C Archive

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++).

Beginning with C programming:

1) Finding a Compiler:
Before we start C programming, we need to have a compiler to compile and run our programs. There are certain online compilers like http://code.geeksforgeeks.org/, http://ideone.com/ or http://codepad.org/ that can be used to start C without installing a compiler.

Windows: There are many compilers available freely for compilation of C programs like Code Blocks and Dev-CPP. We strongly recommend Code Blocks.

Linux: For Linux, gcc comes bundled with the linux, Code Blocks can also be used with Linux.

2) Writing first program:
Following is first program in C

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

Output:

```
GeeksQuiz
```

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.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

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() { }
```

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;
}
```

```
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
```

Source: Stackoverflow

Which compiler is right?

The answer to all such questions is C standard. In all such cases we need to see what C standard says about such programs.

What is C standard?

The latest C standard is ISO/IEC 9899:2011, also known as C11 as the final draft was published in 2011. Before C11, there was C99. The C11 final draft is available here. See this for complete history of C standards.

Can we know behavior of all programs from C standard?

C standard leaves some behavior of many C constructs as undefined and some as unspecified to simplify the specification and allow some flexibility in implementation. For example, in C the use of any automatic variable before it has been initialized yields undefined behavior and order of evaluations of subexpressions is unspecified. This specifically frees the compiler to do whatever is easiest or most efficient, should such a program be submitted.

So what is the conclusion about above two examples?

Let us consider the first example which is “void main() {}”, the standard says following about prototype of main().

```
The function called at program startup is named main. The implementation
declares no prototype for this function. It shall be defined with a return
type of int and with no parameters:
      int main(void) { /* ... */ }
or with two parameters (referred to here as argc and argv, though any names
may be used, as they are local to the function in which they are declared):
      int main(int argc, char *argv[]) { /* ... */ }
or equivalent;10) or in some other implementation-defined manner.
```

So the return type void doesn’t follow the standard and it’s something allowed by certain compilers.

Let us talk about second example. Note the following statement in C standard is listed under unspecified behavior.

```
The order in which the function designator, arguments, and
```

```
subexpressions within the arguments are evaluated in a function
call (6.5.2.2).
```

What to do with programs whose behavior is undefined or unspecified in standard?
As a programmer, it is never a good idea to use programming constructs whose behaviour is undefined or unspecified, such programs should always be discouraged. The output of such programs may change with compiler and/or machine.

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
c-basics
cpp-basics

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

The definition

```
void main() { /* ... */ }
```

is not and never has been C++, nor has it even been C. See the ISO C++ standard 3.6.1[2] or the ISO C standard 5.1.2.2.1. A conforming implementation accepts

```
int main() { /* ... */ }
```

and

```
int main(int argc, char* argv[]) { /* ... */ }
```

A conforming implementation may provide more versions of main(), but they must all have return type int. The int returned by main() is a way for a program to return a value to "the system" that invokes it. On systems that doesn't provide such a facility the return value is ignored, but that doesn't make "void main()" legal C++ or legal C. ***Even if your compiler accepts "void main()" avoid it, or risk being considered ignorant by C and C++ programmers.***
In C++, main() need not contain an explicit return statement. In that case, the value returned is 0, meaning successful execution.
For example:

```
#include <iostream>
int main()
{
   std::cout << "This program returns the integer value 0\n";
}
```

Note also that neither ISO C++ nor C99 allows you to leave the type out of a declaration. That is, in contrast to C89 and ARM C++ ,"int" is not assumed where a type is missing in a declaration. Consequently:

```
#include <iostream>

main() { /* ... */ }
```

is an error because the return type of main() is missing.

Source: http://www.stroustrup.com/bs_faq2.html#void-main

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.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
c-basics
cpp-basics
cpp-functions

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

Consider the following two definitions of main().

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

and

```
int main(void)
{
  /*  */
  return 0;
}
```

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

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

Question 2

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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
c-basics
cpp-functions
cpp-main

Interesting Facts about Macros and Preprocessors in C

In a C program, all lines that start with **#** are processed by preprocessor 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 preprocessor to look in the standard folder where all header files are held. Double quotes " and " instruct the preprocessor 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
```

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
```

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
```

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

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

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
```

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
```

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
```

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
```

9) Preprocessors also support if-else directives which are typically used for conditional compilation.

```
int main()
```

```
{
#if VERBOSE >= 2
  printf("Trace Message");
#endif
}
```

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 */
```

You may like to take a Quiz on Macros and Preprocessors in C

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
c-basics
cpp-macros

Compiling a C program:- Behind the Scenes

C is a high level language and it needs a compiler to convert it into an executable code so that the program can be run on our machine.

How do we compile and run a C program?

Below are the steps we use on an Ubuntu machine with gcc compiler.

- compilation

  We first create a C program using an editor and save the file as filename.c

  ```
  $ vi filename.c
  ```

  The diagram on right shows a simple program to add two numbers.

- compil31

  Then compile it using below command.

  ```
  $ gcc –Wall filename.c –o filename
  ```

  The option -Wall enables all compiler's warning messages. This option is recommended to generate better code.
  The option -o is used to specify output file name. If we do not use this option, then an output file with name a.out is generated.

- compil21 After compilation executable is generated and we run the generated executable using below command.

  ```
  $ ./filename
  ```

What goes inside the compilation process?

Compiler converts a C program into an executable. There are four phases for a C program to become an executable:

1. Pre-processing
2. Compilation
3. Assembly
4. Linking

By executing below command, We get the all intermediate files in the current directory along with the executable.

```
$gcc –Wall –save-temps filename.c –o filename
```

The following screenshot shows all generated intermediate files.

compil4

Let us one by one see what these intermediate files contain.

Pre-processing

This is the first phase through which source code is passed. This phase include:

- Removal of Comments
- Expansion of Macros
- Expansion of the included files.

The preprocessed output is stored in the **filename.i**. Let's see what's inside filename.i: using **$vi filename.i**

compil5 compile6

In the above output, source file is filled with lots and lots of info, but at the end our code is preserved.

Analysis:

- printf contains now a + b rather than add(a, b) that's because macros have expanded.
- Comments are stripped off.
- **#include<stdio.h>** is missing instead we see lots of code. So header files has been expanded and included in our source file.

Compiling

The next step is to compile filename.i and produce an; intermediate compiled output file **filename.s**. This file is in assembly level instructions. Let’s see through this file using **$vi filename.s**

image

The snapshot shows that it is in assembly language, which assembler can understand.

Assembly

In this phase the filename.s is taken as input and turned into **filename.o** by assembler. This file contain machine level instructions. At this phase, only existing code is converted into machine language, the function calls like printf() are not resolved. Let’s view this file using **$vi filename.o**

compil7

Linking

This is the final phase in which all the linking of function calls with their definitions are done. Linker knows where all these functions are implemented. Linker does some extra work also, it adds some extra code to our program which is required when the program starts and ends. For example, there is a code which is required for setting up the environment like passing command line arguments. This task can be easily verified by using **$size filename.o** and **$size filename**. Through these commands, we know that how output file increases from an object file to an executable file. This is because of the extra code that linker adds with our program.

compil8

Note that GCC by default does dynamic linking, so printf() is dynamically linked in above program. Refer this, this and this for more details on static and dynamic linkings.

This article is contributed by **Vikash Kumar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
c-basics
cpp-basics

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;
}
```

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.

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.

Let's implement the keywords which we have discussed above. See the following code which is a working example to demonstrate these keywords:

```
#include <stdio.h>

// declaring and initializing an extern variable
extern int x = 9;

// declaring and initializing a global variable
// simply int z; would have initialized z with
// the default value of a global variable which is 0
int z=10;

// using typedef to give a short name to long long int
// very convenient to use now due to the short name
typedef long long int LL;

// function which prints square of a no. and which has void as its
// return data type
void calSquare(int arg)
{
    printf("The square of %d is %d\n",arg,arg*arg);
}

// Here void means function main takes no parameters
int main(void)
{
    // declaring a constant variable, its value cannot be modified
    const int a = 32;

    // declaring a  char variable
    char b = 'G';

    // telling the compiler that the variable z is an extern variable
    // and has been defined elsewhere (above the main function)
    extern int z;

    LL c = 1000000;

    printf("Hello World!\n");

    // printing the above variables
    printf("This is the value of the constant variable 'a': %d\n",a);
    printf("'b' is a char variable. Its value is %c\n",b);
    printf("'c' is a long long int variable. Its value is %lld\n",c);
    printf("These are the values of the extern variables 'x' and 'z'"
    " respectively: %d and %d\n",x,z);

    // value of extern variable x modified
    x=2;

    // value of extern variable z modified
    z=5;

    // printing the modified values of extern variables 'x' and 'z'
    printf("These are the modified values of the extern variables"
    " 'x' and 'z' respectively: %d and %d\n",x,z);

    // using a static variable
    printf("The value of static variable 'y' is NOT initialized to 5 after the "
           "first iteration! See for yourself :)\n");

    while (x > 0)
    {
        static int y = 5;
        y++;
        // printing value at each iteration
        printf("The value of y is %d\n",y);
        x--;
    }

    // print square of 5
    calSquare(5);

    printf("Bye! See you soon. :)\n");

    return 0;
}
```

Output:

```
Hello World
This is the value of the constant variable 'a': 32
'b' is a char variable. Its value is G
'c' is a long long int variable. Its value is 1000000
These are the values of the extern variables 'x' and 'z' respectively: 9 and 10
These are the modified values of the extern variables 'x' and 'z' respectively: 2 and 5
The value of static variable 'y' is NOT initialized to 5 after the first iteration! See for yourself :)
The value of y is 6
The value of y is 7
The square of 5 is 25
Bye! See you soon. :)
```

This article is contributed by Ayush Jaggi. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

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();
}
```

References:
http://en.wikipedia.org/wiki/Scope_%28programming%29

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
GFacts

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
```

What about functions and parameters passed to functions?
A function itself is a block. Parameters and other local variables of a function follow the same block scope rules.

*Can variables of block be accessed in another subsequent block?**
No, a variabled declared in a block can only be accessed inside the block and all inner blocks of this block. For example, following program produces compiler error.

```
int main()
{
  {
      int x = 10;
  }
  {
      printf("%d", x);  // Error: x is not accessible here
  }
  return 0;
}
```

Output:

```
error: 'x' undeclared (first use in this function)
```

As an exercise, predict the output of following program.

```
int main()
{
  int x = 1, y = 2, z = 3;
  printf(" x = %d, y = %d, z = %d \n", x, y, z);
  {
       int x = 10;
       float y = 20;
       printf(" x = %d, y = %f, z = %d \n", x, y, z);
       {
             int z = 100;
             printf(" x = %d, y = %f, z = %d \n", x, y, z);
       }
  }
  return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

How Linkers Resolve Global Symbols Defined at Multiple Places?

At compile time, the compiler exports each global symbol to the assembler as either strong or weak, and the assembler encodes this information implicitly in the symbol table of the relocatable object file. Functions and initialized global variables get strong symbols. Uninitialized global variables get weak symbols.
For the following example programs, *buf, bufp0, main,* and *swap* are strong symbols; *bufp1* is a weak symbol.

```
/* main.c */
void swap();
int buf[2] = {1, 2};
int main()
{
  swap();
  return 0;
}

/* swap.c */
extern int buf[];

int *bufp0 = &buf[0];
int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

Given this notion of strong and weak symbols, Unix linkers use the following rules for dealing with multiply defined symbols:

Rule 1: Multiple strong symbols are not allowed.
Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol.
Rule 3: Given multiple weak symbols, choose any of the weak symbols.

For example, suppose we attempt to compile and link the following two C modules:

```
/* foo1.c */
int main()
{
  return 0;
}

/* bar1.c */
int main()
{
  return 0;
}
```

In this case, the linker will generate an error message because the strong symbol *main* is defined multiple times (rule 1):

unix> gcc foo1.c bar1.c

/tmp/cca015022.o: In function 'main':

/tmp/cca015022.o(.text+0x0): multiple definition of 'main'

/tmp/cca015021.o(.text+0x0): first defined here

Similarly, the linker will generate an error message for the following modules because the strong symbol *x* is defined twice (rule 1):

```
/* foo2.c */
int x = 15213;
int main()
{
  return 0;
}

/* bar2.c */
int x = 15213;
void f()
{
}
```

However, if *x* is uninitialized in one module, then the linker will quietly choose the strong symbol defined in the other (rule 2) as is the case in following program:

```
/* foo3.c */
#include <stdio.h>
void f(void);
int x = 15213;
int main()
{
  f();
  printf("x = %d\n", x);
  return 0;
}

/* bar3.c */
int x;
void f()
{
  x = 15212;
}
```

At run time, function f() changes the value of x from 15213 to 15212, which might come as a unwelcome surprise to the author of function main! Notice that the linker normally gives no indication that it has detected multiple definitions of x.

