Unit I – Basic of C Programming

Fundamental of algorithms:

Notion of an algorithm: Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms −

Search: Algorithm to search an item in a data structure.

Sort: Algorithm to sort items in a certain order**.**

Insert: Algorithm to insert item in a data structure.

Update: Algorithm to update an existing item in a data structure.

Delete: Algorithm to delete an existing item from a data structure.

Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics −

Unambiguous: Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.

Input: An algorithm should have 0 or more well-defined inputs.

Output: An algorithm should have 1 or more well-defined outputs, and should match the desired output.

Finiteness: Algorithms must terminate after a finite number of steps.

Feasibility: Should be feasible with the available resources.

Independent: An algorithm should have step-by-step directions, which should be independent of any programming code.

How to Write an Algorithm?

There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

Example

Let's try to learn algorithm-writing by using an example.

Problem − Design an algorithm to add two numbers and display the result.

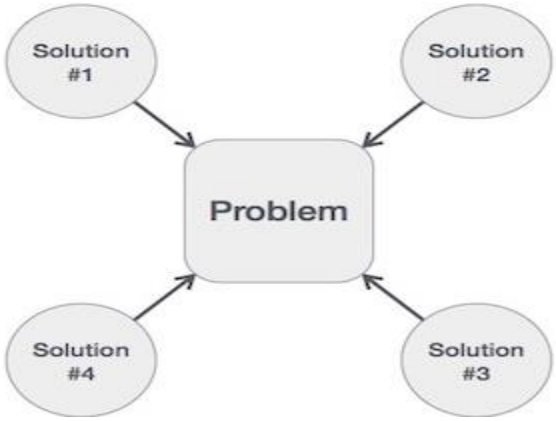
Step 1 − START
Step 2 − Declare three integers **a**, **b** & **c**
Step 3 − Define values of **a** & **b**
Step 4 − Add values of **a** & **b**
Step 5 − Store output of step 4 to **c**
Step 6 − Print **c**
Step 7 − STOP

Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as −

Step 1 − START ADD
Step 2 − Get values of **a** & **b**
Step 3 − c ← a + b
Step 4 − Display c
Step 5 − STOP

In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.

Writing **step numbers**, is optional. We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways.



Hence, many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.

Notion is used for Basic Control Structures:

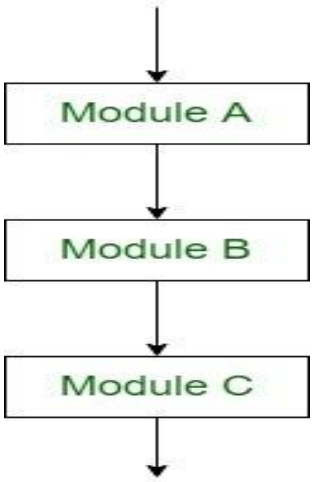
Control Structures are just a way to specify flow of control in programs. Any algorithm or program can be clearer and more understood if they use self-contained modules called as logic or control structures. It basically analyzes and chooses in which direction a program flows based on certain parameters or conditions.

There are three basic types of logic, or flow of control, known as:

1. Sequence logic, or sequential flow
2. Selection logic, or conditional flow
3. Iteration logic, or repetitive flow

1. Sequential Logic (Sequential Flow):

Sequential logic as the name suggests follows a serial or sequential flow in which the flow depends on the series of instructions given to the computer. Unless new instructions are given, the modules are executed in the obvious sequence. The sequences may be given, by means of numbered steps explicitly. Also, implicitly follows the order in which modules are written. Most of the processing, even some complex problems, will generally follow this elementary flow pattern.



Sequential Control flow

2. Selection Logic (Conditional Flow):

Selection Logic simply involves a number of conditions or parameters which decides one out of several written modules. The structures which use this type of logic are known as Conditional Structures.

These structures can be of three types:

Single Alternative: This structure has the form:

```
If (condition) then:
    [Module A]
[End of If structure]
```

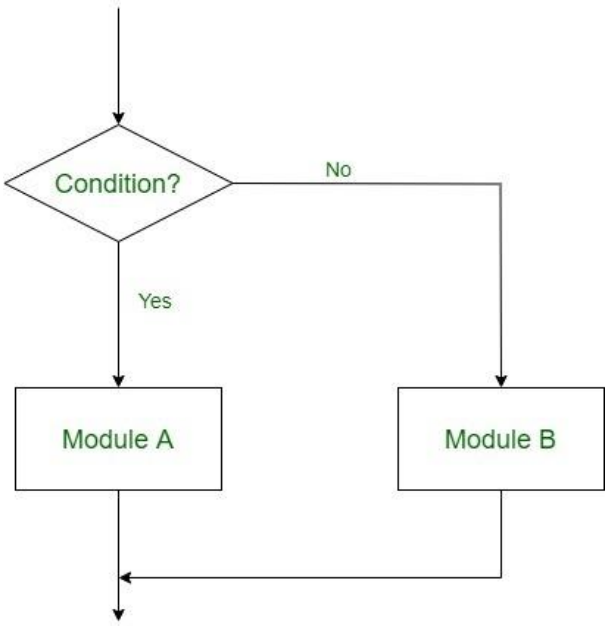
Double Alternative: This structure has the form:

```
If (Condition), then:
    [Module A]
Else:
    [Module B]
[End if structure]
```

Multiple Alternatives: This structure has the form:

```
If (condition A), then:
    [Module A]
Else if (condition B), then:
    [Module B]
      ..
      ..
Else if (condition N), then:
    [Module N]
[End If structure]
```

In this way, the flow of the program depends on the set of conditions that are written. This can be more understood by the following flow charts:



Double Alternative Control Flow

3. Iteration Logic (Repetitive Flow)

The Iteration logic employs a loop which involves a repeat statement followed by a module known as the body of a loop.

