this point.
2. Code c=22; makes the value of c equal to 22, i.e.,22 is stored in the memory location of variable c.
3. Code pc=&c; makes pointer, point to address of c. Note that, &c is the address of variable c (because c is normal variable) and pc is the address of pc (because pc is the pointer variable). Since the address of pc and address of c is same, *pc (value of pointer pc) will be equal to the value of c.
4. Code c=11; makes the value of c, 11. Since, pointer pc is pointing to address of c. Value of *pc will also be 11.
5. Code *pc=2; change the address pointed by pointer pc to change to 2. Since, address of pointer pc is same as address of c, value of c also changes to 2.

Pointers and Arrays

Arrays are closely related to pointers in C programming. Arrays and pointers are synonymous in terms of how they use to access memory. But, the important difference between them is that, a pointer variable can take different addresses as value whereas, in case of array it is fixed. This can be demonstrated by an example:

```
#include<stdio.h>
int main()
{
char c[4];
int i;
for(i=0;i<4;++i)
{
printf("Address of c[%d]=%x\n",i,&c[i]);
}
return 0;
}
```

Output

```
Address of c[0]=28ff44
Address of c[1]=28ff45
Address of c[2]=28ff46
Address of c[3]=28ff47
```

Notice, that there is equal difference (difference of 1 byte) between any two consecutive elements of array.

Note: You may get different address of an array.

Relation between Arrays and Pointers

Consider an array:

```
int arr[4];
```

In arrays of C programming, name of the array always points to the

first element of an array. Here, address of first element of an array is &arr[0]. Also, arr represents the address of the pointer where it is pointing. Hence, &arr[0] is equivalent to *arr*.

Also, value inside the address &arr[0] and address arr are equal. Value in address &arr[0] is arr[0] and value in address arr is *arr. Hence, arr[0] is equivalent to *arr.

Similarly,
&a[1] is equivalent to (a+1) AND, a[1] is equivalent to *(a+1).
&a[2] is equivalent to (a+2) AND, a[2] is equivalent to *(a+2).
&a[3] is equivalent to (a+1) AND, a[3] is equivalent to *(a+3).
.
&a[i] is equivalent to (a+i) AND, a[i] is equivalent to *(a+i).

In C, you can declare an array and can use pointer to alter the data of an array.

```
//Program to find the sum of six numbers with arrays and pointers.
#include<stdio.h>
int main()
{
int i,class[6],sum=0;
printf("Enter 6 numbers:\n");
for(i=0;i<6;++i)
{
scanf("%d",(class+i)); // (class+i) is equivalent to &class[i]
sum += *(class+i); // *(class+i) is equivalent to class[i]
}
printf("Sum=%d",sum);
return 0;
}
```

Output

```
Enter 6 numbers: 2 3 4 5 3 4
Sum=21
```

Dynamic Memory Allocation

The exact size of array is unknown untill the compile time,i.e., time when a compier compiles code written in a programming language into a executable form. The size of array you have declared initially can be sometimes insufficient and sometimes more than required. Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.

Although, C language inherently does not has any technique to allocated memory dynamically, there are 4 library functions under "stdlib.h" for dynamic memory allocation.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	dellocate the previously allocated space
realloc()	Change the size of previously allocated space

malloc()

The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

Syntax of malloc()

ptr=(cast-type*)malloc(byte-size)

Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

ptr=(int*)malloc(100*sizeof(int));

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

calloc()

The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax of calloc()

ptr=(cast-type*)calloc(n,element-size);

This statement will allocate contiguous space in memory for an array of n elements. For example:

ptr=(float*)calloc(25,sizeof(float));

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

free()

Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

syntax of free()

free(ptr);

This statement cause the space in memory pointer by ptr to be deallocated.

