C Language Introduction

C is a procedural programming language. It was initially developed by Dennis Ritchie between 1969 and 1973. It was mainly developed as a system programming language to write operating system. The main features of C language include low-level access to memory, simple set of keywords, and clean style, these features make C language suitable for system programming like operating system or compiler development. Many later languages have borrowed syntax/features directly or indirectly from C language. Like syntax of Java, PHP, JavaScript and many other languages is mainly based on C language. C++ is nearly a superset of C language (There are few programs that may compile in C, but not in C++).

Following is first program in C

```
#include <stdio.h>
int main(void)
{
    printf("GeeksQuiz");
    return 0;
}
```

Let us analyze the program line by line.

Line 1: [#include <stdio.h>] In a C program, all lines that start with # are processed by preporcessor which is a program invoked by the compiler. In a very basic term, preprocessor takes a C program and produces another C program. The produced program has no lines starting with #, all such lines are processed by the preprocessor. In the above example, preprocessor copies the preprocessed code of stdio.h to our file. The .h files are called header files in C. These header files generally contain declaration of functions. We need stdio.h for the function printf() used in the program.

Line 2 [int main(void)] There must to be starting point from where execution of compiled C program begins. In C, the execution typically begins with first line of main(). The void written in brackets indicates that the main doesn't take any parameter (See this for more details). main() can be written to take parameters also. We will be covering that in future posts.
The int written before main indicates return type of main(). The value returned by main indicates status of program termination. See this post for more details on return type.

Line 3 and 6: [{ and }] In C language, a pair of curly brackets define a scope and mainly used in functions and control statements like if, else, loops. All functions must start and end with curly brackets.

Line 4 [printf("GeeksQuiz");] printf() is a standard library function to print something on standard output. The semiolon at the end of printf indicates line termination. In C, semicolon is always used to indicate end of statement.

Line 5 [return 0;] The return statement returns the value from main(). The returned value may be used by operating system to know termination status of your program. The value 0 typically means successful termination.

C Programming Language Standard

The idea of this article is to introduce C standard.

What to do when a C program produces different results in two different compilers?
For example, consider the following simple C program.

```
void main() {  }
```
Run on IDE
The above program fails in gcc as the return type of main is void, but it compiles in Turbo C. How do we decide whether it is a legitimate C program or not?

Consider the following program as another example. It produces different results in different compilers.

```
#include<stdio.h>
int main()
{
    int i = 1;
    printf("%d %d %d\n", i++, i++, i);
    return 0;
}
```
Run on IDE

```
2 1 3 - using g++ 4.2.1 on Linux.i686

1 2 3 - using SunStudio C++ 5.9 on Linux.i686

2 1 3 - using g++ 4.2.1 on SunOS.x86pc

1 2 3 - using SunStudio C++ 5.9 on SunOS.x86pc

1 2 3 - using g++ 4.2.1 on SunOS.sun4u

1 2 3 - using SunStudio C++ 5.9 on SunOS.sun4u
```

.

Is it fine to write “void main()” or “main()” in C/C++?

To summarize above, it is never a good idea to use “void main()” or just “main()” as it doesn’t confirm standards. It may be allowed by some compilers though.

Difference between “int main()” and “int main(void)” in C/C++?

Consider the following two definitions of main().

```
int main()
{
   /*  */
   return 0;
}
```
Run on IDE
and

```
int main(void)
{
   /*  */
   return 0;
}
```
Run on IDE
What is the difference?

In C++, there is no difference, both are same.

Both definitions work in C also, but the second definition with void is considered technically better as it clearly specifies that main can only be called without any parameter.
In C, if a function signature doesn't specify any argument, it means that the function can be called with any number of parameters or without any parameters. For example, try to compile and run following two C programs (remember to save your files as .c). Note the difference between two signatures of fun().

```
// Program 1 (Compiles and runs fine in C, but not in C++)
void fun() {  }
int main(void)
{
    fun(10, "GfG", "GQ");
    return 0;
}
```

The above program compiles and runs fine (See this), but the following program fails in compilation (see this)

```
// Program 2 (Fails in compilation in both C and C++)
void fun(void) {  }
int main(void)
{
    fun(10, "GfG", "GQ");
    return 0;
}
```

Unlike C, in C++, both of the above programs fails in compilation. In C++, both fun() and fun(void) are same.

So the difference is, in C, *int main()* can be called with any number of arguments, but *int main(void)* can only be called without any argument. Although it doesn't make any difference most of the times, using "int main(void)" is a recommended practice in C.

Exercise:
Predict the output of following **C** programs.
Question 1// correct

```
#include <stdio.h>
int main()
{
    static int i = 5;
    if (--i){
        printf("%d ", i);
        main(10);
    }
}
```

Run on IDE

Question 2 //error

```
#include <stdio.h>
int main(void)
{
    static int i = 5;
    if (--i){
        printf("%d ", i);
        main(10);// x
    }
```

Interesting Facts about Macros and Preprocessors in C

In a C program, all lines that start with # are processed by preporcessor which is a special program invoked by the compiler. In a very basic term, preprocessor takes a C program and produces another C program without any #.
Following are some interesting facts about preprocessors in C.