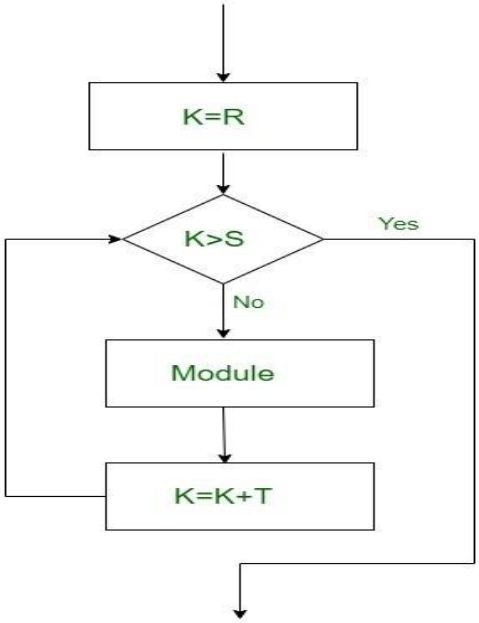
The two types of these structures are:

Repeat-For Structure

This structure has the form:

Repeat for i = A to N by I:
    [Module]
[End of loop]

Here, A is the initial value, N is the end value and I is the increment. The loop ends when A>B. K increases or decreases according to the positive and negative value of I respectively.

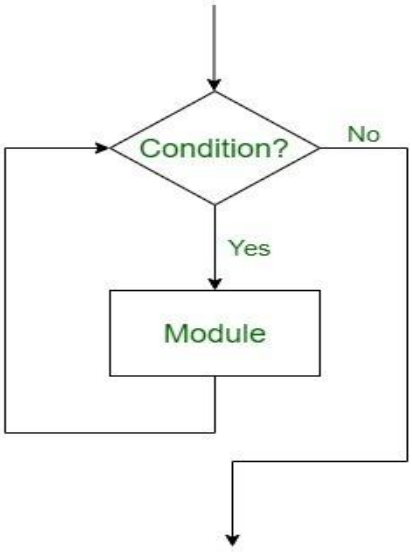


Repeat-For Flow

Repeat-While Structure

It also uses a condition to control the loop. This structure has the form:

Repeat while condition:
    [Module]
[End of Loop]



Repeat While Flow

Algorithmic Problems:

Develop fundamental algorithm to solve simple problem such as:

1) Solve simple arithmetic expression

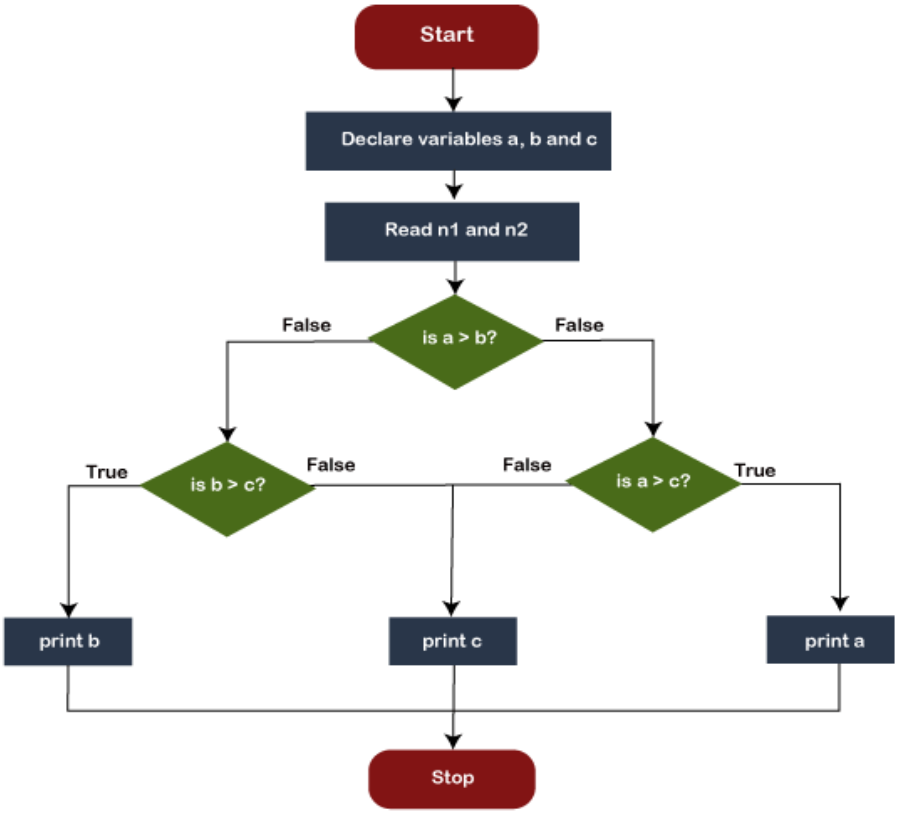
Rules for Arithmetic Expression Evaluation:

1. Solve the Parenthesis first. In other words, firstly solve the sub-expressions inside the parenthesis. ...
2. If an expression or sub-expression does not contain parenthesis, the expressions are solved according to the precedence of the operators.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a,b,c,d,e,f,g,result;
    printf("enter value of a : ");
    scanf("%d",& a);
    printf("enter value of b : ");
    scanf("%d",& b);
    printf("enter value of c : ");
    scanf("%d",& c);
    printf("enter value of d : ");
    scanf("%d",& d);
    printf("enter value of e : ");
    scanf("%d",& e);
    printf("enter value of f : ");
    scanf("%d",& f);
```