```
unix> gcc -o foobar3 foo3.c bar3.c
unix> ./foobar3
x = 15212
```

The same thing can happen if there are two weak definitions of x (rule 3).

See below source link for more detailed explanation and more examples.

Source:
http://csapp.cs.cmu.edu/public/ch7-preview.pdf

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

Question 1
Consider the following two C lines

```
int var1;
extern int var2;
```

Which of the following statements is correct
ABoth statements only declare variables, don't define them.

BFirst statement declares and defines var1, but second statement only declares var2
CBoth statements declare define variables var1 and var2
Variable Declaration and Scope
Discuss it
Question 1 Explanation:
See Understanding “extern” keyword in C
Question 2
Predict the output

```
#include <stdio.h>
int var = 20;
int main()
{
   int var = var;
   printf("%d ", var);
   return 0;
}
```

AGarbage Value
B20
CCompiler Error
Variable Declaration and Scope
Discuss it
Question 2 Explanation:
First var is declared, then value is assigned to it. As soon as var is declared as a local variable, it hides the global variable var.
Question 3

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

ACompiler Error: var is not defined
B20
C0
Variable Declaration and Scope
Discuss it
Question 3 Explanation:
var is only declared and not defined (no memory allocated for it) Refer: Understanding “extern” keyword in C
Question 4

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

A10
BCompiler Error: var is not defined
C0
Variable Declaration and Scope
Discuss it
Question 4 Explanation:
If a variable is only declared and an initializer is also provided with that declaration, then the memory for that variable will be allocated i.e. that variable will be considered as defined. Refer: Understanding “extern” keyword in C
Question 5
Output?

```
int main()
{
  {
     int var = 10;
  }
  {
    printf("%d", var);
  }
  return 0;
}
```

A10
BCompiler Errror
CGarbage Value
Variable Declaration and Scope
Discuss it
Question 5 Explanation:
x is not accessible. The curly brackets define a block of scope. Anything declared between curly brackets goes out of scope after the closing bracket.
Question 6
Output?

```
#include <stdio.h>
int main()
{
  int x = 1, y = 2, z = 3;
  printf(" x = %d, y = %d, z = %d \n", x, y, z);
  {
      int x = 10;
      float y = 20;
      printf(" x = %d, y = %f, z = %d \n", x, y, z);
      {
            int z = 100;
            printf(" x = %d, y = %f, z = %d \n", x, y, z);
      }
  }
  return 0;
}
```

A
```
x = 1, y = 2, z = 3
x = 10, y = 20.000000, z = 3
x = 1, y = 2, z = 100
```

BCompiler Error
C
```
x = 1, y = 2, z = 3
x = 10, y = 20.000000, z = 3
x = 10, y = 20.000000, z = 100
```

D
```
x = 1, y = 2, z = 3
x = 1, y = 2, z = 3
x = 1, y = 2, z = 3
```

Variable Declaration and Scope
Discuss it
Question 6 Explanation:
See Scope rules in C
Question 7

```
int main()
{
  int x = 032;
  printf("%d", x);
  return 0;
}
```

A32
B0
C26
D50
Variable Declaration and Scope
Discuss it
Question 7 Explanation:
When a constant value starts with 0, it is considered as octal number. Therefore the value of x is 3*8 + 2 = 26
Question 8
Consider the following C program, which variable has the longest scope?

```
int a;
int main()
{
  int b;
  // ..
  // ..
}
int c;
```

Aa
Bb
Cc
DAll have same scope
Variable Declaration and Scope
Discuss it
Question 8 Explanation:
a is accessible everywhere. b is limited to main() c is accessible only after its declaration.
Question 9
Consider the following variable declarations and definitions in C

```
i) int var_9 = 1;
ii) int 9_var = 2;
iii) int _ = 3;
```

Choose the correct statement w.r.t. above variables.
ABoth i) and iii) are valid.
BOnly i) is valid.
CBoth i) and ii) are valid.
DAll are valid.
Variable Declaration and Scope **C Quiz - 101**
Discuss it
Question 9 Explanation:
In C language, a variable name can consists of letters, digits and underscore i.e. _ . But a variable name has to start with either letter or underscore. It can't start with a digit. So valid variables are var_9 and _ from the above question. Even two back to back underscore i.e. __ is also a valid variable name. Even _9 is a valid variable. But 9var and 9_ are invalid variables in C. This will be caught at the time of compilation itself. That's why the correct answer is A).
Question 10
Find out the correct statement for the following program.

```
#include "stdio.h"

int * gPtr;

int main()
{
 int * lPtr = NULL;

 if(gPtr == lPtr)
 {
   printf("Equal!");
 }
 else
 {
  printf("Not Equal");
 }

 return 0;
}
```

AIt'll always print Equal.
BIt'll always print Not Equal.
CSince gPtr isn't initialized in the program, it'll print sometimes Equal and at other times Not Equal.
Variable Declaration and Scope **C Quiz - 109**
Discuss it
Question 10 Explanation:
It should be noted that global variables such gPtr (which is a global pointer to int) are initialized to ZERO. That's why gPtr (which is a global pointer and initialized implicitly) and lPtr (which a is local pointer and initialized explicitly) would have same value i.e. correct answer is a.
Question 11
Consider the program below in a hypothetical language which allows global variable and a choice of call by reference or call by value methods of parameter passing.

```
 int i ;
program main ()
{
   int j = 60;
   i = 50;
   call f (i, j);
   print i, j;
}
procedure f (x, y)
{
   i = 100;
   x = 10;
   y = y + i ;
}
```

Which one of the following options represents the correct output of the program for the two parameter passing mechanisms?
ACall by value : i = 70, j = 10; Call by reference : i = 60, j = 70
BCall by value : i = 50, j = 60; Call by reference : i = 50, j = 70
CCall by value : i = 10, j = 70; Call by reference : i = 100, j = 60
DCall by value : i = 100, j = 60; Call by reference : i = 10, j = 70
Variable Declaration and Scope   C Quiz - 113   Gate IT 2007
Discuss it
Question 11 Explanation:
Call by value: A copy of parameters will be passed and whatever updations are performed will be valid only for that copy, leaving original values intact.
Call by reference: A link to original variables will be passed, by allowing the function to manipulate the original variables.
There are 11 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Complicated declarations in C

Most of the times declarations are simple to read, but it is hard to read some declarations which involve pointer to functions. For example, consider the following declaration from “signal.h”.

```
void (*bsd_signal(int, void (*)(int)))(int);
```

Let us see the steps to read complicated declarations.

1) Convert C declaration to postfix format and read from left to right.
2) To convert experssion to postfix, start from innermost parenthesis, If innermost parenthesis is not present then start from declarations name and go right first. When first ending parenthesis encounters then go left. Once whole parenthesis is parsed then come out from parenthesis.
3) Continue until complete declaration has been parsed.

Let us start with simple example. Below examples are from “K & R” book.

```
1)  int (*fp) ();
```

Let us convert above expression to postfix format. For the above example, there is no innermost parenthesis, that’s why, we will print declaration name i.e. “fp”. Next step is, go to right side of expression, but there is nothing on right side of “fp” to parse, that’s why go to left side. On left side we found “*”, now print “*” and come out of parenthesis. We will get postfix expression as below.

```
fp  *  ()  int
```

Now read postfix expression from left to right. e.g. fp is pointer to function returning int

Let us see some more examples.

```
2) int (*daytab)[13]
```

Postfix : daytab * [13] int

Meaning : daytab is pointer to array of 13 integers.

```
3) void (*f[10]) (int, int)
```

Postfix : f[10] * (int, int) void
Meaning : f is an array of 10 of pointer to function(which takes 2 arguments of type int) returning void

```
4) char (*(*x())[]) ()
```

Postfix : x () * [] * () char
Meaning : x is a function returning pointer to array of pointers to function returnging char

```
5) char (*(*x[3])())[5]
```

Postfix : x[3] * () * [5] char
Meaning : x is an array of 3 pointers to function returning pointer to array of 5 char's

```
6) int *(*(*arr[5])()) ()
```

Postfix : arr[5] * () * () * int
Meaning : arr is an array of 5 pointers to functions returning pointer to function returning pointer to integer

```
7) void (*bsd_signal(int sig, void (*func)(int)))(int);
```

Postfix : bsd_signal(int sig, void(*func)(int)) * (int) void
Meaning : bsd_signal is a function that takes integer & a pointer to a function(that takes integer as argument and returns void) and returns pointer to a function(that take integer as argument and returns void)

This article is compiled by "Narendra Kangralkar" and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
pointer

Redeclaration of global variable in C

Consider the below two programs:

```
// Program 1
int main()
{
  int x;
  int x = 5;
  printf("%d", x);
  return 0;
}
```

Output in C:

```
redeclaration of 'x' with no linkage
```

```
// Program 2
int x;
int x = 5;

int main()
{
  printf("%d", x);
  return 0;
}
```

Output in C:

```
5
```

In C, the first program fails in compilation, but second program works fine. In C++, both programs fail in compilation.

C allows a global variable to be declared again when first declaration doesn't initialize the variable.

The below program fails in both C also as the global variable is initialized in first declaration itself.

```
int x = 5;
int x = 10;

int main()
{
  printf("%d", x);
  return 0;
}
```

Output:

```
error: redefinition of 'x'
```

This article is contributed **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-storage-classes

Data Types in C

Each variable in C has an associated data type. Each data type requires different amounts of memory and has some specific operations which can be performed over it. Let us briefly describe them one by one:

Following are the examples of some very common data types used in C:

char: The most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.
int: As the name suggests, an int variable is used to store an integer.
float: It is used to store decimal numbers (numbers with floating point value) with single precision.
double: It is used to store decimal numbers (numbers with floating point value) with double precision.

Different data types also have different ranges upto which they can store numbers. These ranges may vary from compiler to compiler. Below is list of ranges along with the memory requirement and format specifiers on 32 bit gcc compiler.

```
Data Type           Memory (bytes)         Range                   Format Specifier
short int                2         -32,768 to 32,767                    %hd
unsigned short int        2          0 to 65,535                        %hu
unsigned int             4          0 to 4,294,967,295                  %u
int                    4         -2,147,483,648 to 2,147,483,647        %d
long int                4         -2,147,483,648 to 2,147,483,647        %ld
unsigned long int         4          0 to 4,294,967,295                  %lu
long long int            8         -(2^63) to (2^63)-1                  %lld
unsigned long long int     8          0 to 18,446,744,073,709,551,615       %llu
signed char              1         -128 to 127                          %c
unsigned char             1          0 to 255                            %c
float                   4                                        %f
double                   8                                        %lf
long double              12                                         %Lf
```

We can use the sizeof() operator to check the size of a variable. See the following C program for the usage of the various data types:

```
#include <stdio.h>
int main()
{
   int a = 1;
   char b ='G';
```

```
    double c = 3.14;
    printf("Hello World!\n");

    //printing the variables defined above along with their sizes
    printf("Hello! I am a character. My value is %c and "
          "my size is %lu byte.\n", b,sizeof(char));
    //can use sizeof(b) above as well

    printf("Hello! I am an integer. My value is %d and "
          "my size is %lu  bytes.\n", a,sizeof(int));
    //can use sizeof(a) above as well

    printf("Hello! I am a double floating point variable."
          " My value is %lf and my size is %lu bytes.\n",c,sizeof(double));
    //can use sizeof(c) above as well

    printf("Bye! See you soon. :)\n");

    return 0;
}
```

Output:

```
Hello World!
Hello! I am a character. My value is G and my size is 1 byte.
Hello! I am an integer. My value is 1 and my size is 4  bytes.
Hello! I am a double floating point variable. My value is 3.140000 and my size i
s 8 bytes.
Bye! See you soon. :)
```

Quiz on Data Types in C

This article is contributed by Ayush Jaggi. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C

Use of bool in C

The C99 standard for C language supports bool variables. Unlike C++, where no header file is needed to use bool, a header file “stdbool.h” must be included to use bool in C. If we save the below program as .c, it will not compile, but if we save it as .cpp, it will work fine.