1) When we use ***include*** directive, the contents of included header file (after preprocessing) are copied to the current file.
Angular brackets **<** and **>** instruct the preprcessor to look in the standard folder where all header files are held. Double quotes **“** and **“** instruct the preprcessor to look into the current folder and if the file is not present in current folder, then in standard folder of all header files.

2) When we use ***define*** for a constant, the preprocessor produces a C program where the defined constant is searched and matching tokens are replaced with the given expression. For example in the following program *max*is defined as 100.

```
#include<stdio.h>
#define max 100
int main()
{
    printf("max is %d", max);
    return 0;
}
// Output: max is 100
// Note that the max inside "" is not replaced
```

Run on IDE

3) The macros can take function like arguments, the arguments are not checked for data type. For example, the following macro INCREMENT(x) can be used for x of any data type.

```
#include <stdio.h>
#define INCREMENT(x) ++x
int main()
{
    char *ptr = "GeeksQuiz";
    int x = 10;
    printf("%s  ", INCREMENT(ptr));
    printf("%d", INCREMENT(x));
    return 0;
}
// Output: eeksQuiz 11
```

Run on IDE

4) The macro arguments are not evaluated before macro expansion. For example consider the following program

```
#include <stdio.h>
#define MULTIPLY(a, b) a*b
int main()
{
    // The macro is expended as 2 + 3 * 3 + 5, not as 5*8
    printf("%d", MULTIPLY(2+3, 3+5));
    return 0;
}
// Output: 16
```

Run on IDE

5) The tokens passed to macros can be contcatehanted using operator ## called Token-Pasting operator.

```
#include <stdio.h>
#define merge(a, b) a##b
int main()
{
    printf("%d ", merge(12, 34));
}
// Output: 1234
```

Run on IDE

6) A token passed to macro can be converted to a sting literal by using # before it.

```
#include <stdio.h>
#define get(a) #a
int main()
{
    // GeeksQuiz is changed to "GeeksQuiz"
    printf("%s", get(GeeksQuiz));
}
// Output: GeeksQuiz
```

Run on IDE

7) The macros can be written in multiple lines using '\'. The last line doesn't need to have '\'.

```
#include <stdio.h>
#define PRINT(i, limit) while (i < limit) \
                        { \
                            printf("GeeksQuiz "); \
                            i++; \
                        }
int main()
{
    int i = 0;
    PRINT(i, 3);
    return 0;
}
// Output: GeeksQuiz  GeeksQuiz  GeeksQuiz
```

Run on IDE

8) The macros with arguments should be avoided as they cause problems sometimes. And Inline functions should be preferred as there is type checking parameter evaluation in inline functions. From C99 onward, inline functions are supported by C language also.
For example consider the following program. From first look the output seems to be 1, but it produces 36 as output.

```
#define square(x) x*x
int main()
{
  int x = 36/square(6); // Expended as 36/6*6
  printf("%d", x);
  return 0;
}
// Output: 36
```

Run on IDE
If we use inline functions, we get the expected output. Also the program given in point 4 above can be corrected using inline functions.

```
inline int square(int x) { return x*x; }
```

```
int main()
{
  int x = 36/square(6);
  printf("%d", x);
  return 0;
}
// Output: 1
```
Run on IDE

9) Preprocessors also support if-else directives which are typically used for conditional compilation.
```
int main()
{
#if VERBOSE >= 2
  printf("Trace Message");
#endif
}
```
Run on IDE

10) A header file may be included more than one time directly or indirectly, this leads to problems of redeclaration of same variables/functions. To avoid this problem, directives like ***defined***, ***ifdef*** and ***ifndef*** are used.

11) There are some standard macros which can be used to print program file (__FILE__), Date of compilation (__DATE__), Time of compilation (__TIME__) and Line Number in C code (__LINE__)
```
#include <stdio.h>

int main()
{
   printf("Current File :%s\n", __FILE__ );
   printf("Current Date :%s\n", __DATE__ );
   printf("Current Time :%s\n", __TIME__ );
   printf("Line Number :%d\n", __LINE__ );
   return 0;
}

/* Output:
Current File :C:\Users\GfG\Downloads\deleteBST.c
Current Date :Feb 15 2014
Current Time :07:04:25
Line Number :8 */
```

Variables and Keywords in C

A **variable** in simple terms is a storage place which has some memory allocated to it. So basically a variable used to store some form of data. Different types of variables require different amounts of memory and have some specific set of operations which can be applied on them.

Variable **Declaration:**

A typical variable declaration is of the form:

```
type variable_name;

  or for multiple variables:

type variable1_name, variable2_name, variable3_name;
```

A variable name can consist of alphabets (both upper and lower case), numbers and the underscore '_' character. However, the name must not start with a number.