```
    printf("enter value of g : ");
    scanf("%d",& g);
    result=((a -(((b / c) * d) + e)) * (f +g));
    printf("after evaluation result is :%d ",result);
}
```

2) Find greatest of three number



Algorithm to find greatest number of three given numbers

1. Ask the user to enter three integer values.
2. Read the three integer values in num1, num2, and num3 (integer variables).
3. Check if num1 is greater than num2.
4. If true, then check if num1 is greater than num3. ...
5. If false, then check if num2 is greater than num3.

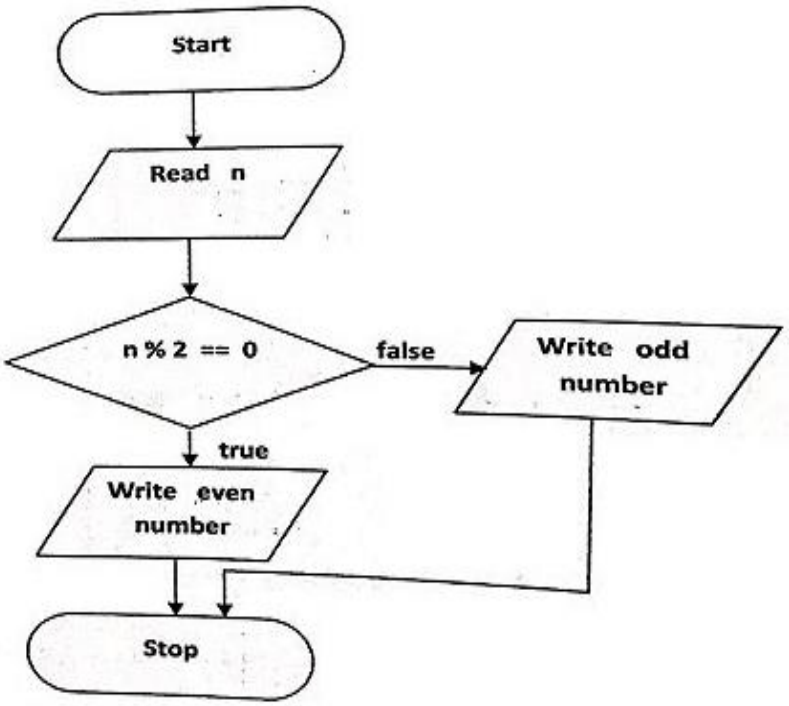
```
#include <stdio.h>
int main()
{
    int A, B, C;
      printf("Enter the numbers A, B and C: ");
    scanf("%d %d %d", &A, &B, &C);
      if (A >= B && A >= C)
        printf("%d is the largest number.", A);
     if (B >= A && B >= C)
```

```
        printf("%d is the largest number.", B);
    if (C >= A && C >= B)
        printf("%d is the largest number.", C);
      return 0;
}
```

3) Determine whether a given no is even or odd

Algorithm

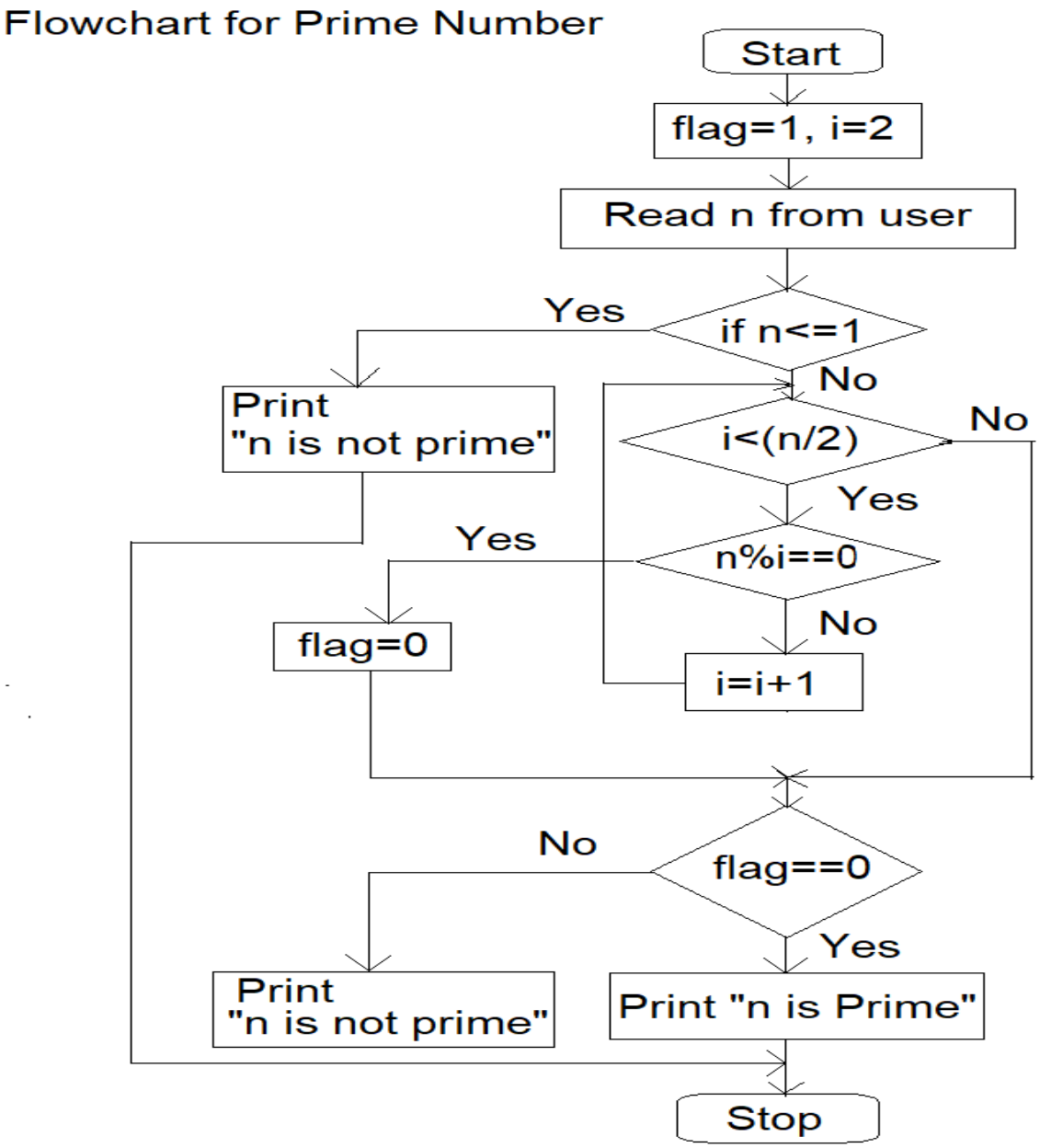
Step 1: Start
Step 2: [Take Input] Read: Number]
Step 3: Check: If Number%2 == 0 Then
Print : N is an Even Number. Else
Print : N is an Odd Number.
Step 4: Exit