Examples of calloc() and malloc()

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
int n,i,*ptr,sum=0;
printf("Enter number of elements: ");
scanf("%d",&n);
ptr=(int*)malloc(n*sizeof(int)); //memory allocated using malloc
if(ptr==NULL)
{
printf("Error! memory not allocated.");
exit(0);
}
printf("Enter elements of array: ");
for(i=0;i<n;++i)
{
scanf("%d",ptr+i);
sum+=*(ptr+i);
}
printf("Sum=%d",sum);
free(ptr);
return 0;
}
```

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
```

```
int n,i,*ptr,sum=0;
printf("Enter number of elements: ");
scanf("%d",&n);
ptr=(int*)calloc(n,sizeof(int));
if(ptr==NULL)
{
printf("Error! memory not allocated.");
exit(0);
}
printf("Enter elements of array: ");
for(i=0;i<n;++i>)
{
scanf("%d",ptr+i);
sum+=*(ptr+i);
}
printf("Sum=%d",sum);
free(ptr);
return 0;
}
```

realloc()

If the previously allocated memory is insufficient or more than sufficient. Then, you can change memory size previously allocated using realloc().

Syntax of realloc()

```
ptr=realloc(ptr,newsize);
```

Here, ptr is reallocated with size of newsize.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
```

```
int *ptr,i,n1,n2;
printf("Enter size of array: ");
scanf("%d",&n1);
ptr=(int*)malloc(n1*sizeof(int));
printf("Address of previously allocated memory: ");
for(i=0;i<n1;++i)
printf("%u\t",ptr+i);
printf("\nEnter new size of array: ");
scanf("%d",&n2);
ptr=realloc(ptr,n2);
for(i=0;i<n2;++i)
printf("%u\t",ptr+i);
return 0;
}
```

Strings

In C programming, array of character are called strings. A string is terminated by null character /0. For example:

"c string tutorial"

Here, "c string tutorial" is a string. When, compiler encounters strings, it appends null character at the end of string.

Declaration of strings

Strings are declared in C in similar manner as arrays. Only difference is that, strings are of char type.

char s[5];

Strings can also be declared using pointer.

char *p

Initialization of strings

In C, string can be initialized in different number of ways.

```
char c[]="abcd";
OR,
char c[5]="abcd";
OR,
char c[]={'a','b','c','d','\0'};
OR;
char c[5]={'a','b','c','d','\0'};
```

String can also be initialized using pointers

char *c="abcd";
Reading Strings from user. Reading words from user. char c[20];
scanf("%s",c);

String variable c can only take a word. It is beacause when white space is encountered, the scanf() function terminates.

Write a C program to illustrate how to read string from terminal.

```
#include<stdio.h>
int main()
{
char name[20];
printf("Enter name: ");
scanf("%s",name);
printf("Your name is %s.",name);
return 0;
}
```

Output

```
Enter name: Dennis Ritchie
Your name is Dennis.
```

Here, program will ignore Ritchie because, scanf() function takes only string before the white space.

Reading a line of text

C program to read line of text manually.

```
#include<stdio.h>
int main()
{
char name[30],ch;
int i=0;
printf("Enter name: ");
```

```
while(ch!='\n') // terminates if user hit enter
{
ch=getchar();
name[i]=ch;
i++;
}
name[i]='\0'; // inserting null character at end
printf("Name: %s",name);
return 0;
}
```

This process to take string is tedious. There are predefined functions gets() and puts in C language to read and display string respectively.

```
int main()
{
char name[30];
printf("Enter name: ");
gets(name); //Function to read string from user.
printf("Name: ");
puts(name); //Function to display string.
return 0;
}
```

Both, the above program has same output below:

Output

```
Enter name: Tom Hanks
Name: Tom Hanks
```

Passing Strings to Functions

String can be passed to function in similar manner as arrays as, string is also an array.

```
#include<stdio.h>
void Display(char ch[]);
int main()
{
char c[50];
printf("Enter string: ");
gets(c);
Display(c); // Passing string c to function.
return 0;
}
void Display(char ch[])
{
printf("String Output: ");
puts(ch);
}
```

Here, string c is passed from main() function to user-defined function Display(). In function declaration, ch[] is the formal argument.

String handling functions

You can perform different type of string operations manually like: finding length of string, concatenating(joining) two strings etc. But, for programmers ease, many library function are defined under header file to handle these commonly used talk in C programming. You will learn more about string hadling function in next chapter.

Strings Functions

Strings are often needed to be manipulated by programmer according to the need of a problem. All string manipulation can be done manually by the programmer but, this makes programming complex and large. To solve this, the C supports a large number of string handling functions.There are numerous functions defined in *"string.h"* header file. Few commonly used string handling functions are discussed below:

Function	Work of Function
strlen()	Calculates the length of string
strcpy()	Copies a string to another string
strcat()	Concatenates(joins) two strings
strcmp()	Compares two string
strlwr()	Converts string to lowercase
strupr()	Converts string to uppercase

Strings handling functions are defined under "string.h" header file, i.e, you have to include the code below to run string handling functions.

#include<*string.h*>

gets() and puts(): Functions gets() and puts() are two string functions to take string input from user and display string respectively as mentioned in previous chapter.