Difference b/w variable declaration and definition
Variable declaration refers to the part where a variable is first declared or introduced before its first use. Variable definition is the part where the variable is assigned a memory location and a value. Most of the times, variable declaration and definition are done together
See the following C program for better clarification:

```
#include <stdio.h>

int main()

{

    // declaration and definition of variable 'a123'

    char a123 = 'a';


    // This is also both declaration and definition as 'b' is allocated

    // memory and assigned some garbage value.

    float b;


    // multiple declarations and definitions

    int _c, _d45, e;


    // Let us print a variable

    printf("%c \n", a123);


    return 0;

}
```

Run on IDE

Output:

```
a
```

Is it possible to have separate declaration and definition?
It is possible in case of extern variables and functions. See question 1 of this for more details.

```
// This is only declaration. y is not allocated memory by this statement

  extern int y;

// This is both declaration and definition, memory to x is allocated by this statement.

  int x;
```

Keywords are specific reserved words in C each of which has a specific feature associated with it. Almost all of the words which help us use the functionality of the C language are included in the list of keywords. So you can imagine that the list of keywords is not going to be a small one!
There are a total of 32 keywords in C:

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

Most of these keywords have already been discussed in the various sub-sections of the C language, like Data Types, Storage Classes, Control Statements, Functions etc.
Let us discuss some of the other keywords which allow us to use the basic functionality of C:

const: const can be used to declare constant variables. Constant variables are variables which when initialized, can't change their value. Or in other words, the value assigned to them is unmodifiable. Syntax:

```
const data_type var_name = var_value;
```

Note: Constant variables need to be compulsorily be initialized during their declaration itself. const keyword is also used with pointers. Please refer the const qualifier in C for understanding the same.

extern: extern simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block. Also, a normal global variable can me made extern as well by placing the 'extern' keyword before its declaration/definition in any function/block. This basically signifies that we are not initializing a new variable but instead we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program. Syntax:

```
extern data_type var_name = var_value;
```

static: static keyword is used to declare static variables which are popularly used while writing programs in C language. Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope. So we can say that they are initialized only once and exist till the termination of the program. Thus, no new memory is allocated because they are not re-declared. Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere in the program. By default, they are assigned the value 0 by the compiler.
Syntax:

```
static data_type var_name = var_value;
```

void: void is a special data type only. But what makes it so special? void, as it literally means an empty data type. It means it has nothing or it holds no value. For example, when it is used as the return data type for a function, it simply represents that the function returns no value. Similarly, when it is added to a function heading, it represents that the function takes no arguments.

Note: void also has a significant use with pointers. Please refer the void pointer in C for understanding the same.
typedef: typedef is used to give a new name to an already existing or even a custom data type (like a structure). It comes in very handy at times, for example in a case when the name of the structure defined by you is very long or you just need a short-hand notation of a per-existing data type.

How are variables scoped in C – Static or Dynamic?

In C, variables are always statically (or lexically) scoped i.e., binding of a variable can be determined by program text and is independent of the run-time function call stack.
For example, output for the below program is 0, i.e., the value returned by f() is not dependent on who is calling it. f() always returns the value of global variable x.

```
int x = 0;
int f()
{
   return x;
}
int g()
{
   int x = 1;
   return f();
}
int main()
{
  printf("%d", g());
  printf("\n");
  getchar();
}
```

Scope rules in C

Scope of an identifier is the part of the program where the identifier may directly be accessible. In C, all identifiers are lexically (or statically) scoped. C scope rules can be covered under following two categories.
Global Scope: Can be accessed anywhere in a program.

```
// filename: file1.c
int a;
int main(void)
```

```
{
   a = 2;
}
// filename: file2.c
// When this file is linked with file1.c, functions of this file can access a
extern int a;
int myfun()
{
   a = 2;
}
```

To restrict access to current file only, global variables can be marked as static.

Block Scope: A Block is a set of statements enclosed within left and right braces ({ and } respectively). Blocks may be nested in C (a block may contain other blocks inside it). A variable declared in a block is accessible in the block and all inner blocks of that block, but not accessible outside the block.
What if the inner block itself has one variable with the same name?
If an inner block declares a variable with the same name as the variable declared by the outer block, then the visibility of the outer block variable ends at the pint of declaration by inner block.

```
int main()
{
  {
      int x = 10, y  = 20;
      {
          // The outer block contains declaration of x and y, so
          // following statement is valid and prints 10 and 20
          printf("x = %d, y = %d\n", x, y);
          {
              // y is declared again, so outer block y is not accessible
              // in this block
              int y = 40;

              x++;  // Changes the outer block variable x to 11
              y++;  // Changes this block's variable y to 41

              printf("x = %d, y = %d\n", x, y);
          }
```

```
          // This statement accesses only outer block's variables
          printf("x = %d, y = %d\n", x, y);
      }
  }
  return 0;
}
```

Output:

```
x = 10, y = 20
x = 11, y = 41
x = 11, y = 20
```

Integer Promotions in C

Some data types like *char* , *short int* take less number of bytes than *int*, these data types are automatically promoted to *int* or *unsigned int* when an operation is performed on them. This is called integer promotion. For example no arithmetic calculation happens on smaller types like *char*, *short* and *enum*. They are first converted to*int* or *unsigned int*, and then arithmetic is done on them. If an *int* can represent all values of the original type, the value is converted to an *int* . Otherwise, it is converted to an *unsigned int.*
For example see the following program.

```
#include <stdio.h>
int main()
{
    char a = 30, b = 40, c = 10;
    char d = (a * b) / c;
    printf ("%d ", d);
    return 0;
}
```

Output:

```
120
```

At first look, the expression (a*b)/c seems to cause arithmetic overflow because signed characters can have values only from -128 to 127 (in most of the C compilers), and the value of subexpression '(a*b)' is 1200 which is greater than 128. But integer promotion happens here in arithmetic done on char types and we get the appropriate result without any overflow.