```
#include <stdio.h>
int main()
{
    int number;
    printf("Enter an integer: ");
    scanf("%d", &number);
    // True if the number is perfectly divisible by 2
    if(number % 2 == 0)
        printf("%d is even.", number);
    else
        printf("%d is odd.", number);
     return 0;
```

}

3) Determine whether a given no is prime



Algorithm

Step 1: Start
Step 2: Read number n
Step 3: Set f=0
Step 4: For i=2 to n-1
Step 5: If n mod 1=0 then
Step 6: Set f=1 and break
Step 7: Loop
Step 8: If f=0 then

print 'The given number is prime'

else

print 'The given number is not prime'

Step 9: Stop

```
#include<stdio.h>
#include<conio.h>
void main( )
{
clrscr();
int n,i,f=0;
printf("Enter the number: ");
scanf("%d",&n);
for(i=2;i<n;i++)
{
  if(n%i==0)
   {
     f=1;
     break;
   }
}
if(f==0)
   printf("The given number is prime");
else
   printf("The given number is not prime");
getch();
}
```

Introduction of C Language

C is a programming language developed at AT & T's Bell Laboratories of USA in 1972. It was designed and written by a man named Dennis Ritchie.

Possibly why C seems so popular is because it is reliable, simple and easy to use.

Construct of C Program:

Documentation Section

```
# include Section
Preprocessor Section
Global Declaration
Main Function
{
     Local Declaration
     Executable Code
     -
     -
     -
     -
```

```
      –
}
Function 1
Function 2
–
–
–
–
Function n
```

The sections of a C program are listed below:

1. Documentation section
2. Preprocessor section
3. Definition section
4. Global declaration
5. Main function
6. User defined functions

Documentation section:

It includes the statement specified at the beginning of a program, such as a program's name, date, description, **and** title**. It is represented as:**

```
//name of a program
```

Or

```
/*
Overview of the code
.
*/
```

Both methods work as the document section in a program. It provides an overview of the program. Anything written inside will be considered a part of the documentation section and will not interfere with the specified code.

Preprocessor section:

The preprocessor section contains all the header files used in a program. It informs the system to link the header files to the system libraries.

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

The **#include** statement includes the specific file as a part of a function at the time of the compilation. Thus, the contents of the included file are compiled along with the function being compiled.

The **#include<stdio.h>** consists of the contents of the standard input output files, which contains the definition of stdin, stdout, and stderr. Whenever the definitions stdin, stdout, and stderr are used in a function, the statement #include<stdio.h> need to be used.

There are various header files available for different purposes. For example, **# include <math.h>.** It is used for mathematic functions in a program.

Define section:

The define section comprises of different constants declared using the define keyword. It is given by:

```
#define a = 2
```

Global declaration:

The global section comprises of all the global declarations in the program.

```
float num = 2.54;
int a = 5;
char ch ='z';
```

The size of the above global variables is listed as follows:

char = 1 byte
float = 4 bytes
int = 4 bytes

We can also declare user defined functions in the global variable section.

Main function:

main( ) is the first function to be executed by the computer. It is necessary for a code to include the main( ). It is like any other function available in the C library. Parenthesis ( ) are used for passing parameters (if any) to a function.

The main function is declared as:

main( )

We can also use int or main with the main ( ). The void main( ) specifies that the program will not return any value. The int main( ) specifies that the program can return integer type data.

```
int main()
Or
void main()
```

Main function is further categorized into local declarations, statements, and expressions.

Local declarations:

The variable that is declared inside a given function or block refers to as local declarations.

```
main()
{
      int i = 2;
      i++;
}
```

Statements:

The statements refer to **if, else, while, do, for**, etc. used in a program within the main function.

Expressions:

An expression is a type of formula where operands are linked with each other by the use of operators.

```
a - b;
a +b;
```

User defined functions:

The user defined functions specified the functions specified as per the requirements of the user. For example, color( ), sum( ), division( ), etc.

The program (basic or advance) follows the same sections as listed above.

Return function:

It is generally the last section of a code but, it is not necessary to include. It is used when we want to return a value. The return function returns a value when the return type other than the void is specified with the function.

Return type ends the execution of the function. It further returns control to the specified calling function.

return;
Or

return expression;
For example,
return 0;

Examples:

Let's begin with a simple program in C language.

Example 1: To find the sum of two numbers given by the user.

```
/* Sum of two numbers */
#include<stdio.h>
int main()
```

```
{
      int a, b, sum;
      printf("Enter two numbers to be added ");
      scanf("%d %d", &a, &b);
      // calculating sum
      sum = a + b;
      printf("%d + %d = %d", a, b, sum);
      return 0;  // return the integer value in the sum
}
```

Output:

```
Enter two numbers to be added 3 5
3 + 5 = 8
```

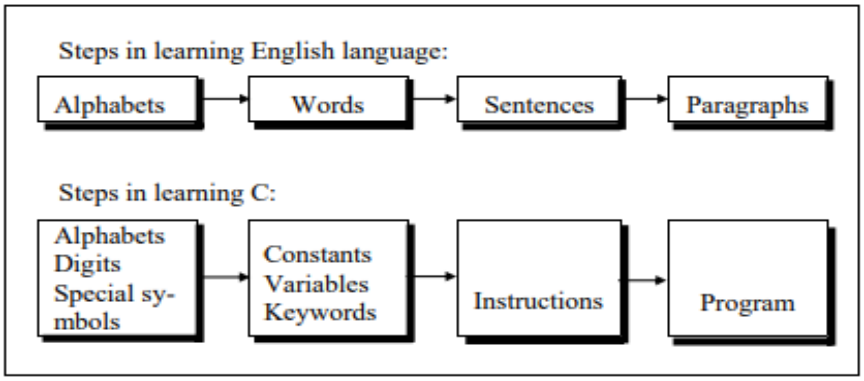
The detailed explanation of each part of a code is as follows:

/* Sum of the two numbers */	It is the comment section. Any statement described in it is not considered as a code. It is a part of the description section in a code. The comment line is optional. It can be in a separate line or part of an executable line.
#include<stdio.h>	It is the standard input-output header file. It is a command of the preprocessor section.
int main( )	main( ) is the first function to be executed in every program. We have used int with the main( ) in order to return an integer value.
{… }	The curly braces mark the beginning and end of a function. It is mandatory in all the functions.
printf( )	The printf( ) prints text on the screen. It is a function for displaying constant or variables data. Here, 'Enter two numbers to be added' is the parameter passed to it.
scanf( )	It reads data from the standard input stream and writes the result into the specified arguments.
sum = a + b	The addition of the specified two numbers will be passed to the sum parameter in the output.
return 0	A program can also run without a return 0 function. It simply states that a program is free from error and can be successfully exicuted.

Getting Started with C

Communicating with a computer involves speaking the language the computer understands, which immediately rules out English as the language of communication with computer. However, there is a close analogy between learning English language and learning C language. The classical method of learning English is to first learn the alphabets used in the language, then learn to combine these alphabets to form words, which in turn are combined to form sentences and sentences are combined to form paragraphs. Learning C is similar and easier. Instead of straight-away learning how to write programs, we must first know what alphabets, numbers and special symbols are used in C, then how

using them constants, variables and keywords are constructed, and finally how are these combined to form an instruction. A group of instructions would be combined later on to form a program



The C Character Set:

A character denotes any alphabet, digit or special symbol used to represent information.

Alphabets	A, B, ….., Y, Z a, b, ……, y, z
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special symbols	~ ‘ ! @ # % ^ & * ( ) _ - + = | \ { } [ ] : ; " ' < > , . ? /

Tokens:

A token in C can be defined as the smallest individual element of the C programming language that is meaningful to the compiler. It is the basic component of a C program.

Types of Tokens in C

The tokens of C language can be classified into six types based on the functions they are used to perform. The types of C tokens are as follows:

Keywords

Keywords are the words whose meaning has already been explained to the C compiler (or in a broad sense to the computer). The keywords cannot be used as variable names because if we do so we are trying to assign a new meaning to the keyword, which is not allowed by the computer. Some C compilers allow you to construct variable names that exactly resemble the keywords. However, it would be safer not to mix up the variable names and the keywords. The keywords are also called 'Reserved words.

There are only 32 keywords available in C.

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

Note that compiler vendors (like Microsoft, Borland, etc.) provide their own keywords apart from the ones mentioned above. These include extended keywords like near, far, asm, etc.

Identifiers in C:

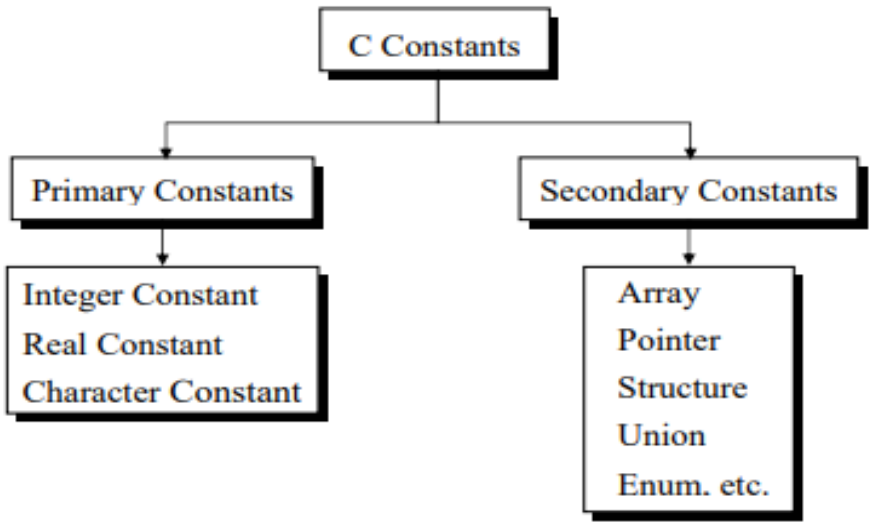
Identifiers in C are used for naming variables, functions, arrays, structures, etc. Identifiers in C are the user-defined words. It can be composed of uppercase letters, lowercase letters, underscore, or digits, but the starting letter should be either an underscore or an alphabet. Identifiers cannot be used as keywords. Rules for constructing identifiers in C are given below:

- The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the character, digit, or underscore.
- It should not begin with any numerical digit.
- In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive.
- Commas or blank spaces cannot be specified within an identifier.
- Keywords cannot be represented as an identifier.
- The length of the identifiers should not be more than 31 characters.
- Identifiers should be written in such a way that it is meaningful, short, and easy to read.

Constants

Constants in the C programming language represent values that do not change during the execution of a program. They play a crucial role in enhancing code readability, maintainability, and program robustness. Understanding the use of constants provides programmers with a powerful mechanism to create reliable and easily adaptable code.

Types of Constants:



Rules for Constructing Integer Constants:

- An integer constant must have at least one digit.
- It must not have a decimal point.
- It can be either positive or negative.
- If no sign precedes an integer constant it is assumed to be positive
- No commas or blanks are allowed within an integer constant.
- The allowable range for integer constants is -32768 to 32767.

Truly speaking the range of an Integer constant depends upon the compiler. For a 16-bit compiler like Turbo C or Turbo C++ the range is –32768 to 32767. For a 32-bit compiler the range would be even greater.

Rules for Constructing Real Constants

Real constants are often called Floating Point constants. The real constants could be written in two forms—Fractional form and Exponential form.

Following rules must be observed while constructing real constants expressed in fractional form:

- A real constant must have at least one digit.
- It must have a decimal point.
- It could be either positive or negative.
- Default sign is positive.
- No commas or blanks are allowed within a real constant.

Rules for Constructing Character Constants:

- A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas. Both the inverted commas should point to the left. For example, ’A’ is a valid character constant whereas ‘A’ is not.
- The maximum length of a character constant can be 1 character. e.g.: 'A' 'I' '5' '='

Variable:

an entity that may vary during program execution is called a variable. Variable names are names given to locations in memory.

Types of C Variables:

Variable names are names given to locations in memory. These locations can contain integer, real or character constants. In any language, the types of variables that it can support depend on the types of constants that it can handle. This is because a particular type of variable can hold only the same type of constant. For example, an integer variable can hold only an integer constant, a real variable can hold only a real constant and a character variable can hold only a character constant.

The rules for constructing different types of constants are different. However, for constructing variable names of all types the same set of rules apply.

These rules are given below:

Rules for Constructing Variable Names:

- A variable name is any combination of 1 to 31 alphabets, digits or underscores. Some compilers allow variable names whose length could be up to 247 characters. Still, it would be safer to stick to the rule of 31 characters. Do not create unnecessarily long variable names as it adds to your typing effort.

- The first character in the variable name must be an alphabet or underscore.
- No commas or blanks are allowed within a variable name.
- No special symbol other than an underscore (as in gross_sal) can be used in a variable name. Ex.: si_int m_hra pop_e_89

C compiler is able to distinguish between the variable names by making it compulsory for you to declare the type of any variable name that you wish to use in a program. This type declaration is done at the beginning of the program.

Data Type Conversion in C:

Converting one datatype into another is known as type casting or, type-conversion. For example, if you want to store a 'long' value into a simple integer then you can type cast 'long' to 'int'. You can convert the values from one type to another explicitly using the cast operator as follows –

(type_name) expression

Let's see an example