```
#include<stdio.h>
int main(){
char name[30];
printf("Enter name: ");
gets(name);       //Function to read string from user.
printf("Name: ");
puts(name);       //Function to display string.
return 0;}
```

Though, gets() and puts() function handle string, both these functions are defined in "stdio.h" header file.

Structure

Structure is the collection of variables of different types under a single name for better handling. For example: You want to store the information about person about his/her name, citizenship number and salary. You can create these information separately but, better approach will be collection of these information under single name because all these information are related to person.

Structure Definition in C

Keyword *struct* is used for creating a structure.

Syntax of structure

```
struct structure_name
{
data_type member1;
data_type member2;
.
.
data_type memeber;
};
```

We can create the structure for a person as mentioned above as:

```
struct person
{
char name[50];
int cit_no;
float salary;
};
```

This declaration above creates the derived data type struct person.

Structure variable declaration

When a structure is defined, it creates a user-defined type but, no storage is allocated. For the above structure of person, variable can be declared as:

```
struct person
{
char name[50];
int cit_no;
float salary;
};
```

Inside main function:
struct person p1, p2, p[20];

Another way of creating sturcture variable is:

```
struct person
{
char name[50];
int cit_no;
float salary;
}p1 ,p2 ,p[20];
```

In both cases, 2 variables p1, p2 and array p having 20 elements of type struct person are created.

Accessing members of a structure

There are two types of operators used for accessing members of a structure.

- Member operator(.)
- Structure pointer operator(->)

Any member of a structure can be accessed as: *structure_variable_name.member_name*

Suppose, we want to access salary for variable p2. Then, it can be accessed as:

p2.salary

Example of structure

Write a C program to add two distances entered by user. Measurement of distance should be in inch and feet.(Note: 12 inches = 1 foot)

```
#include<stdio.h>
struct Distance
{
int feet;
float inch;
}d1,d2,sum;
int main()
{
printf("1st distance\n");
printf("Enter feet: ");
scanf("%d",&d1.feet); /* input of feet for structure variable d1 */
printf("Enter inch: ");
scanf("%f",&d1.inch); /* input of inch for structure variable d1 */
printf("2nd distance\n");
printf("Enter feet: ");
scanf("%d",&d2.feet); /* input of feet for structure variable d2 */
printf("Enter inch: ");
scanf("%f",&d2.inch); /* input of inch for structure variable d2 */
sum.feet=d1.feet+d2.feet;
sum.inch=d1.inch+d2.inch;
```

```
if (sum.inch>12)
{ //If inch is greater than 12, changing it to feet.
++sum.feet;
sum.inch=sum.inch-12;
}
printf("Sum of distances=%d\'-%.1f\"",sum.feet,sum.inch);
/* printing sum of distance d1 and d2 */
return 0;
}
```

Output

```
1st distance
Enter feet: 12
Enter inch: 7.9
2nd distance
Enter feet: 2
Enter inch: 9.8
Sum of distances= 15'-5.7"
```

Keyword typedef while using structure

Programmer generally use typedef while using structure in C language. For example:

```
typedef struct complex
{
int imag;
float real;
}comp;

Inside main:
comp c1,c2;
```

Here, *typedef* keyword is used in creating a type comp(which is of

type as struct complex). Then, two structure variables c1 and c2 are created by this comp type.

Structures within structures

Structures can be nested within other structures in C programming.