Consider the following program as **another example**.

```
#include <stdio.h>


int main()
{
    char a = 0xfb;
    unsigned char b = 0xfb;


    printf("a = %c", a);
    printf("\nb = %c", b);


    if (a == b)
      printf("\nSame");
    else
      printf("\nNot Same");
    return 0;
}
```

Output:

```
a = ?
b = ?
Not Same
```

When we print ‘a’ and ‘b’, same character is printed, but when we compare them, we get the output as “Not Same”.
‘a’ and ‘b’ have same binary representation as *char*. But when comparison operation is performed on ‘a’ and ‘b’, they are first converted to int. ‘a’ is a signed *char*, when it is converted to *int*, its value becomes -5 (signed value of 0xfb). ‘b’ is *unsigned char*, when it is converted to *int*, its value becomes 251. The values -5 and 251 have different representations as *int*, so we get the output as “Not Same”.

In C, printf() returns the number of **characters** successfully written on the output and scanf() returns number of**items** successfully read.
For example, below program prints geeksforgeeks **13**

```
int main()
{
  printf(" %d", printf("%s", "geeksforgeeks"));
  getchar();
}
```

Run on IDE
Irrespective of the string user enters, below program prints **1**.

```
int main()
{
  char a[50];
  printf(" %d", scanf("%s", a));
  getchar();
}
```

In C, return type of getchar(), fgetc() and getc() is int (not char). So it is recommended to assign the returned values of these functions to an integer type variable.

```
char ch;  /* May cause problems */
  while ((ch = getchar()) != EOF)
  {
     putchar(ch);
  }
```

Here is a version that uses integer to compare the value of getchar().

```
  int in;
  while ((in = getchar()) != EOF)
  {
     putchar(in);
  }
```

Scansets in C

scanf family functions support scanset specifiers which are represented by %[]. Inside scanset, we can specify single character or range of characters. While processing scanset, scanf will process only those characters which are part of scanset. We can define scanset by putting characters inside squre brackets. Please note that the scansets are case-sensitive.
Let us see with example. Below example will store only capital letters to character array ‘str’, any other character will not be stored inside character array.

```
/* A simple scanset example */

#include <stdio.h>


int main(void)
```

```
{
    char str[128];

    printf("Enter a string: ");
    scanf("%[A-Z]s", str);

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

    return 0;
}
```

Run on IDE

```
[root@centos-6 C]# ./scan-set
Enter a string: GEEKs_for_geeks
You entered: GEEK
```

If first character of scanset is '^', then the specifier will stop reading after first occurence of that character. For example, given below scanset will read all characters but stops after first occurence of 'o'

```
scanf("%[^o]s", str);
```

Run on IDE

Let us see with example.

```
/* Another scanset example with ^ */
#include <stdio.h>

int main(void)
{
    char str[128];

    printf("Enter a string: ");
    scanf("%[^o]s", str);

    printf("You entered: %s\n", str);
```

```
    return 0;
}
```

Run on IDE

```
[root@centos-6 C]# ./scan-set
Enter a string: http://geeks for geeks
You entered: http://geeks f
[root@centos-6 C]#
```

Let us implement gets() function by using scan set. gets() fucntion reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF found.

```
/* implementation of gets() function using scanset */
#include <stdio.h>

int main(void)
{
    char str[128];

    printf("Enter a string with spaces: ");
    scanf("%[^\n]s", str);

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

    return 0;
}
```

Run on IDE

```
[root@centos-6 C]# ./gets
Enter a string with spaces: Geeks For Geeks
You entered: Geeks For Geeks
[root@centos-6 C]#
```

As a side note, using gets() may not be a good indea in general. Check below note from Linux man page.

Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer,

it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead. Also see this pos

What is use of %n in printf() ?

In C printf(), %n is a special format specifier which instead of printing something causes printf() to load the variable pointed by the corresponding argument with a value equal to the number of characters that have been printed by printf() before the occurrence of %n.

```
#include<stdio.h>

int main()
{
  int c;
  printf("geeks for %ngeeks ", &c);
  printf("%d", c);
  getchar();
  return 0;
}
```

sprintf:

Syntax:

```
int sprintf(char *str, const char *string,...);
```

String print function it is stead of printing on console store it on char buffer which are specified in sprintf

Example:

sizeof operator in C

Sizeof is a much used in the C programming language. It is a compile time unary operator which can be used to compute the size of its operand. The result of sizeof is of 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.

Usage

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

1. When operand is a Data Type.

When *sizeof()* is used with the data types such as int, float, char… etc it simply return amount of memory is allocated to that data types.