```c
#include <stdio.h>

int main() {

  int number = 34.78;

  printf("%d", number);

  return 0;
}

// Output: 34
```

Here, we are assigning the double value **34.78** to the integer variable number. In this case, the double value is automatically converted to integer value **34**.

This type of conversion is known as implicit type conversion. In C, there are two types of type conversion:

1. Implicit Conversion
2. Explicit Conversion

Implicit Type Conversion In C

As mentioned earlier, in implicit type conversion, the value of one type is automatically converted to the value of another type. For example,

```c
#include<stdio.h>

int main() {

  // create a double variable
  double value = 4150.12;
  printf("Double Value: %2f\n", value);

  // convert double value to integer
  int number = value;

  printf("Integer Value: %d", number);

  return 0;
}
```

Output

```
Double Value: 4150.12
Integer Value: 4150
```

The above example has a double variable with a value **4150.12**. Notice that we have assigned the double value to an integer variable.

```c
int number = value;
```

Here, the C compiler automatically converts the double value 4150.12 to integer value 4150.

Since the conversion is happening automatically, this type of conversion is called implicit type conversion.

Example: Implicit Type Conversion

```c
int main() {

  // character variable
  char alphabet = 'a';
  printf("Character Value: %c\n", alphabet);

  // assign character value to integer variable
```

```
  int number = alphabet;

  printf("Integer Value: %d", number);

  return 0;
}
```

Output

```
Character Value: a
Integer Value: 97
```

The code above has created a character variable `alphabet` with the value `'a'`. Notice that we are assigning `alphabet` to an integer variable.

```
int number = alphabet;
```

Here, the C compiler automatically converts the character 'a' to integer **97**. This is because, in C programming, characters are internally stored as integer values known as **ASCII Values**.

ASCII defines a set of characters for encoding text in computers. In ASCII code, the character 'a' has integer value **97**, that's why the character 'a' is automatically converted to integer **97**.

Explicit Type Conversion In C

In explicit type conversion, we manually convert values of one data type to another type. For example,

```
#include<stdio.h>

int main() {

  // create an integer variable
  int number = 35;
  printf("Integer Value: %d\n", number);

  // explicit type conversion
  double value = (double) number;

  printf("Double Value: %.2lf", value);

  return 0;
}
```

Output

```
Integer Value: 35
Double Value: 35.00
```

We have created an integer variable named `number` with the value **35** in the above program. Notice the code,

```
// explicit type conversion
double value = (double) number;
```

Here,

- `(double)` - represents the data type to which `number` is to be converted
- `number` - value that is to be converted to `double` type

Example: Explicit Type Conversion

```
#include<stdio.h>

int main() {

  // create an integer variable
  int number = 97;
  printf("Integer Value: %d\n", number);

  // (char) converts number to character
  char alphabet = (char) number;

  printf("Character Value: %c", alphabet);

  return 0;
}
```

Output

```
Integer Value: 97
Character Value: a
```

We have created a variable `number` with the value **97** in the code above. Notice that we are converting this integer to character.

```
char alphabet = (char) number;
```

Here,

- (char) - explicitly converts number into character
- number - value that is to be converted to char type

Data Loss In Type Conversion

In our earlier examples, when we converted a double type value to an integer type, the data after decimal was lost.

```
#include<stdio.h>

int main() {

  // create a double variable
  double value = 4150.12;
  printf("Double Value: %.2lf\n", value);

  // convert double value to integer
  int number = value;

  printf("Integer Value: %d", number);