```
struct complex
{
int imag_value;
float real_value;
};
struct number
{
struct complex c1;
int real;
}n1,n2;
```

Suppose you want to access imag_value for n2 structure variable then, structure member n1.c1.imag_value is used.

Structure and Pointers

Pointers can be accessed along with structures. A pointer variable of structure can be created as below:

```
struct name
{
member1;
member2;
.
.
};
-------- Inside function -------
struct name *ptr;
```

Here, the pointer variable of type struct name is created.

Structure's member through pointer can be used in two ways:

- Referencing pointer to another address to access memory
- Using dynamic memory allocation

Consider an example to access structure's member through pointer.

```
#include<stdio.h>
struct name
{
int a;
float b;
};
int main()
{
struct name *ptr,p;
```

```
ptr=&p; /* Referencing pointer to memory address of p */
printf("Enter integer: ");
scanf("%d",&(*ptr).a);
printf("Enter number: ");
scanf("%f",&(*ptr).b);
printf("Displaying: ");
printf("%d%f",(*ptr).a,(*ptr).b);
return 0;
}
```

In this example, the pointer variable of type struct name is referenced to the address of p. Then, only the structure member through pointer can can accessed.

Structure pointer member can also be accessed using -> operator.

(*ptr).a is same as ptr->a
(*ptr).b is same as ptr->b

Accessing structure member through pointer using dynamic memory allocation

To access structure member using pointers, memory can be allocated dynamically using malloc() function defined under "stdlib.h" header file.

Syntax to use malloc()

```
ptr=(cast-type*)malloc(byte-size)
```

Example to use structure's member through pointer using malloc() function.

```
#include<stdio.h>
#include<stdlib.h>
struct name
{
int a;
float b;
```

```
char c[30];
};
int main()
{
struct name *ptr;
int i,n;
printf("Enter n: ");
scanf("%d",&n);
ptr=(struct name*)malloc(n*sizeof(struct name));
/* Above statement allocates the memory for n structures with pointer
ptr pointing to base address */
for(i=0;i<n;++ii>)
{
printf("Enter string, integer and floating number respectively:\n");
scanf("%s%d%f",&(ptr+i)->c,&(ptr+i)->a,&(ptr+i)->b);
}
printf("Displaying Infromation:\n");
for(i=0;i<n;++i)
printf("%s\t%d\t%.2f\n",(ptr+i)->c,(ptr+i)->a,(ptr+i)->b);
return 0;
}
```

Output

```
Enter n: 2
Enter string, integer and floating number respectively:
Programming
2
3.2
Enter string, integer and floating number respectively:
Structure
6
2.3
Displaying Information
Programming 2 3.20
Structure 6 2.30
```

Structure and Functions

In C, structure can be passed to functions by two methods:

- Passing by value (passing actual value as argument)
- Passing by reference (passing address of an argument)

Passing structure by value

A structure variable can be passed to the function as an argument as normal variable. If structure is passed by value, change made in structure variable in function definition does not reflect in original structure variable in calling function.

Write a C program to create a structure student, containing name and roll. Ask user the name and roll of a student in main function. Pass this structure to a function and display the information in that function.

```
#include<stdio.h>
struct student
{
char name[50];
int roll;
};
void Display(struct student stu);
/* function prototype should be below to the structure declaration
otherwise compiler shows error */
int main()
{
struct student s1;
printf("Enter student's name: ");
scanf("%s",&s1.name);
printf("Enter roll number:");
scanf("%d",&s1.roll);
```

```
Display(s1); // passing structure variable s1 as argument
return 0;
}
void Display(struct student stu)
{
printf("Output\nName: %s",stu.name);
printf("\nRoll: %d",stu.roll);
}
```

Output

Enter student's name: Kevin Amla
Enter roll number: 149

Output

Name: Kevin Amla
Roll: 149

Passing structure by reference

The address location of structure variable is passed to function while passing it by reference. If structure is passed by reference, change made in structure variable in function definition reflects in original structure variable in the calling function.

Write a C program to add two distances(feet-inch system) entered by user. To solve this program, make a structure. Pass two structure variable (containing distance in feet and inch) to add function by reference and display the result in main function without returning it.