Let see example:

```
#include<stdio.h>
int main()
{
    printf("%d\n",sizeof(char));
    printf("%d\n",sizeof(int));
    printf("%d\n",sizeof(float));
    printf("%d", sizeof(double));
    return 0;
}
```

Run on IDE

Output

```
1

4

4

8
```

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

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

```
#include<stdio.h>
int main()
{
    int i = 0;
    double d = 10.21;
    printf("%d", sizeof(a+d));
    return 0;
}
```

Run on IDE

Output

```
8
```

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

Need of Sizeof

1. To find out number of elements in a array.
Sizeof can be used to calculate number of elements of the array automatically. Let see Example :

```
#include<stdio.h>
int main()
{
   int arr[] = {1, 2, 3, 4, 7, 98, 0, 12, 35, 99, 14};
   printf("Number of elements :%d", sizeof(arr)/sizeof(arr[0]));
   return 0;
}
```

Run on IDE

Output

```
Number of elements :11
```

2. To allocate block of memory dynamically.
sizeof is greatly used in dynamic memory allocation. For example, if we want to allocate memory for which 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.

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

A comma operator question

Consider the following C programs.

```
// PROGRAM 1
#include<stdio.h>

int main(void)
{
    int a = 1, 2, 3;//variable name cant start from number and here , work as seprator
    printf("%d", a);
    return 0;
}
```

The above program fails in compilation, but the following program compiles fine and prints 1.

```
// PROGRAM 2
#include<stdio.h>

int main(void)
{
    int a;
    a = 1, 2, 3;// , act as operator
    printf("%d", a);
    return 0;
}
```

And the following program prints 3, why?

```
// PROGRAM 3
#include<stdio.h>

int main(void)
{
    int a;
    a = (1, 2, 3);
    printf("%d", a);
    return 0;
}
```

In a C/C++ program, comma is used in two contexts: (1) A separator (2) An Operator. (See this for more details).

Comma works just as a separator in PROGRAM 1 and we get compilation error in this program.

Comma works as an operator in PROGRAM 2. Precedence of comma operator is least in operator precedence table. So the assignment operator takes precedence over comma and the expression "a = 1, 2, 3″ becomes equivalent to "(a = 1), 2, 3″. That is why we get output as 1 in the second program.
In PROGRAM 3, brackets are used so comma operator is executed first and we get the output as 3 (See the Wiki page for more details).

```
#include<stdio.h>
int main()
{
   int i = 10;
   printf("%d", ++(-i));
   return 0;
}
```

Run on IDE

```
A) 11 B) 10 C) -9 D) None
```

Answer: D, None – Compilation Error.

Precedence of postfix ++ and prefix ++ in C/C++

In C/C++, precedence of Prefix ++ (or Prefix –) and dereference (*) operators is same, and precedence of Postfix ++ (or Postfix –) is higher than both Prefix ++ and *.
If p is a pointer then *p++ is equivalent to *(p++) and ++*p is equivalent to ++(*p) (both Prefix ++ and * are right associative).

For example, program 1 prints *'h'* and program 2 prints *'e'*.

```
// Program 1
#include<stdio.h>
int main()
{
  char arr[] = "geeksforgeeks";
  char *p = arr;
  ++*p;//value of p
  printf(" %c", *p);
  getchar();
  return 0;
}
```

Run on IDE
Output: h

```
// Program 2
#include<stdio.h>
int main()
{
  char arr[] = "geeksforgeeks";
  char *p = arr;
  *p++;
  printf(" %c", *p);
```

```
  getchar();
  return 0;
}
```

Run on IDE
Output: e

Modulus on Negative Numbers

What will be the output of the following C program?

```
int main()
{
   int a = 3, b = -8, c = 2;
   printf("%d", a % b / c);
   return 0;
}
```

The output is 1.
% and / have same precedence and left to right associativity. So % is performed first which results in 3 and / is performed next resulting in 1. The emphasis is, ***sign of left operand is appended to result in case of modulus operator in C***.

C/C++ Ternary Operator – Some Interesting Observations

Predict the output of following C++ program.

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

int main()
{
   int test = 0;
   cout << "First  character " << '1' << endl;
   cout << "Second character " << (test ? 3 : '1') << endl;

   return 0;
}
```

One would expect the output will be same in both the print statements. However, the output will be,

```
First  character 1
Second character 49
```

Why the second statement printing 49? Read on the ternary expression.

Ternary Operator (C/C++):
A ternary operator has the following form,

$exp_1 \; ? \; exp_2 \; : \; exp_3$

The expression exp_1 will be evaluated always. Execution of exp_2 and exp_3 depends on the outcome of exp_1. If the outcome of exp_1 is non zero exp_2 will be evaluated, otherwise exp_3 will be evaluated.

Side Effects:
Any side effects of exp_1 will be evaluated and updated immediately before executing exp_2 or exp_3. In other words, there is sequence point after the evaluation of condition in the ternary expression. If either exp_2 or exp_3 have side effects, only one of them will be evaluated. See the related post.

Return Type:
It is another interesting fact. The ternary operator has return type. The return type depends on exp_2, and *convertibility* of exp_3 into exp_2 as per usual\overloaded conversion rules. If they are not convertible, the compiler throws an error. See the examples below,
The following program compiles without any error. The return type of ternary expression is expected to be *float* (as that of exp_2) and exp_3 (i.e. literal *zero – int* type) is implicitly convertible to *float.*

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

int main()
{
   int test = 0;
   float fvalue = 3.111f;
   cout << (test ? fvalue : 0) << endl;

   return 0;
}
```

The following program will not compile, because the compiler is unable to find return type of ternary expression or implicit conversion is unavailable between exp_2 (*char array*) and exp_3 (*int*).