  return 0;
}
```

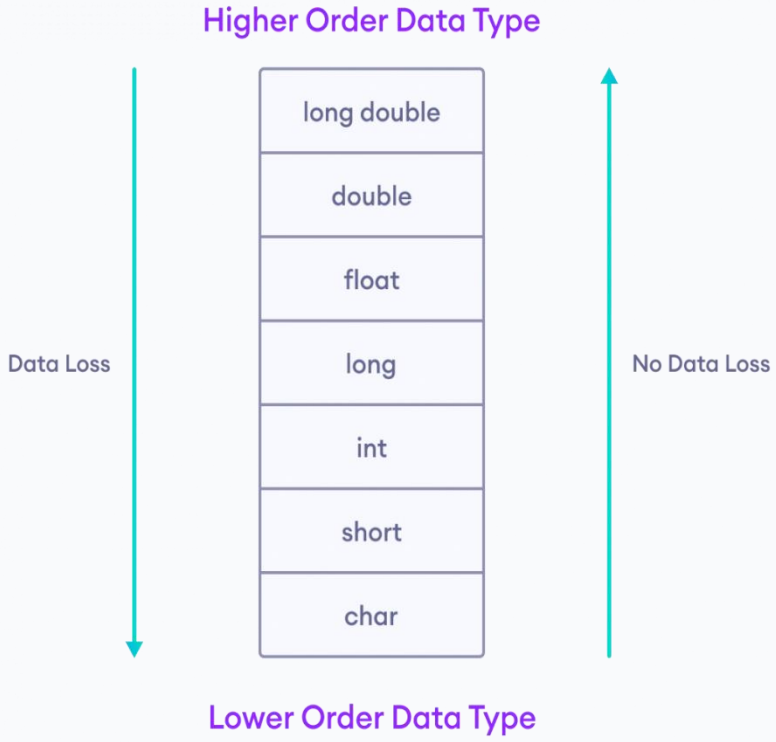
Output

```
Double Value: 4150.12
Integer Value: 4150
```

Here, the data **4150.12** is converted to **4150**. In this conversion, data after the decimal, **.12** is lost.

This is because double is a larger data type (**8 bytes**) than int (**4 bytes**), and when we convert data from larger type to smaller, there will be data loss.

Similarly, there is a hierarchy of data types in C programming. Based on the hierarchy, if a higher data type is converted to lower type, data is lost, and if lower data type is converted to higher type, no data is lost.



Possible Data Loss During C Type Conversion

Here,

- **data loss** - if long double type is converted to double type
- **no data loss** - if char is converted to int

Arithmetic Operator:

Plus Operator

It is a simple Plus (+) Operator used to add two given operands. We can use Plus Operator with different data types such as integer, float, long, double, enumerated and string type data to add the given operand.

Syntax:

```
C = A + B;
```

Minus Operator

The minus operator is denoted by the minus (-) symbol. It is used to return the subtraction of the first number from the second number. The data type of the given number can be different types, such as int, float, double, long double, etc., in the programing language.

Syntax:

```
C = A - B;
```

Multiplication Operator

The multiplication operator is represented as an asterisk (*) symbol, and it is used to return the product of n1 and n2 numbers. The data type of the given number can be different types such as int, float, and double in the C programing language.

Syntax

```
C = A * B;
```

Division Operator

The division operator is an arithmetic operator that divides the first (n1) by the second (n2) number.

Using division operator (/), we can divide the int, float, double and long data types variables.

Syntax

```
C = A / B;
```

Modulus Operator

The modulus operator is represented by the percentage sign (%), and it is used to return the remainder by dividing the first number by the second number.

Syntax

```
C = A % B;
```

Increment Operator

Increment Operator is the type of Arithmetic operator, which is denoted by double plus (++) operator. It is used to increase the integer value by 1.

Syntax

```
B = A++;
```

Decrement Operator

Decrement Operator is denoted by the double minus (--) symbol, which decreases the operand value by 1.

Syntax

```
B = A--;
```

The following table shows all the arithmetic operators supported by the C language. Assume variable **A** holds 10 and variable **B** holds 20, then

Operator	Description	Example
+	Adds two operands.	A + B = 30
−	Subtracts second operand from the first.	A − B = -10
*	Multiplies both operands.	A * B = 200

/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

Program:

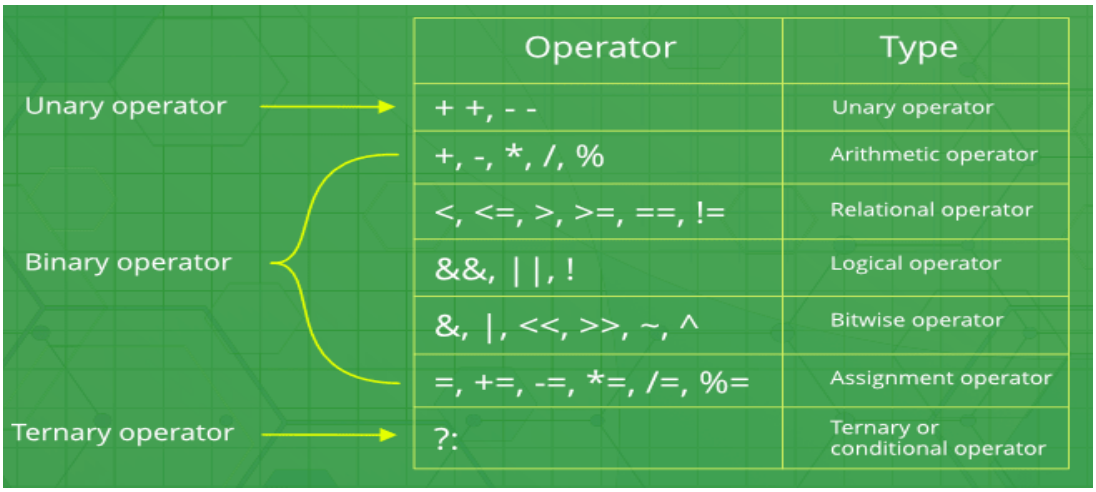
```
#include <stdio.h>
main()
{
   int a = 21;
   int b = 10;
   int c ;

   c = a + b;
   printf("Addition: %d\n", c );

   c = a - b;
   printf("Subtraction: %d\n", c );

   c = a * b;
   printf("Multiplication:%d\n", c );

   c = a / b;
   printf("Division: %d\n", c );

   c = a % b;
   printf("Modulus: %d\n", c );

   c = a++;
   printf("Increment:  %d\n", c );