```
int main()
{
  bool arr[2] = {true, false};
  return 0;
}
```

If we include the header file “stdbool.h” in the above program, it will work fine as a C program.

```
#include <stdbool.h>
int main()
{
  bool arr[2] = {true, false};
  return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

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".

We will soon be discussing integer conversion rules between signed and unsigned, int and long int, etc.

References:

http://www.tru64unix.compaq.com/docs/base_doc/DOCUMENTATION/V40F_HTML/AQTLTBTE/DOCU_067.HTM

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-data-types

Question 1
Predict the output of following program. Assume that the numbers are stored in 2's complement form.

```
#include<stdio.h>
int  main()
{
   unsigned int x = -1;
   int y = ~0;
   if (x == y)
      printf("same");
   else
      printf("not same");
   return 0;
}
```

Asame
Bnot same
Data Types
Discuss it
Question 1 Explanation:
-1 and ~0 essentially have same bit pattern, hence x and y must be same. In the comparison, y is promoted to unsigned and compared against x (See this for promotion rules). The result is “same”. However, when interpreted as signed and unsigned their numerical values will differ. x is MAXUNIT and y is -1. Since we have %u for y also, the output will be MAXUNIT and MAXUNIT.
Question 2
Which of the following is not a valid declaration in C?

```
1. short int x;
```

```
2. signed short x;
```

```
3. short x;
```

```
4. unsigned short x;
```

A3 and 4
B2
C1
DAll are valid
Data Types
Discuss it
Question 2 Explanation:
All are valid. First 3 mean the same thing. 4th means unsigned.
Question 3
Predict the output

```
#include <stdio.h>

int main()
{
   float c = 5.0;
   printf ("Temperature in Fahrenheit is %.2f", (9/5)*c + 32);
   return 0;
}
```

ATemperature in Fahrenheit is 41.00
BTemperature in Fahrenheit is 37.00

CTemperature in Fahrenheit is 0.00
DCompiler Error
Data Types
Discuss it
Question 3 Explanation:
Since 9 and 5 are integers, integer arithmetic happens in subexpression (9/5) and we get 1 as its value. To fix the above program, we can use 9.0 instead of 9 or 5.0 instead of 5 so that floating point arithmetic happens. 1
Question 4
Predict the output of following C program

```
#include <stdio.h>
int main()
{
   char a = '\012';

   printf("%d", a);

   return 0;
}
```

ACompiler Error
B12
C10
DEmpty
Data Types
Discuss it
Question 4 Explanation:
The value '\012' means the character with value 12 in octal, which is decimal 10
Question 5
In C, sizes of an integer and a pointer must be same.
ATrue
BFalse
Data Types
Discuss it
Question 5 Explanation:
Sizes of integer and pointer are compiler dependent. The both sizes need not be same.
Question 6
Output?

```
int main()
{
   void *vptr, v;
   v = 0;
   vptr = &v;
   printf("%v", *vptr);
   getchar();
   return 0;
}
```

A0
BCompiler Error
CGarbage Value
Data Types
Discuss it
Question 6 Explanation:
void is not a valid type for declaring variables. void * is valid though.
Question 7
Assume that the size of char is 1 byte and negatives are stored in 2's complement form

```
#include<stdio.h>
int main()
{
   char c = 125;
   c = c+10;
   printf("%d", c);
   return 0;
}
```

A135
B+INF
C-121

D-8
Data Types
Discuss it
Question 8

```
#include <stdio.h>
int main()
{
    if (sizeof(int) > -1)
        printf("Yes");
    else
        printf("No");
    return 0;
}
```

AYes
BNo
CCompiler Error
DRuntime Error
Data Types
Discuss it
Question 8 Explanation:
In C, when an intger value is compared with an unsigned it, the int is promoted to unsigned. Negative numbers are stored in 2's complement form and unsigned value of the 2's complement form is much higher than the sizeof int.
Question 9
Suppose n and p are unsigned int variables in a C program. We wish to set p to nC3. If n is large, which of the following statements is most likely to set p correctly?
Ap = n * (n-1) * (n-2) / 6;
Bp = n * (n-1) / 2 * (n-2) / 3;
Cp = n * (n-1) / 3 * (n-2) / 2;
Dp = n * (n-1) * (n-2) / 6.0;
Data Types **GATE-CS-2014-(Set-2)**
Discuss it
Question 9 Explanation:
As n is large, the product n*(n-1)*(n-2) will go out of the range(overflow) and it will return a value different from what is expected. So we consider a shorter product n*(n-1). n*(n-1) is always an even number. So the subexpression "n * (n-1) / 2 " in option B would always produce an integer, which means no precision loss in this subexpression. And when we consider `n*(n-1)/2*(n-2)`, it will always give a number which is a multiple of 3. So dividing it with 3 won't have any loss.
Question 10
Output of following program?

```
#include<stdio.h>
int main()
{
    float x = 0.1;
    if ( x == 0.1 )
        printf("IF");
    else if (x == 0.1f)
        printf("ELSE IF");
    else
        printf("ELSE");
}
```

AELSE IF
BIF
CELSE
Data Types
Discuss it
Question 11
Suppose a C program has floating constant 1.414, what's the best way to convert this as "float" data type?
A(float)1.414
Bfloat(1.414)
C1.414f or 1.414F
D1.414 itself of "float" data type i.e. nothing else required.
Data Types **C Quiz - 102**
Discuss it
Question 11 Explanation:
By default floating constant is of double data type. By suffixing it with f or F, it can be converted to float data type. For more details, this post can be referred here.

Question 12
In a C program, following variables are defined:

```
float     x = 2.17;
double   y = 2.17;
long double z = 2.17;
```

Which of the following is correct way for printing these variables via printf.
Aprintf("%f %lf %Lf",x,y,z);
Bprintf(“%f %f %f”,x,y,z);
Cprintf("%f %ff %fff",x,y,z);
Dprintf("%f %lf %llf",x,y,z);
Data Types   C Quiz - 106
Discuss it
Question 12 Explanation:
In C language, float, double and long double are called real data types. For “float”, “double” and “long double”, the right format specifiers are %f, %lf and %Lf from the above options. It should be noted that C standard has specified other format specifiers as well for real types which are %g, %e etc.
Question 13
“typedef” in C basically works as an alias. Which of the following is correct for “typedef”?
Atypedef can be used to alias compound data types such as struct and union.
Btypedef can be used to alias both compound data types and pointer to these compound types.
Ctypedef can be used to alias a function pointer.
Dtypedef can be used to alias an array.
EAll of the above.
Data Types   C Quiz - 106
Discuss it
Question 13 Explanation:
In C, typedef can be used to alias or make synonyms of other types. Since function pointer is also a type, typedef can be used here. Since, array is also a type of symmetric data types, typedef can be used to alias array as well.
Question 14
In a C program snippet, followings are used for definition of Integer variables?

```
signed s;
unsigned u;
long l;
long long ll;
```

Pick the best statement for these.
AAll of the above variable definitions are incorrect because basic data type int is missing.
BAll of the above variable definitions are correct because int is implicitly assumed in all of these.
COnly “long l;” and “long long ll;” are valid definitions of variables.
DOnly “unsigned u;” is valid definition of variable.
Data Types   C Quiz - 107
Discuss it
Question 14 Explanation:
Please note that *signed*, *unsigned* and *long* all three are Type specifiers. And *int* is implicitly assumed in all of three. As per C standard, “int, signed, or signed int” are equivalent. Similarly, “unsigned, or unsigned int” are equivalent. Besides, “long, signed long, long int, or signed long int” are all equivalent. And “long long, signed long long, long long int, or signed long long int“ are equivalent.
There are 14 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Comparison of a float with a value in C

Predict the output of following C program.

```
#include<stdio.h>
int main()
{
   float x = 0.1;
   if (x == 0.1)
      printf("IF");
   else if (x == 0.1f)
```

```
        printf("ELSE IF");
    else
        printf("ELSE");
}
```

The output of above program is “***ELSE IF***” which means the expression “x == 0.1” returns false and expression “x == 0.1f” returns true.

Let consider the of following program to understand the reason behind the above output.

```
#include<stdio.h>
int main()
{
   float x = 0.1;
   printf("%d %d %d", sizeof(x), sizeof(0.1), sizeof(0.1f));
   return 0;
}
```

The output of above program is “***4 8 4***” on a typical C compiler. It actually prints size of float, size of double and size of float.

The values used in an expression are considered as double (double precision floating point format) unless a ‘f’ is specified at the end. So the expression “x==0.1” has a double on right side and float which are stored in a single precision floating point format on left side. In such situations float is promoted to double (see this). The double precision format uses uses more bits for precision than single precision format. Note that the promotion of float to double can only cause mismatch when a value (like 0.1) uses more precision bits than the bits of single precision. For example, the following C program prints “IF”.

```
#include<stdio.h>
int main()
{
    float x = 0.5;
    if (x == 0.5)
        printf("IF");
    else if (x == 0.5f)
        printf("ELSE IF");
    else
        printf("ELSE");
}
```

Output:

```
IF
```

You can refer Floating Point Representation – Basics for representation of floating point numbers.

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-data-types

Storage Classes in C

Storage Classes are used to describe about the features of a variable/function. These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program.

C language uses 4 storage classes, namely:

auto: This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope). Of course, these can be accessed within nested blocks within the parent block/function in which the auto variable was declared. However, they can be accessed outside their scope as well using the concept of pointers given here by pointing to the very exact memory location where the variables resides. They are assigned a garbage value by default whenever they are declared.

extern: Extern storage class 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 be 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. For more information on how extern variables work, have a look at this link.

static: This storage class 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.

register: This storage class declares register variables which have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available. This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program. If a free register is not available, these are then stored in the memory only. Usually few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program. An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

To specify the storage class for a variable, the following syntax is to be followed:

Syntax:

```
storage_class var_data_type var_name;
```

Functions follow the same syntax as given above for variables. Have a look at the following C example for further clarification:

```
// A C program to demonstrate different storage
// classes
#include <stdio.h>

// declaring and initializing an extern variable
extern int x = 9;

// declaring and initialing a global variable z
// simply int z; would have initialized z with
// the default value of a global variable which is 0
int z = 10;

int main()
{
    // declaring an auto variable (simply
    // writing "int a=32;" works as well)
    auto int a = 32;

    // declaring a register variable
    register char b = 'G';

    // telling the compiler that the variable
    // x is an extern variable and has been
    // defined elsewhere (above the main
    // function)
    extern int z;

    printf("Hello World!\n");

    // printing the auto variable 'a'
    printf("\nThis is the value of the auto "
           " integer 'a': %d\n",a);

    // printing the extern variables 'x'
    // and 'z'
    printf("\nThese are the values of the"
           " extern integers 'x' and 'z'"
           " respectively: %d and %d\n", x, z);

    // printing the register variable 'b'
    printf("\nThis is the value of the "
        "register character 'b': %c\n",b);

    // value of extern variable x modified
    x = 2;

    // value of extern variable z modified
    z = 5;

    // printing the modified values of
    // extern variables 'x' and 'z'
    printf("\nThese are the modified values "
        "of the extern integers 'x' and "
        "'z' respectively: %d and %d\n",x,z);

    // using a static variable 'y'
    printf("\n'y' is a static variable and its "
        "value is NOT initialized to 5 after"
        " the first iteration! See for"
        " yourself :)\n");

    while (x > 0)
    {
        static int y = 5;
        y++;

        // printing value of y at each iteration
        printf("The value of y is %d\n",y);
        x--;
    }

    // exiting
    printf("\nBye! See you soon. :)\n");

    return 0;
}
```

Output:

```
Hello World!

This is the value of the auto  integer 'a': 32

These are the values of the extern integers 'x' and 'z'
respectively: 9 and 10

This is the value of the register character 'b': G

These are the modified values of the extern integers 'x'
and 'z' respectively: 2 and 5

'y' is a static variable and its value is NOT initialized
to 5 after the first iteration! See for yourself :)
The value of y is 6
The value of y is 7

Bye! See you soon. :)
```

Quiz on Storage Classes

This article is contributed by Ayush Jaggi. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C

Static Variables in C

Static variables have a property of preserving their value even after they are out of their scope!Hence, static variables preserve their previous value in their previous scope and are not initialized again in the new scope.
Syntax:

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

Following are some interesting facts about static variables in C.

1) A static int variable remains in memory while the program is running. A normal or auto variable is destroyed when a function call where the variable was declared is over.