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

int main()
{
   int test = 0;
```

```
   cout << test ? "A String" : 0 << endl;


   return 0;
}
```

The following program *may* compile, or but fails at runtime. The return type of ternary expression is bounded to type (*char **), yet the expression returns *int*, hence the program fails. Literally, the program tries to print string at 0th address at runtime.

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


int main()
{
   int test = 0;
   cout << (test ? "A String" : 0) << endl;


   return 0;
}
```

We can observe that exp_2 is considered as output type and exp_3 will be converted into exp_2 at runtime. If the conversion is implicit the compiler inserts stubs for conversion. If the conversion is explicit the compiler throws an error. If any compiler misses to catch such error, the program may fail at runtime.

In C++, pre-increment (or pre-decrement) can be used as l-value, but post-increment (or post-decrement) can not be used as l-value.
For example, following program prints *a = 20* (++a is used as l-value)

```
#include<stdio.h>


int main()
{
  int a = 10;
  ++a = 20; // works
  printf("a = %d", a);
  getchar();
  return 0;
}
```

And following program fails in compilation with error *"non-lvalue in assignment"* (a++ is used as l-value)

```
#include<stdio.h>


int main()
{
  int a = 10;
  a++ = 20; // error
  printf("a = %d", a);
  getchar();
  return 0;
}
```

Results of comparison operations in C and C++

In C, data type of result of comparison operations is int.in c++ it is bool

ADD WITHOUT SUM OPERATOR

```
int add(int x, int y)
{
    return printf("%*c%*c",  x, ' ',  y, ' ');
}

int main()
{
    printf("Sum = %d", add(3, 4));
    return 0;
}
```

```
#include <stdio.h>
int main()
{
   int i = 8;
   int p = i++*i++;//undefined output
   printf("%d\n", p);
}
```

- Output of the program | Dereference, Reference, Dereference, Reference// it is left

An Uncommon representation of array elements

Consider the below program.

```
int main( )
{
```

```
  int arr[2] = {0,1};
  printf("First Element = %d\n",arr[0]);
  getchar();
  return 0;
}
```

Pretty Simple program.. huh… Output will be 0.

Now if you replace *arr[0]* with *0[arr]*, the output would be same. Because compiler converts the array operation in pointers before accessing the array elements.
e.g. *arr[0]* would be **(arr + 0)* and therefore 0[arr] would be **(0 + arr)* and you know that both **(arr + 0)* and **(0 + arr)* are same.

Pointer vs Array in C

Most of the time, pointer and array accesses can be treated as acting the same, the major exceptions being:

1) the sizeof operator
o sizeof(array) returns the amount of memory used by all elements in array
o sizeof(pointer) only returns the amount of memory used by the pointer variable itself

2) the & operator
o &array is an alias for &array[0] and returns the address of the first element in array
o &pointer returns the address of pointer

3) a string literal initialization of a character array
o char array[] = “abc” sets the first four elements in array to ‘a’, ‘b’, ‘c’, and ‘\0′
o char *pointer = “abc” sets pointer to the address of the “abc” string (which may be stored in read-only memory and thus unchangeable)

4) Pointer variable can be assigned a value whereas array variable cannot be.

```
int a[10];
int *p;
p=a; /*legal*/
a=p; /*illegal*/
```

5) Arithmetic on pointer variable is allowed.

```
p++; /*Legal*/
a++; /*illegal*/
```

void pointer in C

A void pointer is a pointer that has no associated data type with it. A void pointer can hold address of any type and can be typcasted to any type.

```
int a = 10;

char b = 'x';


void *p = &a;  // void pointer holds address of int 'a'

p = &b; // void pointer holds address of char 'b'
```

Run on IDE

Advantages of void pointers:

1) malloc() and calloc() return void * type and this allows these functions to be used to allocate memory of any data type (just because of void *)

```
int main(void)

{

    // Note that malloc() returns void * which can be

    // typecasted to any type like int *, char *, ..

    int *x = malloc(sizeof(int) * n);

}
```

Run on IDE

Note that the above program compiles in C, but doesn't compile in C++. In C++, we must explicitly typecast return value of malloc to (int *).

2) void pointers in C are used to implement generic functions in C. For example compare function which is used in qsort().

Some Interesting Facts:

1) void pointers cannot be dereferenced. For example the following program doesn't compile.

```
#include<stdio.h>

int main()

{

    int a = 10;

    void *ptr = &a;

    printf("%d", *ptr);

    return 0;

}
```

Run on IDE

Output:

```
Compiler Error: 'void*' is not a pointer-to-object type
```

The following program compiles and runs fine.

```
#include<stdio.h>

int main()

{

    int a = 10;

    void *ptr = &a;

    printf("%d", *(int *)ptr);

    return 0;

}
```

Run on IDE

Output:

```
10
```

MCQ BULLETS

->Char arr[100];
`printf("%d", scanf("%s", arr));`

//it output 1 it take whole array as1 input

->Usage of \b and \r in C **output depend on terminal configuration**

->printf(5 + "GeeksQuiz");//out put is Quiz

->printf("%c ", 5["GeeksQuiz"]);//output is Q eq to printf("%c ","GeeksQuiz"[5]);
Reason
The crux of the program lies in the expression: 5["GeeksQuiz"] **This expression is broken down by the compiler as:** *(5 + "GeeksQuiz")**. Adding 5 to the base address of the string increments the pointer(lets say a pointer was pointing to the start(**G**) of the string initially) to point to** Q**.**
Applying value-of **operator gives the character at the location pointed to by the pointer i.e. Q.**

-> gets() can read a string with spaces but a normal scanf() with %s can no

->
```
char *s = "Geeks Quiz";

    int n = 7;
    printf("%.*s", n, s);
```
//output is Geeks Q

.* means The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

-> User applications are not allowed to perform I/O in user mode - All I/O requests are handled through system calls that must be performed in kernel mode.