   c = a--;
   printf("Decrement: %d\n", c );
}
```

Output

```
Addition: 31
Subtraction: 11
Multiplication: 210
Division: 2
Modulus: 1
Increment: 21
Decrement: 22
```

Bitwise Operators in C

In C, the following 6 operators are bitwise operators (also known as bit operators as they work at the bit-level). They are used to perform bitwise operations in C.



Operator	Type
+ +, - -	Unary operator
+, -, *, /, %	Arithmetic operator
<, <=, >, >=, ==, !=	Relational operator
&&, | |, !	Logical operator
&, |, <<, >>, ~, ^	Bitwise operator
=, +=, -=, *=, /=, %=	Assignment operator
?:	Ternary or conditional operator

The **& (bitwise AND)** in C takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.
The | **(bitwise OR)** in C takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.
The ^ **(bitwise XOR)** in C takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.
The << **(left shift)** in C takes two numbers, the left shifts the bits of the first operand, and the second operand decides the number of places to shift.
The >> **(right shift)** in C takes two numbers, right shifts the bits of the first operand, and the second operand decides the number of places to shift.
The **~ (bitwise NOT)** in C takes one number and inverts all bits of it.

Let's look at the truth table of the bitwise operators:

X	Y	X&Y	X|Y	X ^ Y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Example of Bitwise Operators in C:

The following program uses bitwise operators to perform bit operations in C.

```
// C Program to demonstrate use of bitwise operators
#include <stdio.h>
int main()
{
     // a = 5(00000101), b = 9(00001001)
     unsigned char a = 5, b = 9;
     // The result is 00000001
     printf("a = %d, b = %d\n", a, b);
     printf("a&b = %d\n", a & b);
     // The result is 00001101
     printf("a|b = %d\n", a | b);
     // The result is 00001100
     printf("a^b = %d\n", a ^ b);
     // The result is 11111010
     printf("~a = %d\n", a = ~a);
     // The result is 00010010
     printf("b<<1 = %d\n", b << 1);
     // The result is 00000100
     printf("b>>1 = %d\n", b >> 1);
     return 0;
}
```

Output

```
a = 5, b = 9
a&b = 1
a|b = 13
a^b = 12
~a = 250
b<<1 = 18
b>>1 = 4
```

Example: Below program demonstrates the use XOR operator to find odd occurring elements in an array.

```
// C program to find odd occcuring elements in an array
#include <stdio.h>
// Function to return the only odd
// occurring element
int findOdd(int arr[], int n)
{
    int res = 0, i;
    for (i = 0; i < n; i++)
        res ^= arr[i];
    return res;
}
int main(void)
{
    int arr[] = { 12, 12, 14, 90, 14, 14, 14 };
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    printf("The odd occurring element is %d ",findOdd(arr, n));
    return 0;
}
```

Output: The odd occurring element is 90

Example

The below example demonstrates the use bitwise & operator to find if the given number is even or odd.

```
#include <stdio.h>
int main()
{
    int x = 19;
    (x & 1) ? printf("Odd") : printf("Even");
    return 0;
}
```

Output: Odd

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e,. 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Example: Try the following example to understand all the bitwise operators available in C –

```
#include <stdio.h>
Int main()
{
   unsigned int a = 60;   /* 60 = 0011 1100 */
   unsigned int b = 13;   /* 13 = 0000 1101 */
   int c = 0;
   c = a & b;       /* 12 = 0000 1100 */
   printf("Line 1 - Value of c is %d\n", c );
```

```
   c = a | b;       /* 61 = 0011 1101 */
   printf("Line 2 - Value of c is %d\n", c );
   c = a ^ b;       /* 49 = 0011 0001 */
   printf("Line 3 - Value of c is %d\n", c );
   c = ~a;          /*-61 = 1100 0011 */
   printf("Line 4 - Value of c is %d\n", c );
   c = a << 2;     /* 240 = 1111 0000 */
   printf("Line 5 - Value of c is %d\n", c );
   c = a >> 2;     /* 15 = 0000 1111 */
   printf("Line 6 - Value of c is %d\n", c );
}
```

Output

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15
```

sizeof operator in C

Sizeof is a much-used operator in the C. It is a compile-time unary operator which can be used to compute the size of its operand. The result of sizeof is of the unsigned integral type which is usually denoted by size_t. sizeof can be applied to any data type, including primitive types such as integer and floating-point types, pointer types, or compound datatypes such as Structure, union, etc.

Syntax:

sizeof(Expression);

where ‘Expression ‘can be a data type or a variable of any type.

Return: It returns the size size of the given expression.

Usage of sizeof() operator

sizeof() operator is used in different ways according to the operand type.

1. When the operand is a Data Type: When sizeof() is used with the data types such as int, float, char… etc it simply returns the amount of memory allocated to that data types.

Example:

```
// C Program To demonstrate
// sizeof operator
#include <stdio.h>
int main()
```

```
{
     printf("%lu\n", sizeof(char));
     printf("%lu\n", sizeof(int));
     printf("%lu\n", sizeof(float));
     printf("%lu", sizeof(double));
     return 0;
}
```

Output:

1

4

4

8

Note: sizeof() may give different output according to machine, we have run our program on a 32-bit gcc compiler.

2. When the operand is an expression: When sizeof() is used with the expression, it returns the size of the expression.

Example:

```
// C Program To demonstrate
// operand as expression
#include <stdio.h>
int main()
{
     int a = 0;
     double d = 10.21;
     printf("%lu", sizeof(a + d));
     return 0;
}
```

Output: 8

As we know from the first case size of int and double is 4 and 8 respectively, a is int variable while d is a double variable. The final result will be double, Hence the output of our program is 8 bytes.

Need of Sizeof

1. To find out the number of elements in an array: Sizeof can be used to calculate the number of elements of the array automatically.

Example:

```
// C Program
```

```
// demonstrate the method
// to find the number of elements
// in an array
#include <stdio.h>
int main()
{
     int arr[] = { 1, 2, 3, 4, 7, 98, 0, 12, 35, 99, 14 };
     printf("Number of elements:%lu ",
          sizeof(arr) / sizeof(arr[0]));
     return 0;
}
```

Output: Number of elements:11

2. To allocate a block of memory dynamically: sizeof is greatly used in dynamic memory allocation. For example, if we want to allocate memory that is sufficient to hold 10 integers and we don't know the sizeof(int) in that particular machine. We can allocate with the help of sizeof.

Syntax:

int* ptr = (int*)malloc(10 * sizeof(int));