For example, we can use static int to count number of times a function is called, but an auto variable can’t be sued for this purpose.

For example below program prints “1 2″

```
#include<stdio.h>
int fun()
{
  static int count = 0;
  count++;
  return count;
}

int main()
{
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}
```

Output:

```
1 2
```

But below program prints 1 1

```
#include<stdio.h>
int fun()
{
  int count = 0;
  count++;
  return count;
}

int main()
{
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}
```

Output:

```
1 1
```

2) Static variables are allocated memory in data segment, not stack segment. See memory layout of C programs for details.

3) Static variables (like global variables) are initialized as 0 if not initialized explicitly. For example in the below program, value of x is printed as 0, while value of y is something garbage. See this for more details.

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

```
int main()
{
    static int x;
    int y;
    printf("%d \n %d", x, y);
}
```

Output:

```
0
[some_garbage_value]
```

4) In C, static variables can only be initialized using constant literals. For example, following program fails in compilation. See this for more details.

```
#include<stdio.h>
int initializer(void)
{
    return 50;
}

int main()
{
    static int i = initializer();
    printf(" value of i = %d", i);
    getchar();
    return 0;
}
```

Output

```
 In function 'main':
9:5: error: initializer element is not constant
     static int i = initializer();
     ^
```

Please note that this condition doesn't hold in C++. So if you save the program as a C++ program, it would compile \and run fine.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C

Understanding "extern" keyword in C

I'm sure that this post will be as interesting and informative to C virgins (i.e. beginners) as it will be to those who are well versed in C. So let me start with saying that extern keyword applies to C variables (data objects) and C functions. Basically extern keyword extends the visibility of the C variables and C functions. Probably that's is the reason why it was named as extern.

Though (almost) everyone knows the meaning of declaration and definition of a variable/function yet for the sake of completeness of this post, I would like to clarify them. Declaration of a variable/function simply declares that the variable/function exists somewhere in the program but the memory is not allocated for them. But the declaration of a variable/function serves an important role. And that is the type of the variable/function. Therefore, when a variable is declared, the program knows the data type of that variable. In case of function declaration, the program knows what are the arguments to that functions, their data types, the order of arguments and the return type of the function. So that's all about declaration. Coming to the definition, when we define a variable/function, apart from the role of declaration, it also allocates memory for that variable/function. Therefore, we can think of definition as a super set of declaration. (or declaration as a subset of definition). From this explanation, it should be obvious that a variable/function can be declared any number of times but it can be defined only once. (Remember the basic principle that you can't have two locations of the same variable/function). So that's all about declaration and definition.

Now coming back to our main objective: Understanding “extern” keyword in C. I’ve explained the role of declaration/definition because it’s mandatory to understand them to understand the “extern” keyword. Let us first take the easy case. Use of extern with C functions. By default, the declaration and definition of a C function have “extern” prepended with them. It means even though we don’t use extern with the declaration/definition of C functions, it is present there. For example, when we write.

```
int foo(int arg1, char arg2);
```

There’s an extern present in the beginning which is hidden and the compiler treats it as below.

```
extern int foo(int arg1, char arg2);
```

Same is the case with the definition of a C function (Definition of a C function means writing the body of the function). Therefore whenever we define a C function, an extern is present there in the beginning of the function definition. Since the declaration can be done any number of times and definition can be done only once, we can notice that declaration of a function can be added in several C/H files or in a single C/H file several times. But we notice the actual definition of the function only once (i.e. in one file only). And as the extern extends the visibility to the whole program, the functions can be used (called) anywhere in any of the files of the whole program provided the declaration of the function is known. (By knowing the declaration of the function, C compiler knows that the definition of the function exists and it goes ahead to compile the program). So that’s all about extern with C functions.

Now let us the take the second and final case i.e. use of extern with C variables. I feel that it more interesting and information than the previous case where extern is present by default with C functions. So let me ask the question, how would you declare a C variable without defining it? Many of you would see it trivial but it’s important question to understand extern with C variables. The answer goes as follows.

```
extern int var;
```

Here, an integer type variable called var has been declared (remember no definition i.e. no memory allocation for var so far). And we can do this declaration as many times as needed. (remember that declaration can be done any number of times) So far so good.

Now how would you define a variable. Now I agree that it is the most trivial question in programming and the answer is as follows.

```
int var;
```

Here, an integer type variable called var has been declared as well as defined. (remember that definition is the super set of declaration). Here the memory for var is also allocated. Now here comes the surprise, when we declared/defined a C function, we saw that an extern was present by default. While defining a function, we can prepend it with extern without any issues. But it is not the case with C variables. If we put the presence of extern in variable as default then the memory for them will not be allocated ever, they will be declared only. Therefore, we put extern explicitly for C variables when we want to declare them without defining them. Also, as the extern extends the visibility to the whole program, by externing a variable we can use the variables anywhere in the program provided we know the declaration of them and the variable is defined somewhere.

Now let us try to understand extern with examples.

Example 1:

```
int var;
int main(void)
{
  var = 10;
  return 0;
}
```

Analysis: This program is compiled successfully. Here var is defined (and declared implicitly) globally.

Example 2:

```
extern int var;
int main(void)
{
 return 0;
}
```

Analysis: This program is compiled successfully. Here var is declared only. Notice var is never used so no problems.

Example 3:

```
extern int var;
int main(void)
{
 var = 10;
 return 0;
}
```

Analysis: This program throws error in compilation. Because var is declared but not defined anywhere. Essentially, the var isn't allocated any memory. And the program is trying to change the value to 10 of a variable that doesn't exist at all.

Example 4:

```
#include "somefile.h"
extern int var;
int main(void)
{
 var = 10;
 return 0;
}
```

Analysis: Supposing that somefile.h has the definition of var. This program will be compiled successfully.

Example 5:

```
extern int var = 0;
int main(void)
{
 var = 10;
 return 0;
}
```

Analysis: Guess this program will work? Well, here comes another surprise from C standards. They say that..if a variable is only declared and an initializer is also provided with that declaration, then the memory for that variable will be allocated i.e. that variable will be considered as defined. Therefore, as per the C standard, this program will compile successfully and work.

So that was a preliminary look at "extern" keyword in C.

I'm sure that you want to have some take away from the reading of this post. And I would not disappoint you.
In short, we can say

1. Declaration can be done any number of times but definition only once.
2. "extern" keyword is used to extend the visibility of variables/functions().
3. Since functions are visible through out the program by default. The use of extern is not needed in function declaration/definition. Its use is redundant.
4. When extern is used with a variable, it's only declared not defined.
5. As an exception, when an extern variable is declared with initialization, it is taken as definition of the variable as well.

What are the default values of static variables in C?

In C, if an object that has static storage duration is not initialized explicitly, then:
— if it has pointer type, it is initialized to a NULL pointer;
— if it has arithmetic type, it is initialized to (positive or unsigned) zero;
— if it is an aggregate, every member is initialized (recursively) according to these rules;
— if it is a union, the first named member is initialized (recursively) according to these rules.

For example, following program prints:
Value of g = 0
Value of sg = 0
Value of s = 0

```
#include<stdio.h>
int g;  //g = 0, global objects have static storage duration
static int gs; //gs = 0, global static objects have static storage duration
int main()
{
  static int s; //s = 0, static objects have static storage duration
  printf("Value of g = %d", g);
  printf("\nValue of gs = %d", gs);
  printf("\nValue of s = %d", s);

  getchar();
  return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:
The C99 standard

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

Understanding “volatile” qualifier in C | Set 2 (Examples)

The volatile keyword is intended to prevent the compiler from applying any optimizations on objects that can change in ways that cannot be determined by the compiler.

Objects declared as volatile are omitted from optimization because their values can be changed by code outside the scope of current code at any time. The system always reads the current value of a volatile object from the memory location rather than keeping its value in temporary register at the point it is requested, even if a previous instruction asked for a value from the same object. So the simple question is, how can value of a variable change in such a way that compiler cannot predict. Consider the following cases for answer to this question.

1) Global variables modified by an interrupt service routine outside the scope: For example, a global variable can represent a data port (usually global pointer referred as memory mapped IO) which will be updated dynamically. The code reading data port must be declared as volatile in order to fetch latest data available at the port. Failing to declare variable as volatile, the compiler will optimize the code in such a way that it will read the port only once and keeps using the same value in a temporary register to speed up the program (speed optimization). In general, an ISR used to update these data port when there is an interrupt due to availability of new data

2) Global variables within a multi-threaded application: There are multiple ways for threads communication, viz, message passing, shared memory, mail boxes, etc. A global variable is weak form of shared memory. When two threads sharing information via global variable, they need to be qualified with volatile. Since threads run asynchronously, any update of global variable due to one thread should be fetched freshly by another consumer thread. Compiler can read the global variable and can place them in temporary variable of current thread context. To nullify the effect of compiler optimizations, such global variables to be qualified as volatile

If we do not use volatile qualifier, the following problems may arise
1) Code may not work as expected when optimization is turned on.
2) Code may not work as expected when interrupts are enabled and used.

Let us see an example to understand how compilers interpret volatile keyword. Consider below code, we are changing value of const object using pointer and we are compiling code without optimization option. Hence compiler won’t do any optimization and will change value of const object.

```
/* Compile code without optimization option */
#include <stdio.h>
int main(void)
{
    const int local = 10;
    int *ptr = (int*) &local;

    printf("Initial value of local : %d \n", local);

    *ptr = 100;

    printf("Modified value of local: %d \n", local);
    return 0;
}
```

When we compile code with "–save-temps" option of gcc it generates 3 output files

1) preprocessed code (having .i extention)
2) assembly code (having .s extention) and
3) object code (having .o option).

We compile code without optimization, that's why the size of assembly code will be larger (which is highlighted in red color below).

Output:

```
[narendra@ubuntu]$ gcc volatile.c -o volatile –save-temps
[narendra@ubuntu]$ ./volatile
Initial value of local : 10
Modified value of local: 100
[narendra@ubuntu]$ ls -l volatile.s
-rw-r–r– 1 narendra narendra 731 2016-11-19 16:19 volatile.s
[narendra@ubuntu]$
```

Let us compile same code with optimization option (i.e. -O option). In thr below code, "local" is declared as const (and non-volatile), GCC compiler does optimization and ignores the instructions which try to change value of const object. Hence value of const object remains same.

```
/* Compile code with optimization option */
#include <stdio.h>

int main(void)
{
    const int local = 10;
    int *ptr = (int*) &local;

    printf("Initial value of local : %d \n", local);

    *ptr = 100;

    printf("Modified value of local: %d \n", local);

    return 0;
}
```

For above code, compiler does optimization, that's why the size of assembly code will reduce.

Output:

```
[narendra@ubuntu]$ gcc -O3 volatile.c -o volatile –save-temps
[narendra@ubuntu]$ ./volatile
Initial value of local : 10
Modified value of local: 10
[narendra@ubuntu]$ ls -l volatile.s
-rw-r–r– 1 narendra narendra 626 2016-11-19 16:21 volatile.s
```

Let us declare const object as volatile and compile code with optimization option. Although we compile code with optimization option, value of const object will change, because variable is declared as volatile that means don't do any optimization.

```
/* Compile code with optimization option */
#include <stdio.h>

int main(void)
{
    const volatile int local = 10;
    int *ptr = (int*) &local;

    printf("Initial value of local : %d \n", local);

    *ptr = 100;

    printf("Modified value of local: %d \n", local);
    return 0;
}
```

Output:

```
[narendra@ubuntu]$ gcc -O3 volatile.c -o volatile –save-temp
[narendra@ubuntu]$ ./volatile
Initial value of local : 10
Modified value of local: 100
[narendra@ubuntu]$ ls -l volatile.s
-rw-r–r– 1 narendra narendra 711 2016-11-19 16:22 volatile.s
[narendra@ubuntu]$
```

The above example may not be a good practical example, the purpose was to explain how compilers interpret volatile keyword. As a practical example, think of touch sensor on mobile phones. The driver abstracting touch sensor will read the location of touch and send it to higher level applications. The driver itself should not modify (const-ness) the read location, and make sure it reads the touch input every time fresh (volatile-ness). Such driver must read the touch sensor input in const volatile manner.

Note : The above codes are compiler specific and may not work on all compilers. The purpose of the examples is to make readers understand the concept.

Related Article :
Understanding “volatile” qualifier in C | Set 1 (Introduction)

Refer following links for more details on volatile keyword:
Volatile: A programmer’s best friend
Do not use volatile as a synchronization primitive

This article is compiled by “Narendra Kangralkar“ and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

Const Qualifier in C

The qualifier const can be applied to the declaration of any variable to specify that its value will not be changed (Which depends upon where const variables are stored, we may change value of const variable by using pointer). The result is implementation-defined if an attempt is made to change a const (See this forum topic).

1) Pointer to variable.