POINTER MCQ

```
->(*ptr)++;         /* increment the value at ptr by 1 */
```

->

```
#include <stdio.h>

int main()
{
    float arr[5] = {12.5, 10.0, 13.5, 90.5, 0.5};
    float *ptr1 = &arr[0];
    float *ptr2 = ptr1 + 3;

    printf("%f ", *ptr2);
    printf("%d", ptr2 - ptr1);

   return 0;
}
```

Run on IDE

90.500000 3 //interger subtration

->

```
#include<stdio.h>
int main()
{
    int arr[] = {10, 20, 30, 40, 50, 60};
    int *ptr1 = arr;
    int *ptr2 = arr + 5;
    printf("Number of elements between two pointer are: %d.",
                                (ptr2 - ptr1));
    printf("Number of bytes between two pointers are: %d",
                              (char*)ptr2 - (char*) ptr1);
    return 0;
}
```

Run on IDE

Assume that an int variable takes 4 bytes and a char variable takes 1 byte

Number of elements between two pointer are: 5. Number of bytes between two pointers are: 20

->never use sizeof for finding length of array inside calling function

->Java uses reference variables to implement these data structures. Note that C doesn't support reference variables.

->

->

```
#include<stdio.h>
void f(int *p, int *q)
{
```

```
  p = q; //p starts pointing to q
  *p = 2; //j get changed
}
int i = 0, j = 1;
int main()
{
  f(&i, &j);
  printf("%d %d \n", i, j);
  getchar();
  return 0;
}
```
// output is 0 2

->

```
int f(int x, int *py, int **ppz)
{
  int y, z;
  **ppz += 1;//c changed to 5
   z  = **ppz;
  *py += 2; // c changed to 7
   y = *py;
   x += 3; //note x is still 4 so sx is 7
   return x + y + z;
}

void main()
{
   int c, *b, **a;
   c = 4;
   b = &c;
   a = &b;
   printf("%d ", f(c, b, a));
   return 0;
}
```
// output is 19

->

```
#include<stdio.h>
int main()
{
    int a = 12;
    void *ptr = (int *)&a;
    printf("%d", *ptr);//compiler error
    printf("%d", *(int *)ptr);//it is fine
    getchar();
    return 0;
}
```
->

```
#include<stdio.h>

void swap (char *x, char *y)//need to pass double pointer
```

```
{
    char *t = x;
    x = y;
    y = t;
}

int main()
{
    char *x = "geeksquiz";
    char *y = "geeksforgeeks";
    char *t;
    swap(x, y);//x,y not get swapped their copy is passed
    printf("(%s, %s)", x, y);
    t = x;
    x = y;
    y = t;
    printf("\n(%s, %s)", x, y);
    return 0;
}
```

->

```
int ( * f) (int * ) ;
```

//A pointer to a function that takes an integer pointer as argument and returns an integer

->Expression gets evaluated in cases (switch). The control goes to the second case block after evaluating **1+2 = 3** and **Quiz** is printed.

->divide by 0 is runtime error

->inside switch control goes directly to label

->A switch expression can be int, char and enum. A float variable/expression cannot be used inside switch.

->

```
#include <stdio.h>
int main()
{
    char check = 'a';
    switch (check)
    {
        case 'a' || 1: printf("Geeks ");// case 1:

        case 'b' || 2: printf("Quiz ");// case 1;
                    break;
        default: printf("GeeksQuiz");
    }
    return 0;
}//so compile time error
```

->

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

```
    int check = 20, arr[] = {10, 20, 30};
    switch (check)
    {
        case arr[0]: printf("Geeks ");//case ke inside their must be constant
        case arr[1]: printf("Quiz ");
        case arr[2]: printf("GeeksQuiz");
    }
    return 0;
}
```

//compile error

->static keyword use

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

class Test
{
    static int x;
public:
    Test() { x++; }
    static int getX() {return x;}
};

int Test::x = 0;

int main()
{
    cout << Test::getX() << " ";
    Test t[5];
    cout << Test::getX();
}

->
```

Static methods can be overloaded
Static data members can be accessed by any methods.
Non-static data members cannt be accessed by static methods.
Static methods can only access static members (data and methods)

*These functions(*Static Member Functions

) cannot access ordinary data members and member functions, but only static data members and static member functions.

It doesn't have any "this" keyword which is the reason it cannot access ordinary members. We will study about "this" keyword later.

*Also, it(*Static data member in class

) must be initialized explicitly, always outside the class. If not initialized, Linker will give error

If you don't initialize a static variable(Static data member inside function), they are by default initialized to zero.

->constant Keyword

If you make any variable as constant, using const keyword, you cannot change its value. Also, the constant variables must be initialized while declared.

Pointer to Const

```
const int* u;

int const* v;
```

Const pointer

To make the pointer const, we have to put the **const** keyword to the right of the `*`.