```
#include<stdio.h>
struct distance
{
int feet;
float inch;
};
void Add(struct distance d1,struct distance d2, struct distance *d3);
int main()
```

```
{
struct distance dist1, dist2, dist3;
printf("First distance\n");
printf("Enter feet: ");
scanf("%d",&dist1.feet);
printf("Enter inch: ");
scanf("%f",&dist1.inch);
printf("Second distance\n");
printf("Enter feet: ");
scanf("%d",&dist2.feet);
printf("Enter inch: ");
scanf("%f",&dist2.inch);
Add(dist1, dist2, &dist3);
/*passing structure variables dist1 and dist2 by value whereas passing
structure variable dist3 by reference */
printf("\nSum of distances = %d\'-%.1f\"",dist3.feet, dist3.inch);
return 0;
}
void Add(struct distance d1,struct distance d2, struct distance *d3)
{
/* Adding distances d1 and d2 and storing it in d3 */
d3->feet=d1.feet+d2.feet;
d3->inch=d1.inch+d2.inch;
if (d3->inch>=12)
{ /* if inch is greater or equal to 12, converting it to feet. */
d3->inch-=12;
++d3->feet;
}
}
```

Output

```
First distance
Enter feet: 12
Enter inch: 6.8
```

Second distance
Enter feet: 5
Enter inch: 7.5

Sum of distances = 18'-2.3"

Explaination

In this program, structure variables dist1 and dist2 are passed by value (because value of dist1 and dist2 does not need to be displayed in main function) and dist3 is passed by reference ,i.e, address of dist3 (&dist3) is passed as an argument. Thus, the structure pointer variable d3 points to the address of dist3. If any change is made in d3 variable, effect of it is seed in dist3 variable in main function

Unions

Unions are quite similar to the. Union is also a derived type as structure. Union can be defined in same manner as structures just the keyword used in defining union in union where keyword used in defining structure was struct.

```
union car
{
char name[50];
int price;
};
```

Union variables can be created in similar manner as structure variable.

```
union car
{
char name[50];
int price;
}c1, c2, *c3;
```

OR;

```
union car
{
char name[50];
int price;
};
-------Inside Function----------
union car c1, c2, *c3;
```

In both cases, union variables c1, c2 and union pointer variable c3 of type union car is created.

Accessing members of an union

The member of unions can be accessed in similar manner as that structure. Suppose, we you want to access price for union variable c1 in above example, it can be accessed as c1.price. If you want to access price for union pointer variable c3, it can be accessed as (*c3).price or as c3->price.

Difference between union and structure

Though unions are similar to structure in so many ways, the difference between them is crucial to understand. This can be demonstrated by this example:

```
#include<stdio.h>
union job
{ //defining a union
char name[32];
float salary;
int worker_no;
}u;
struct job1
{
char name[32];
float salary;
int worker_no;
}s;
int main()
{
printf("size of union = %d",sizeof(u));
printf("\nsize of structure = %d", sizeof(s));
return 0;
}
```

Output

size of union = 32
size of structure = 40

There is difference in memory allocation between union and structure as suggested in above example. The amount of memory required to store a structure variables is the sum of memory size of all members.

But, the memory required to store a union variable is the memory required for largest element of an union.

What difference does it make between structure and union?

As you know, all members of structure can be accessed at any time. But, only one member of union can be accessed at a time in case of union and other members will contain garbage value.

```
#include<stdio.h>
union job
{
char name[32];
float salary;
int worker_no;
}u;
int main()
{
printf("Enter name:\n");
scanf("%s",&u.name);
printf("Enter salary: \n");
scanf("%f",&u.salary);
printf("Displaying\nName :%s\n",u.name);
printf("Salary: %.1f",u.salary);
return 0;
}
```

Output

```
Enter name
Hillary
Enter salary
1234.23
Displaying
Name: f%Bary
Salary: 1234.2
```

Note: You may get different garbage value of name

Why this output?

Initially, Hillary will be stored in u.name and other members of union will contain garbage value. But when user enters value of salary, 1234.23 will be stored in u.salary and other members will contain garbage value. Thus in output, salary is printed accurately but, name displays some random string.

Passing Union to a Function

Union can be passed in similar manner as structures in C programming.