```
int *ptr;
```

We can change the value of ptr and we can also change the value of object ptr pointing to. Pointer and value pointed by pointer both are stored in read-write area. See the following code fragment.

```
#include <stdio.h>
int main(void)
{
    int i = 10;
    int j = 20;
    int *ptr = &i;        /* pointer to integer */
    printf("*ptr: %d\n", *ptr);

    /* pointer is pointing to another variable */
    ptr = &j;
    printf("*ptr: %d\n", *ptr);

    /* we can change value stored by pointer */
    *ptr = 100;
    printf("*ptr: %d\n", *ptr);

    return 0;
}
```

Output:

```
    *ptr: 10
    *ptr: 20
    *ptr: 100
```

2) Pointer to constant.

Pointer to constant can be declared in following two ways.

```
const int *ptr;
```

or

```
int const *ptr;
```

We can change pointer to point to any other integer variable, but cannot change value of object (entity) pointed using pointer ptr. Pointer is stored in read-write area (stack in present case). Object pointed may be in read only or read write area. Let us see following examples.

```
#include <stdio.h>
int main(void)
{
    int i = 10;
    int j = 20;
    const int *ptr = &i;    /* ptr is pointer to constant */

    printf("ptr: %d\n", *ptr);
    *ptr = 100;        /* error: object pointed cannot be modified
                 using the pointer ptr */

    ptr = &j;          /* valid */
    printf("ptr: %d\n", *ptr);

    return 0;
}
```

Output:

```
 error: assignment of read-only location '*ptr'
```

Following is another example where variable i itself is constant.

```
#include <stdio.h>

int main(void)
{
    int const i = 10;    /* i is stored in read only area*/
    int j = 20;

    int const *ptr = &i;        /* pointer to integer constant. Here i
                                 is of type "const int", and &i is of
                                 type "const int *".  And p is of type
                                "const int", types are matching no issue */

    printf("ptr: %d\n", *ptr);

    *ptr = 100;        /* error */

    ptr = &j;          /* valid. We call it as up qualification. In
                      C/C++, the type of "int *" is allowed to up
                      qualify to the type "const int *". The type of
                      &j is "int *" and is implicitly up qualified by
                      the compiler to "cons tint *" */

    printf("ptr: %d\n", *ptr);
    return 0;
}
```

Output:

```
 error: assignment of read-only location ‘*ptr’
```

Down qualification is not allowed in C++ and may cause warnings in C. Following is another example with down qualification.

```
#include <stdio.h>

int main(void)
{
    int i = 10;
    int const j = 20;

    /* ptr is pointing an integer object */
    int *ptr = &i;

    printf("*ptr: %d\n", *ptr);

    /* The below assignment is invalid in C++, results in error
       In C, the compiler *may* throw a warning, but casting is
       implicitly allowed */
    ptr = &j;

    /* In C++, it is called 'down qualification'. The type of expression
       &j is "const int *" and the type of ptr is "int *". The
       assignment "ptr = &j" causes to implicitly remove const-ness
       from the expression &j. C++ being more type restrictive, will not
       allow implicit down qualification. However, C++ allows implicit
       up qualification. The reason being, const qualified identifiers
       are bound to be placed in read-only memory (but not always). If
       C++ allows above kind of assignment (ptr = &j), we can use 'ptr'
       to modify value of j which is in read-only memory. The
       consequences are implementation dependent, the program may fail
       at runtime. So strict type checking helps clean code. */

    printf("*ptr: %d\n", *ptr);

    return 0;
}

// Reference http://www.dansaks.com/articles/1999-02%20const%20T%20vs%20T%20const.pdf

// More interesting stuff on C/C++ @ http://www.dansaks.com/articles.htm
```

3) Constant pointer to variable.

```
int *const ptr;
```

Above declaration is constant pointer to integer variable, means we can change value of object pointed by pointer, but cannot change the pointer to point another variable.

```
#include <stdio.h>

int main(void)
{
   int i = 10;
   int j = 20;
   int *const ptr = &i;    /* constant pointer to integer */

   printf("ptr: %d\n", *ptr);

   *ptr = 100;    /* valid */
   printf("ptr: %d\n", *ptr);

   ptr = &j;       /* error */
```

```
    return 0;
}
```

Output:

```
error: assignment of read-only variable ‘ptr’
```

4) constant pointer to constant

```
const int *const ptr;
```

Above declaration is constant pointer to constant variable which means we cannot change value pointed by pointer as well as we cannot point the pointer to other variable. Let us see with example.

```
#include <stdio.h>

int main(void)
{
    int i = 10;
    int j = 20;
    const int *const ptr = &i;        /* constant pointer to constant integer */

    printf("ptr: %d\n", *ptr);

    ptr = &j;           /* error */
    *ptr = 100;        /* error */

    return 0;
}
```

Output:

```
    error: assignment of read-only variable ‘ptr’
    error: assignment of read-only location ‘*ptr’
```

This article is compiled by “**Narendra Kangralkar**“. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
pointer

Initialization of static variables in C

In C, static variables can only be initialized using constant literals. For example, following program fails in compilation.

```
#include<stdio.h>
int initializer(void)
{
    return 50;
}

int main()
{
    static int i = initializer();
    printf(" value of i = %d", i);
    getchar();
    return 0;
}
```

If we change the program to following, then it works without any error.

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

```
{
   static int i = 50;
   printf(" value of i = %d", i);
   getchar();
   return 0;
}
```

The reason for this is simple: All objects with static storage duration must be initialized (set to their initial values) before execution of main() starts. So a value which is not known at translation time cannot be used for initialization of static variables.

Thanks to Venki and Prateek for their contribution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

Understanding “register” keyword in C

Registers are faster than memory to access, so the variables which are most frequently used in a C program can be put in registers using *register* keyword. The keyword *register* hints to compiler that a given variable can be put in a register. It’s compiler’s choice to put it in a register or not. Generally, compilers themselves do optimizations and put the variables in register.

1) If you use & operator with a register variable then compiler may give an error or warning (depending upon the compiler you are using), because when we say a variable is a register, it may be stored in a register instead of memory and accessing address of a register is invalid. Try below program.

```
int main()
{
  register int i = 10;
  int *a = &i;
  printf("%d", *a);
  getchar();
  return 0;
}
```

2) *register* keyword can be used with pointer variables. Obviously, a register can have address of a memory location. There would not be any problem with the below program.

```
int main()
{
  int i = 10;
  register int *a = &i;
  printf("%d", *a);
  getchar();
  return 0;
}
```

3) Register is a storage class, and C doesn’t allow multiple storage class specifiers for a variable. So, *register* can not be used with *static* . Try below program.

```
int main()
{
  int i = 10;
  register static int *a = &i;
  printf("%d", *a);
  getchar();
  return 0;
}
```

4) There is no limit on number of register variables in a C program, but the point is compiler may put some variables in register and some not.

Please write comments if you find anything incorrect in the above article or you want to share more information about register keyword.

Question 1
Which of the following is not a storage class specifier in C?
Aauto
Bregister
Cstatic
Dextern
Evolatile
Ftypedef
Storage Classes and Type Qualifiers
Discuss it
Question 1 Explanation:
volatile is not a storage class specifier. volatile and const are type qualifiers.
Question 2
Output of following program?

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

A4 3 2 1
B1 2 3 4
C0 0 0 0
DCompiler Error
Storage Classes and Type Qualifiers
Discuss it
Question 2 Explanation:
A static variable is shared among all calls of a function. All calls to main() in the given program share the same i. i becomes 0 before the printf() statement in all calls to main().
Question 3

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

A4 3 2 1
B1 2 3 4
C4 4 4 4
D0 0 0 0
Storage Classes and Type Qualifiers
Discuss it
Question 3 Explanation:
Since i is static variable, it is shared among all calls to main(). So is reduced by 1 by every function call.
Question 4

```
#include <stdio.h>
int main()
{
    int x = 5;
    int * const ptr = &x;
    ++(*ptr);
    printf("%d", x);

    return 0;
}
```

ACompiler Error
BRuntime Error
C6
D5
Storage Classes and Type Qualifiers
Discuss it
Question 4 Explanation:
See following declarations to know the difference between constant pointer and a pointer to a constant. **int * const ptr** —> ptr is constant pointer. You can change the value at the location pointed by pointer p, but you can not change p to point to other location. **int const * ptr** —> ptr is a pointer to a constant. You can change ptr to point other variable. But you cannot change the value pointed by ptr. Therefore above program works well because we have a constant pointer and we are not changing ptr to point to any other location. We are only icrementing value pointed by ptr.
Question 5

```
#include <stdio.h>
int main()
{
    int x = 5;
    int const * ptr = &x;
    ++(*ptr);
    printf("%d", x);

    return 0;
}
```

ACompiler Error
BRuntime Error
C6
D5
Storage Classes and Type Qualifiers
Discuss it
Question 5 Explanation:
See following declarations to know the difference between constant pointer and a pointer to a constant. **int * const ptr** —> ptr is constant pointer. You can change the value at the location pointed by pointer p, but you can not change p to point to other location. **int const * ptr** —> ptr is a pointer to a constant. You can change ptr to point other variable. But you cannot change the value pointed by ptr. In the above program, ptr is a pointer to a constant. So the value pointed cannot be changed.
Question 6

```
#include<stdio.h>
int main()
{
  typedef static int *i;
  int j;
  i a = &j;
  printf("%d", *a);
  return 0;
}
```

ARuntime Error
B0
CGarbage Value
DCompiler Error
Storage Classes and Type Qualifiers
Discuss it
Question 6 Explanation:
Compiler Error -> Multiple Storage classes for a. In C, typedef is considered as a storage class. The Error message may be different on different compilers.
Question 7
Output?

```
#include<stdio.h>
int main()
{
   typedef int i;
   i a = 0;
```

```
    printf("%d", a);
    return 0;
  }
```

ACompiler Error
BRuntime Error
C0
D1
Storage Classes and Type Qualifiers
Discuss it
Question 7 Explanation:
There is no problem with the program. It simply creates a user defined type i and creates a variable a of type i.
Question 8

```
#include<stdio.h>
int main()
{
  typedef int *i;
  int j = 10;
  i *a = &j;
  printf("%d", **a);
  return 0;
}
```

ACompiler Error
BGarbage Value
C10
D0
Storage Classes and Type Qualifiers
Discuss it
Question 8 Explanation:
Compiler Error -> Initialization with incompatible pointer type. The line typedef int *i makes i as type int *. So, the declaration of a means a is pointer to a pointer. The Error message may be different on different compilers.
Question 9
Output?

```
#include <stdio.h>
int fun()
{
  static int num = 16;
  return num--;
}

int main()
{
  for(fun(); fun(); fun())
    printf("%d ", fun());
  return 0;
}
```

AInfinite loop
B13 10 7 4 1
C14 11 8 5 2
D15 12 8 5 2
Storage Classes and Type Qualifiers
Discuss it
Question 9 Explanation:
Since *num* is static in *fun()*, the old value of *num* is preserved for subsequent functions calls. Also, since the statement return *num–* is postfix, it returns the old value of *num*, and updates the value for next function call.

```
fun() called first time: num = 16 // for loop initialization done;


In test condition, compiler checks for non zero value

fun() called again : num = 15

printf("%d \n", fun());:num=14 ->printed

Increment/decrement condition check
```

fun(); called again : num = 13

----

fun() called second time: num: 13

In test condition,compiler checks for non zero value

fun() called again : num = 12

printf("%d \n", fun());:**num=11 ->printed**

fun(); called again : num = 10

----

fun() called second time : num = 10

In test condition,compiler checks for non zero value

fun() called again : num = 9

printf("%d \n", fun());:**num=8 ->printed**

fun(); called again   : num = 7

----

fun() called second time: num = 7

In test condition,compiler checks for non zero value

fun() called again : num = 6

printf("%d \n", fun());:**num=5 ->printed**

fun(); called again   : num = 4

----

fun() called second time: num: 4

In test condition,compiler checks for non zero value

fun() called again : num = 3

printf("%d \n", fun());:**num=2 ->printed**

fun(); called again   : num = 1

----

fun() called second time: num: 1

In test condition,compiler checks for non zero value

fun() called again : num = 0 => **STOP**

Question 10

```
#include <stdio.h>
int main()
{
  int x = 10;
  static int y = x;

  if(x == y)
    printf("Equal");
  else if(x > y)
    printf("Greater");
  else
    printf("Less");
  return 0;
}
```

ACompiler Error
BEqual
CGreater
DLess
Storage Classes and Type Qualifiers
Discuss it
Question 10 Explanation:
In C, static variables can only be initialized using constant literals. This is allowed in C++ though. See this GFact for details.
Question 11
Consider the following C function

```
int f(int n)
{
   static int i = 1;
   if (n >= 5)
      return n;
   n = n+i;
   i++;
   return f(n);
}
```

The value returned by f(1) is (GATE CS 2004)
A5
B6
C7
D8
Storage Classes and Type Qualifiers
Discuss it
Question 11 Explanation:
Since i is static, first line of f() is executed only once.