```
int x = 1;

int* const w = &x;
```

Here, w is a pointer, which is const, that points to an int. Now we can't change the pointer but can change the value that it points to.

NOTE : We can also have a const pointer pointing to a const variable.

```
const int* const x;
```

3) Const Function Arguments and Return types

We can make the return type or arguments of a function as const. Then we cannot change any of them.

```
void f(const int i)

{

 i++;    // Error

}


const int g()

{

 return 1;
```

```
}
```

4) Const class Data members

These are data variables in class which are made const. They are not initialized during declaration. Their initialization occur in the constructor.

```
class Test
{
 const int i;
 public:
 Test (int x)
 {
   i=x;
 }
};

int main()
{
 Test t(10);
 Test s(20);
}
```

In this program, **i** is a const data member, in every object its independent copy is present, hence it is initialized with each object using constructor. Once initialized, it cannot be changed.

5) Const class Object

When an object is declared or created with const, its data members can never be changed, during object's lifetime.

Syntax :

```
const class_name object;
```

Mutable Keyword

Mutable keyword is used with member variables of class, which we want to change even if the object is of const type. Hence, mutable data members of a const objects can be modified.

In C++, const qualifier can be applied to 1) Member functions of a class 2) Function arguments 3) To a cl ass data member which is declared as static 4) Reference variables

A const object can only call const functions.

```
int getX() const { return x; }
```

```
#include <stdio.h>
int main()
{
   const int x;//must be intialised here
   x = 10;//compile error
   printf("%d", x);
   return 0;
}
```

The ‘this’ pointer is passed as a hidden argument to all non-static member function calls and is available as a local variable within the body of all non-static functions. ‘this’ pointer is a constant pointer that holds the memory address of the current object. ‘this’ pointer is not available in static member functions as static member functions can be called without any object (with class name). Source: ‘this’ pointer in C++

Final Keyword In Java

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant)

2) Java final method

If you make any method as final, you cannot override it.

3) Java final class

If you make any class as final, you cannot extend it.

Can we initialize blank final variable?

Yes, but only in constructor. For example:

tatic blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example of static blank final variable

```
1. class A{
2.   static final int data;//static blank final variable
3.   static{ data=50;}
4.   public static void main(String args[]){
5.     System.out.println(A.data);
6.  }
7. }
```

Can we declare a constructor final?

No, because constructor is never inherited.

main() is a static method and fun() is a non-static method in class Main. Like C++, in Java calling a non-static function inside a static function is not allowed

Output of following Java Program?

```
class Base {
    public void show() {
       System.out.println("Base::show() called");
    }
}

class Derived extends Base {
    public void show() {
       System.out.println("Derived::show() called");
    }
```

```
}


public class Main {
    public static void main(String[] args) {
        Base b = new Derived();;
        b.show();
    }
}
```

Derived::show() called

B Base::show() called

Discuss it

Question 1 Explanation:
In the above program, b is a reference of Base type and refers to an abject of Derived class. In Java, functions are virtual by default. So the run time polymorphism happens and derived fun() is called.

Question 2

WRONG

```
class Base {
    final public void show() {
       System.out.println("Base::show() called");
    }
}


class Derived extends Base {
    public void show() {
       System.out.println("Derived::show() called");
    }
}


class Main {
    public static void main(String[] args) {
```

```
        Base b = new Derived();;
        b.show();
    }
}
```

A Base::show() called

Derived::show() called

Compiler Error

D Runtime Error

Discuss it

Question 2 Explanation:
Final methods cannot be overridden. See the compiler error here.

Question 3

CORRECT

```
class Base {
    public static void show() {
       System.out.println("Base::show() called");
    }
}

class Derived extends Base {
    public static void show() {
       System.out.println("Derived::show() called");
    }
}

class Main {
    public static void main(String[] args) {
        Base b = new Derived();;
```

```
        b.show();
    }
}
```

Base::show() called

B Derived::show() called

C Compiler Error

Discuss it

Question 3 Explanation:
Like C++, when a function is static, runtime polymorphism doesn't happen

```
1) Private methods are final.
3) Protected methods are final.
4) We cannot override private methods.
```

In Java, it is not allowed to do super.super. We can only access Grandparent's members using Parent. For example, the following program works fine

If we

derive an

abstract

class and

do not

implement

all the

abstract

methods,

then the

derived

class

should

also be

marked

as

abstract

using

'abstract'

keyword

Abstract classes can have constructors

C

A class can be made abstract without any abstract method

A class cannot inherit from multiple abstract classes

```
An interface is similar to a class, but can contain following type of
members.

....public, static, final fields (i.e., constants)

....default and static methods with bodies


2) An instance of interface cannot be created.


3) A class can implement multiple interfaces.


4) Many classes can implement the same interface.
```

The error is in for loop where 0 is used in place of boolean value. Unlike C++, use of non boolean variables in place of bool is not allowed