```
Execution of f(1)
    i = 1
    n = 2
    i = 2
 Call f(2)
    i = 2
    n = 4
    i = 3
 Call f(4)
   i = 3
   n = 7
   i = 4
 Call f(7)
  since n >= 5 return n(7)
```

Question 12
In C, static storage class cannot be used with:
AGlobal variable
BFunction parameter
CFunction name
DLocal variable
Storage Classes and Type Qualifiers
Discuss it
Question 12 Explanation:
Declaring a global variable as static limits its scope to the same file in which it is defined. A static function is only accessible to the same file in which it is defined. A local variable declared as static preserves the value of the variable between the function calls.
Question 13
Output? (GATE CS 2012)

```
#include <stdio.h>
int a, b, c = 0;
void prtFun (void);
int main ()
{
    static int a = 1; /* line 1 */
    prtFun();
    a += 1;
    prtFun();
    printf ( "\n %d %d " , a, b) ;
}

void prtFun (void)
{
    static int a = 2; /* line 2 */
    int b = 1;
    a += ++b;
    printf (" \n %d %d " , a, b);
}
```

A3 1
  4 1
  4 2
B4 2
  6 1
  6 1
C4 2
  6 2
  2 0
D3 1
  5 2
  5 2

Storage Classes and Type Qualifiers
Discuss it

Question 13 Explanation:
'a' and 'b' are global variable. prtFun() also has 'a' and 'b' as local variables. The local variables hide the globals (See Scope rules in C). When prtFun() is called first time, the local 'b' becomes 2 and local 'a' becomes 4. When prtFun() is called second time, same instance of local static 'a' is used and a new instance of 'b' is created because 'a' is static and 'b' is non-static. So 'b' becomes 2 again and 'a' becomes 6. main() also has its own local static variable named 'a' that hides the global 'a' in main. The printf() statement in main() accesses the local 'a' and prints its value. The same printf() statement accesses the global 'b' as there is no local variable named 'b' in main. Also, the defaut value of static and global int variables is 0. That is why the printf statement in main() prints 0 as value of b.
Question 14
What output will be generated by the given code d\segment if: Line 1 is replaced by "auto int a = 1;" Line 2 is replaced by "register int a = 2;" (GATE CS 2012)
A3 1
  4 1
  4 2
B4 2
  6 1
  6 1
C4 2
  6 2
  2 0
D4 2
  4 2
  2 0

Storage Classes and Type Qualifiers
Discuss it

Question 14 Explanation:
If we replace line 1 by "auto int a = 1;" and line 2 by "register int a = 2;", then 'a' becomes non-static in prtFun(). The output of first prtFun() remains same. But, the output of second prtFun() call is changed as a new instance of 'a' is created in second call. So "4 2″ is printed again. Finally, the printf() in main will print "2 0″. Making 'a' a register variable won't change anything in output. Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above.
Question 15
Output?

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

APrints 10 on all compilers
BMay generate compiler Error
CPrints 0 on all compilers

DMay generate runtime Error
Storage Classes and Type Qualifiers
Discuss it
Question 15 Explanation:
See point 1 of Register Keyword
Question 16

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

A0 10
BCompiler Error
C0 0
D10 10
Storage Classes and Type Qualifiers
Discuss it
Question 16 Explanation:
See extern keyword
Question 17
Output?

```
#include <stdio.h>

int main(void)
{
    int i = 10;
    const int *ptr = &i;
    *ptr = 100;
    printf("i = %d\n", i);
    return 0;
}
```

Ai = 100
Bi = 10
CCompiler Error
DRuntime Error
Storage Classes and Type Qualifiers
Discuss it
Question 17 Explanation:
Note that ptr is a pointer to a constant. So value pointed cannot be changed using the pointer ptr. See Const Qualifier in C for more details.
Question 18
Output of following program

```
#include <stdio.h>
int fun(int n)
{
    static int s = 0;
    s = s + n;
    return (s);
}

int main()
{
    int i = 10, x;
    while (i > 0)
    {
        x = fun(i);
        i--;
    }
    printf ("%d ", x);
    return 0;
}
```

A0
B100
C110
D55
Storage Classes and Type Qualifiers
Discuss it
Question 18 Explanation:
Since s is static, different values of i are added to it one by one. So final value of s is s = i + (i-1) + (i-2) + ... 3 + 2 + 1. The value of s is i*(i+1)/2. For i = 10, s is 55.
Question 19

```
#include <stdio.h>
char *fun()
{
   static char arr[1024];
   return arr;
}

int main()
{
   char *str = "geeksforgeeks";
   strcpy(fun(), str);
   str = fun();
   strcpy(str, "geeksquiz");
   printf("%s", fun());
   return 0;
}
```

Ageeksforgeeks
Bgeeksquiz
Cgeeksforgeeks geeksquiz
DCompiler Error
Storage Classes and Type Qualifiers
Discuss it
Question 19 Explanation:
Note that arr[] is static in fun() so no problems of returning address, arr[] will stay there even after the fun() returns and all calls to fun() share the same arr[].

```
   strcpy(fun(), str);  // Copies "geeksforgeeks" to arr[]
   str = fun();    // Assigns address of arr to str
   strcpy(str, "geeksquiz");  // copies geeksquiz to str which is address of arr[]
   printf("%s", fun());   // prints "geeksquiz"
```

Question 20
Consider the following C function, what is the output?

```
#include <stdio.h>
int f(int n)
{
   static int r = 0;
   if (n <= 0) return 1;
   if (n > 3)
   {
      r = n;
      return f(n-2)+2;
   }
   return f(n-1)+r;
}

int main()
{
   printf("%d", f(5));
}
```

A5
B7
C9
D18
Storage Classes and Type Qualifiers   GATE-CS-2007
Discuss it
Question 20 Explanation:

```
f(5) = f(3)+2
The line "r = n" changes value of r to 5.  Since r
is static, its value is shared be all subsequence
calls.  Also, all subsequent calls don't change r
because the statement "r = n" is in a if condition
with n > 3.

f(3) = f(2)+5
f(2) = f(1)+5
f(1) = f(0)+5
f(0) = 1

So f(5) = 1+5+5+5+2 = 18
```

Question 21
In the context of C data types, which of the followings is correct?
A“unsigned long long int” is a valid data type.
B“long long double” is a valid data type.
C“unsigned long double” is a valid data type.
DA), B) and C) all are valid data types.
EA), B) and C) all are invalid data types.
Storage Classes and Type Qualifiers   C Quiz - 102
Discuss it
Question 21 Explanation:
In C, "float" is single precision floating type. "double" is double precision floating type. "long double"is often more precise than double precision floating type. So the maximum floating type is "long double". There's nothing called "long long double". If someone wants to use bigger range than "long double", we need to define our own data type i.e. user defined data type. Besides, Type Specifiers "signed" and "unsigned" aren't applicable for floating types (float, double, long double). Basically, floating types are always signed only.

But integer types i.e. "int", "long int" and "long long int" are valid combinations. As per C standard, "long long int" would be at least 64 bits i.e. 8 bytes. By default integer types would be signed. If we need to make these integer types as unsigned, one can use Type Specifier "unsigned". That's why A) is correct answer.

Question 22
For the following “typedef” in C, pick the best statement

```
typedef int INT, *INTPTR, ONEDARR[10], TWODARR[10][10];
```

AIt will cause compile error because typedef is being used to define multiple aliases of incompatible types in the same statement.
B“INT x” would define x of type int. Remaining part of the statement would be ignored.
C“INT x” would define x of type int and “INTPTR y” would define pointer y of type int *. Remaining part of the statement would be ignored.
D“INT x” would define x of type int. “INTPTR y” would define pointer y of type int *. ONEDARR is an array of 10 int. TWODARR is a 2D array of 10 by 10 int.
E“INT x” would define x of type int. “INTPTR *y” would define pointer y of type int **. “ONEDARR z” would define z as array of 10 int. “TWODARR t” would define t as array of 10 by 10 int.
Storage Classes and Type Qualifiers   C Quiz - 106
Discuss it
Question 22 Explanation:
Here, INT is alias of int. INTPTR is alias of int *. That’s why INTPTR * would be alias of int **. Similarly, ONEDARR is defining the alias not array itself. ONEDARR would be alias to int [10]. That’s why “ONEDARR z” would define array z of int [10]. Similarly, TWODARR would be alias to int [10][10]. Hence “TWODARR t” would define array t of int [10][10]. We can see that typedef can be used to create alias or synonym of other types.
Question 23
Which of the followings is correct for a function definition along with storage-class specifier in C language?
Aint fun(auto int arg)
Bint fun(static int arg)
Cint fun(register int arg)
Dint fun(extern int arg)
EAll of the above are correct.
Storage Classes and Type Qualifiers   C Quiz - 107
Discuss it
Question 23 Explanation:
As per C standard, “*The only storage-class specifier that shall occur in a parameter declaration is register.*” That’s why correct answer is C.
Question 24
Pick the correct statement for const and volatile.
Aconst is the opposite of volatile and vice versa.
Bconst and volatile can’t be used for struct and union.
Cconst and volatile can’t be used for enum.
Dconst and volatile can’t be used for typedef.

Econst and volatile are independent i.e. it’s possible that a variable is defined as both const and volatile.
Storage Classes and Type Qualifiers   **C Quiz - 107**
Discuss it
Question 24 Explanation:
In C, const and volatile are type qualifiers and these two are independent. Basically, const means that the value isn’t modifiable by the program. And volatile means that the value is subject to sudden change (possibly from outside the program). In fact, C standard mentions an example of valid declaration which is both const and volatile. The example is “extern const volatile int real_time_clock;” where real_time_clock may be modifiable by hardware, but cannot be assigned to, incremented, or decremented. So we should already treat const and volatile separately. Besides, these type qualifier applies for struct, union, enum and typedef as well.
Question 25
Pick the best statement for the following program snippet:

```
#include "stdio.h"
void foo(void)
{
 static int staticVar;
 staticVar++;
 printf("foo: %d\n",staticVar);
}

void bar(void)
{
 static int staticVar;
 staticVar++;
 printf("bar: %d\n",staticVar);
}

int main()
{
 foo(), bar(), foo();
 return 0;
}
```

ACompile error because same static variable name is used in both foo and bar. Since these static variables retain their values even after function is over, same name can’t be used in both the functions.
BCompile error because semicolon isn’t used while calling foo() and bar() in side main function.
CNo compile error and only one copy of staticVar would be used across both the functions and that’s why final value of that single staticVar would be 3.
DNo compile error and separate copies of staticVar would be used in both the functions. That’s why staticVar in foo() would be 2 while staticVar in bar() would be 1.
Storage Classes and Type Qualifiers   **C Quiz - 111**
Discuss it
Question 25 Explanation:
Here, even though life of static variables span across function calls but their scope is respective to their function body only. That’s why staticVar of each function has separate copies whose life span across function calls. And d is correct.
There are 25 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Returned values of printf() and scanf()

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();
}
```

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

```
int main()
```

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

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
GFacts

What is return type of getchar(), fgetc() and getc() ?

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);
}
```

See this for more details.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
GFacts

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;
}
```

```
[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);
```

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;
}
```

```
[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;
}
```

```
[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 post.

This article is compiled by “Narendra Kangralkar“ and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

puts() vs printf() for printing a string

In C, given a string variable *str*, which of the following two should be preferred to print it to stdout?

```
1)  puts(str);
```

```
2)  printf(str);
```

puts() can be preferred for printing a string because it is generally less expensive (implementation of puts() is generally simpler than printf()), and if the string has formatting characters like '%', then printf() would give unexpected results. Also, if str is a user input string, then use of printf() might cause security issues (see this for details).
Also note that puts() moves the cursor to next line. If you do not want the cursor to be moved to next line, then you can use following variation of puts().

```
fputs(str, stdout)
```

You can try following programs for testing the above discussed differences between puts() and printf().

Program 1

```
#include<stdio.h>
int main()
{
   puts("Geeksfor");
   puts("Geeks");

   getchar();
   return 0;
}
```

Program 2

```
#include<stdio.h>
int main()
{
   fputs("Geeksfor", stdout);
   fputs("Geeks", stdout);

   getchar();
   return 0;
}
```

Program 3

```
#include<stdio.h>
int main()
{
   // % is intentionally put here to show side effects of using printf(str)
   printf("Geek%sforGeek%s");
   getchar();
   return 0;
}
```

Program 4

```
#include<stdio.h>
int main()
{
   puts("Geek%sforGeek%s");
   getchar();
   return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

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;
}
```

The above program prints “geeks for geeks 10”. The first printf() prints “geeks for geeks”. The second printf() prints 10 as there are 10 characters printed (the 10 characters are “geeks for “) before %n in first printf() and c is set to 10 by first printf().

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

How to print % using printf()?

Asked by Tanuj

Here is the standard prototype of printf function in C.

```
        int printf(const char *format, ...);
```

The format string is composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of argument (and it is an error if insufficiently many arguments are given).

The character % is followed by one of the following characters.

The flag character
The field width
The precision
The length modifier
The conversion specifier:

See http://swoolley.org/man.cgi/3/printf for details of all the above characters. The main thing to note in the standard is the below line about conversion specifier.

```
A `%' is written. No argument is converted. The complete conversion specification is`%%'.
```

So we can print “%” using “%%”

```
/* Program to print %*/
#include<stdio.h>
/* Program to print %*/
int main()
{
   printf("%%");
   getchar();
   return 0;
}
```

We can also print “%” using below.

```
   printf("%c", '%');
   printf("%s", "%");
```

Question 1
Predict the output of following program?

```
#include "stdio.h"
int main()
{
   char arr[100];
   printf("%d", scanf("%s", arr));
   /* Suppose that input value given
      for above scanf is "GeeksQuiz" */
   return 1;
}
```

A9
B1
C10
D100
Input and Output
Discuss it
Question 1 Explanation:
In C, scanf returns the no. of inputs it has successfully read. See http://geeksforgeeks.org/archives/674
Question 2
Predict output of the following program

```
#include <stdio.h>
int main()
{
  printf("\new_c_question\by");
  printf("\rgeeksforgeeks");
  getchar();
  return 0;
}
```

Aew_c_question
  geeksforgeeks
Bnew_c_ques
  geeksforgeeks
C
  geeksforgeeks
DDepends on terminal configuration
Input and Output
Discuss it
Question 2 Explanation:
See http://stackoverflow.com/questions/17236242/usage-of-b-and-r-in-c
Question 3

```
#include <stdio.h>

int main()
{
 printf(" \"GEEKS %% FOR %% GEEKS\"");
 getchar();
 return 0;
}
```

A“GEEKS % FOR % GEEKS”
BGEEKS % FOR % GEEKS
C\"GEEKS %% FOR %% GEEKS\"
DGEEKS %% FOR %% GEEKS
Input and Output

Discuss it
Question 3 Explanation:
Backslash (\\\\) works as escape character for double quote (“). For explanation of %%, see http://www.geeksforgeeks.org/how-to-print-using-printf/
Question 4

```
#include <stdio.h>
// Assume base address of "GeeksQuiz" to be 1000
int main()
{
  printf(5 + "GeeksQuiz");
  return 0;
}
```

AGeeksQuiz
BQuiz
C1005
DCompile-time error
Input and Output
Discuss it
Question 4 Explanation:
printf is a library function defined under *stdio.h* header file. The compiler adds 5 to the base address of the string through the expression ***5 + "GeeksQuiz"*** . Then the string "Quiz" gets passed to the standard library function as an argument.

Question 5
Predict the output of the below program:

```
#include <stdio.h>

int main()
{
   printf("%c ", 5["GeeksQuiz"]);
   return 0;
}
```

ACompile-time error
BRuntime error
CQ
Ds
Input and Output
Discuss it
Question 5 Explanation:
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.
Question 6
Predict the output of below program:

```
#include <stdio.h>
int main()
{
   printf("%c ", "GeeksQuiz"[5]);
   return 0;
}
```

ACompile-time error
BRuntime error
CQ
Ds
Input and Output
Discuss it
Question 6 Explanation:
The crux of the program lies in the expression: **"GeeksQuiz"[5]**. This expression is broken down by the compiler as: *(“GeeksQuiz” + 5). 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.
Question 7
Which of the following is true
Agets() can read a string with newline chacters but a normal scanf() with %s can not.
Bgets() can read a string with spaces but a normal scanf() with %s can not.
Cgets() can always replace scanf() without any additional code.
DNone of the above

Input and Output
Discuss it
Question 7 Explanation:
gets() can read a string with spaces but a normal scanf() with %s can not. Consider following program as an example. If we enter "Geeks Quiz" as input in below program, the program prints "Geeks" 1 But in the following program, if we enter "Geeks Quiz", it prints "Geeks Quiz" 1
Question 8
Which of the following is true
Agets() doesn't do any array bound testing and should not be used.
Bfgets() should be used in place of gets() only for files, otherwise gets() is fine
Cgets() cannot read strings with spaces
DNone of the above
Input and Output
Discuss it
Question 8 Explanation:
See gets() is risky to use!
Question 9
What does the following C statement mean?

```
scanf("%4s", str);
```

ARead exactly 4 characters from console.
BRead maximum 4 characters from console.
CRead a string str in multiples of 4
DNothing
Input and Output
Discuss it
Question 9 Explanation:
Try following program, enter GeeksQuiz, the output would be "Geek"

```
#include <stdio.h>

int main()
{
   char str[50] = {0};
   scanf("%4s", str);
   printf(str);
   getchar();
   return 0;
}
```

Question 10

```
#include<stdio.h>

int main()
{
   char *s = "Geeks Quiz";
   int n = 7;
   printf("%.*s", n, s);
   return 0;
}
```

AGeeks Quiz
BNothing is printed
CGeeks Q
DGeeks Qu
Input and Output
Discuss it
Question 10 Explanation:
.* 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.
Question 11
Predict the output of following program?

```
#include <stdio.h>
int main(void)
{
  int x = printf("GeeksQuiz");
  printf("%d", x);
```

```
    return 0;
}
```

AGeeksQuiz9
BGeeksQuiz10
CGeeksQuizGeeksQuiz
DGeeksQuiz1
Input and Output
Discuss it
Question 11 Explanation:
The printf function returns the number of characters successfully printed on the screen. The string "GeeksQuiz" has 9 characters, so the first printf prints GeeksQuiz and returns 9.
Question 12
Output of following program?

```
#include<stdio.h>
int main()
{
    printf("%d", printf("%d", 1234));
    return 0;
}
```

A12344
B12341
C11234
D41234
Input and Output
Discuss it
Question 12 Explanation:
printf() returns the number of characters successfully printed on the screen.
Question 13
What is the return type of getchar()?
Aint
Bchar
Cunsigned char
Dfloat
Input and Output
Discuss it
Question 13 Explanation:
The return type of getchar() is int to accommodate EOF which indicates failure:
Question 14
Normally user programs are prevented from handling I/O directly by I/O instructions in them. For CPUs having explicit I/O instructions, such I/O protection is ensured by having the I/O instructions privileged. In a CPU with memory mapped I/O, there is no explicit I/O instruction. Which one of the following is true for a CPU with memory mapped I/O?
AI/O protection is ensured by operating system routine (s)
BI/O protection is ensured by a hardware trap
CI/O protection is ensured during system configuration
DI/O protection is not possible
Input and Output **GATE-CS-2005**
Discuss it
Question 14 Explanation:
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.
Question 15
Which of the following functions from “stdio.h” can be used in place of **printf()**?
Afputs() with FILE stream as stdout.
Bfprintf() with FILE stream as stdout.
Cfwrite() with FILE stream as stdout.
DAll of the above three - a, b and c.
EIn “stdio.h”, there’s no other equivalent function of printf().
Input and Output **C Quiz - 103**
Discuss it
Question 15 Explanation:
Though fputs() and fwrite() can accept FILE stream stdout and can output the given string but the input string wouldn’t result in formatted (i.e. containing format specifiers) output. But fprintf() can be used for formatted output. That’s why *fprintf(stdout,"=%d=",a);* and *printf("=%d=",a);* both are equivalent. The correct answer is B.
There are 15 questions to complete.

What is the difference between printf, sprintf and fprintf?

printf:

printf function is used to print character stream of data on stdout console.

Syntax:

```
int printf(const char* str, ...);
```

Example:

```
// simple print on stdout
#include<stdio.h>
int main()
{
   printf("hello geeksquiz");
   return 0;
}
```

Output:

```
hello geeksquiz
```

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:

```
// Example program to demonstrate sprintf()
#include<stdio.h>
int main()
{
    char buffer[50];
    int a = 10, b = 20, c;
    c = a + b;
    sprintf(buffer, "Sum of %d and %d is %d", a, b, c);

    // The string "sum of 10 and 20 is 30" is stored
    // into buffer instead of printing on stdout
    printf("%s", buffer);

    return 0;
}
```

Output:

```
Sum of 10 and 20 is 30
```

fprintf:

fprintf is used to print the sting content in file but not on stdout console.

```
int fprintf(FILE *fptr, const char *str, ...);
```

Example:

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

```
{
   int i, n=2;
   char str[50];

   //open file sample.txt in write mode
   FILE *fptr = fopen("sample.txt", "w");
   if (fptr == NULL)
   {
      printf("Could not open file");
      return 0;
   }

   for (i=0; i<n; i++)
   {
      puts("Enter a name");
      gets(str);
      fprintf(fptr,"%d.%s\n", i, str);
   }
   fclose(fptr);

   return 0;
}
```

```
Input: GeeksforGeeks
      GeeksQuiz
Output:  sample.txt file now having output as
0. GeeksforGeeks
1. GeeksQuiz
```

Thank you for reading, i will soon update with scanf, fscanf, sscanf keep tuned.

This article is contributed by **Vankayala Karunakar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C

Difference between getc(), getchar(), getch() and getche()

All of these functions read a character from input and return an integer value. The integer is returned to accommodate a special value used to indicate failure. The value EOF is generally used for this purpose.

getc():

It reads a single character from a given input stream and returns the corresponding integer value (typically ASCII value of read character) on success. It returns EOF on failure.

Syntax:

```
int getc(FILE *stream);
```

Example:

```
// Example for getc() in C
#include <stdio.h>
int main()
{
  printf("%c", getc(stdin));
  return(0);
}
```

```
Input: g (press enter key)
Output: g
```

getchar():

The difference between getc() and getchar() is getc() can read from any input stream, but getchar() reads from standard input. So getchar() is equivalent to getc(stdin).

Syntax:

```
int getchar(void);
```

Example:

```
// Example for getchar() in C
#include <stdio.h>
int main()
{
  printf("%c", getchar());
  return 0;
}
```

```
Input: g(press enter key)
Output: g
```

getch():

getch() is a nonstandard function and is present in conio.h header file which is mostly used by MS-DOS compilers like Turbo C. It is not part of the C standard library or ISO C, nor is it defined by POSIX (Source: http://en.wikipedia.org/wiki/Conio.h)
Like above functions, it reads also a single character from keyboard. But it does not use any buffer, so the entered character is immediately returned without waiting for the enter key.
Syntax:

```
int getch();
```

Example:

```
// Example for getch() in C
#include <stdio.h>
#include <conio.h>
int main()
{
  printf("%c", getch());
  return 0;
}
```

```
Input:  g (Without enter key)
Output: Program terminates immediately.
     But when you use DOS shell in Turbo C,
     it shows a single g, i.e., 'g'
```

getche()

Like getch(), this is also a non-standard function present in conio.h. It reads a single character from the keyboard and displays immediately on output screen without waiting for enter key.

Syntax:

```
int getche(void);
```

Example:

```
#include <stdio.h>
#include <conio.h>
// Example for getche() in C
int main()
{
 printf("%c", getche());
```

```
  return 0;
}
```

```
Input: g(without enter key as it is not buffered)
Output: Program terminates immediately.
    But when you use DOS shell in Turbo C,
    double g, i.e., 'gg'
```

This article is contributed by **Vankayala Karunakar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C

Operators in C | Set 1 (Arithmetic Operators)

Operators are the foundation of any programming language. Thus the functionality of C language is incomplete without the use of operators. Operators allow us to perform different kinds of operations on operands. In C, operators in Can be categorized in following categories:

- **Arithmetic Operator**s (+, -, *, /, %, post-increment, pre-increment, post-decrement, pre-decrement)
- **Relational Operators** (==, != , >, <, >= & <=) Logical Operators (&&, || and !)
- **Bitwise Operators** (&, |, ^, ~, >> and <<)
- **Assignment Operator**s (=, +=, -=, *=, etc)
- **Other Operators** (conditional, comma, sizeof, address, redirecton)

Arithmetic Operators: These are used to perform arithmetic/mathematical operations on operands. The binary operators falling in this category are:

- **Addition:** The **'+'** operator adds two operands. For example, **x+y**.
- **Subtraction:** The **'-'** operator subtracts two operands. For example, **x-y**.
- **Multiplication:** The **'*'** operator multiplies two operands. For example, **x*y**.
- **Division:** The '/' operator divides the first operand by the second. For example, **x/y**.
- **Modulus:** The **'%'** operator returns the remainder when first operand is divided by the second. For example, **x%y**.

```
// C program to demonstrate working of binary arithmetic operators
#include<stdio.h>

int main()
{
   int a = 10, b = 4, res;

   //printing a and b
   printf("a is %d and b is %d\n", a, b);

   res = a+b; //addition
   printf("a+b is %d\n", res);

   res = a-b; //subtraction
   printf("a-b is %d\n", res);

   res = a*b; //multiplication
   printf("a*b is %d\n", res);

   res = a/b; //division
   printf("a/b is %d\n", res);

   res = a%b; //modulus
   printf("a%%b is %d\n", res);
```

```
    return 0;
}
```

Output:

```
a is 10 and b is 4
a+b is 14
a-b is 6
a*b is 40
a/b is 2
a%b is 2
```

The ones falling into the category of unary arithmetic operators are:

- **Increment:** The '**++**' operator is used to increment the value of an integer. When placed before the variable name (also called pre-increment operator), its value is incremented instantly. For example, **++x**.
  And when it is placed after the variable name (also called post-increment operator), its value is preserved temporarily until the execution of this statement and it gets updated before the execution of the next statement. For example, **x++**.
- **Decrement:** The '**–**' operator is used to decrement the value of an integer. When placed before the variable name (also called pre-decrement operator), its value is decremented instantly. For example, **–x**.
  And when it is placed after the variable name (also called post-decrement operator), its value is preserved temporarily until the execution of this statement and it gets updated before the execution of the next statement. For example, **x–**.

```
// C program to demonstrate working of Unary arithmetic operators
#include<stdio.h>

int main()
{
    int a = 10, b = 4, res;

    // post-increment example:
    // res is assigned 10 only, a is not updated yet
    res = a++;
    printf("a is %d and res is %d\n", a, res); //a becomes 11 now


    // post-decrement example:
    // res is assigned 11 only, a is not updated yet
    res = a--;
    printf("a is %d and res is %d\n", a, res);  //a becomes 10 now


    // pre-increment example:
    // res is assigned 11 now since a is updated here itself
    res = ++a;
    // a and res have same values = 11
    printf("a is %d and res is %d\n", a, res);


    // pre-decrement example:
    // res is assigned 10 only since a is updated here itself
    res = --a;
    // a and res have same values = 10
    printf("a is %d and res is %d\n",a,res);

    return 0;
}
```

Output:

```
a is 11 and res is 10
a is 10 and res is 11
a is 11 and res is 11
a is 10 and res is 10
```

We will soon be discussing other categories of operators in different posts.

To know about **Operator Precedence and Associativity**, refer this link:

Quiz on Operators in C

This article is contributed by Ayush Jaggi. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C

Operators in C | Set 2 (Relational and Logical Operators)

We have discussed introduction to operators in C and Arithmetic Operators. In this article, Relational and Logical Operators are discussed.

Relational Operators:
Relational operators are used for comparison of two values. Let's see them one by one:

- **'=='** operator checks whether the two given operands are equal or not. If so, it returns true. Otherwise it returns false. For example, **5==5** will return true.
- **'!='** operator checks whether the two given operands are equal or not. If not, it returns true. Otherwise it returns false. It is the exact boolean complement of the **'=='** operator. For example, **5!=5** will return false.
- **'>'** operator checks whether the first operand is greater than the second operand. If so, it returns true. Otherwise it returns false. For example, **6>5** will return true.
- **'<'** operator checks whether the first operand is lesser than the second operand. If so, it returns true. Otherwise it returns false. For example, **6<5** will return false.
- **'>='** operator checks whether the first operand is greater than or equal to the second operand. If so, it returns true. Otherwise it returns false. For example, **5>=5** will return true.
- **'<='** operator checks whether the first operand is lesser than or equal to the second operand. If so, it returns true. Otherwise it returns false. For example, **5<=5** will also return true.

```
// C program to demonstrate working of relational operators
#include <stdio.h>

int main()
{
   int a=10, b=4;

   // relational operators
   // greater than example
   if (a > b)
      printf("a is greater than b\n");
   else printf("a is less than or equal to b\n");

   // greater than equal to
   if (a >= b)
      printf("a is greater than or equal to b\n");
   else printf("a is lesser than b\n");

   // less than example
   if (a < b)
      printf("a is less than b\n");
   else printf("a is greater than or equal to b\n");

   // lesser than equal to
   if (a <= b)
      printf("a is lesser than or equal to b\n");
   else printf("a is greater than b\n");

   // equal to
   if (a == b)
```

```
        printf("a is equal to b\n");
    else printf("a and b are not equal\n");

    // not equal to
    if (a != b)
        printf("a is not equal to b\n");
    else printf("a is equal b\n");

    return 0;
}
```

Output:

```
a is greater than b
a is greater than or equal to b
a is greater than or equal to b
a is greater than b
a and b are not equal
a is not equal to b
```

Logical Operators:

They are used to combine two or more conditions/constraints or to complement the evaluation of the original condition in consideration. They are described below:

- **Logical AND:** The **'&&'** operator returns true when both the conditions in consideration are satisfied. Otherwise it returns false. For example, **a && b** returns true when both a and b are true (i.e. non-zero).
- **Logical OR:** The '||' operator returns true when one (or both) of the conditions in consideration is satisfied. Otherwise it returns false. For example, **a** || **b** returns true if one of a or b is true (i.e. non-zero). Of course, it returns true when both a and b are true.
- **Logical NOT:** The **'!'** operator returns true the condition in consideration is not satisfied. Otherwise it returns false. For example, **!a** returns true if a is false, i.e. when a=0.

```
// C program to demonstrate working of logical operators
#include <stdio.h>

int main()
{
    int a=10, b=4, c = 10, d = 20;

    // logical operators

    // logical AND example
    if (a>b && c==d)
        printf("a is greater than b AND c is equal to d\n");
    else printf("AND condition not satisfied\n");

    // logical AND example
    if (a>b || c==d)
        printf("a is greater than b OR c is equal to d\n");
    else printf("Neither a is greater than b nor c is equal "
            " to d\n");

    // logical NOT example
    if (!a)
        printf("a is zero\n");
    else printf("a is not zero");

    return 0;
}
```

Output:

```
AND condition not satisfied
a is greater than b OR c is equal to d
a is not zero
```

Short-Circuiting in Logical Operators:

In case of **logical AND**, the second operand is not evaluated if first operand is false. For example, program 1 below doesn't print

“GeeksQuiz” as the first operand of logical AND itself is false.

```
#include <stdio.h>
#include <stdbool.h>
int main()
{
   int a=10, b=4;
   bool res = ((a == b) && printf("GeeksQuiz"));
   return 0;
}
```

But below program prints “GeeksQuiz” as first operand of logical AND is true.

```
#include <stdio.h>
#include <stdbool.h>
int main()
{
   int a=10, b=4;
   bool res = ((a != b) && printf("GeeksQuiz"));
   return 0;
}
```

In case of **logical OR**, the second operand is not evaluated if first operand is true. For example, program 1 below doesn’t print “GeeksQuiz” as the first operand of logical OR itself is true.

```
#include <stdio.h>
#include <stdbool.h>
int main()
{
   int a=10, b=4;
   bool res = ((a != b) || printf("GeeksQuiz"));
   return 0;
}
```

But below program prints “GeeksQuiz” as first operand of logical OR is false.

```
#include <stdio.h>
#include <stdbool.h>
int main()
{
   int a=10, b=4;
   bool res = ((a == b) || printf("GeeksQuiz"));
   return 0;
}
```

Quiz on Operators in C

This article is contributed by Ayush Jaggi. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C

Interesting Facts about Bitwise Operators in C

In C, following 6 operators are bitwise operators (work at bit-level)

& (bitwise AND) Takes two numbers as operand and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.

| **(bitwise OR)** Takes two numbers as operand and does OR on every bit of two numbers. The result of OR is 1 any of the two bits is 1.

^ (bitwise XOR) Takes two numbers as operand and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.

Takes two numbers, left shifts the bits of first operand, the second operand decides the number of places to shift.

>> (right shift) Takes two numbers, right shifts the bits of first operand, the second operand decides the number of places to shift.

~ **(bitwise NOT)** Takes one number and inverts all bits of it

Following is example C program.

```
/* C Program to demonstrate use of bitwise operators */
#include<stdio.h>
int main()
{
    unsigned char a = 5, b = 9; // a = 4(00000101), b = 8(00001001)
    printf("a = %d, b = %d\n", a, b);
    printf("a&b = %d\n", a&b); // The result is 00000001
    printf("a|b = %d\n", a|b);  // The result is 00001101
    printf("a^b = %d\n", a^b); // The result is 00001100
    printf("~a = %d\n", a = ~a);   // The result is 11111010
    printf("b<<1 = %d\n", b<<1);  // The result is 00010010
    printf("b>>1 = %d\n", b>>1);  // The result is 00000100
    return 0;
}
```

Output:

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

Following are interesting facts about bitwise operators.

1) The left shift and right shift operators should not be used for negative numbers The result of > is undefined behabiour if any of the operands is a negative number. For example results of both -1 this for more details.

2) The bitwise XOR operator is the most useful operator from technical interview perspective. It is used in many problems. A simple example could be “Given a set of numbers where all elements occur even number of times except one number, find the odd occuring number” This problem can be efficiently solved by just doing XOR of all numbers.

```
// 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
```

The following are many other interesting problems which can be used using XOR operator.
Find the Missing Number, swap two numbers without using a temporary variable, A Memory Efficient Doubly Linked List, and Find the two non-repeating elements. There are many more (See this, this, this, this, this and this)

3) The bitwise operators should not be used in-place of logical operators.
The result of logical operators (&&, || and !) is either 0 or 1, but bitwise operators return an integer value. Also, the logical operators consider any non-zero operand as 1. For example consider the following program, the results of & and && are different for same operands.

```
int main()
{
   int x = 2, y = 5;
   (x & y)? printf("True ") : printf("False ");
   (x && y)? printf("True ") : printf("False ");
   return 0;
}
// Output: False True
```

4) The left-shift and right-shift operators are equivalent to multiplication and division by 2 respectively.

As mentioned in point 1, it works only if numbers are positive.

```
int main()
{
   int x = 19;
   printf ("x << 1 = %d\n", x << 1);
   printf ("x >> 1 = %d\n", x >> 1);
   return 0;
}
// Output: 38 9
```

5) The & operator can be used to quickly check if a number is odd or even

The value of expression (x & 1) would be non-zero only if x is odd, otherwise the value would be zero.

```
int main()
{
   int x = 19;
   (x & 1)? printf("Odd"): printf("Even");
   return 0;
}
// Output: Odd
```

6) The ~ operator should be used carefully

The result of ~ operator on a small number can be a big number if result is stored in a unsigned variable. And result may be negative number if result is stored in signed variable (assuming that the negative numbers are stored in 2's complement form where leftmost bit is the sign bit)

```
// Note that the output of following program is compiler dependent
int main()
{
   unsigned int x = 1;
   printf("Signed Result %d \n", ~x);
   printf("Unsigned Result %ud \n", ~x);
   return 0;
}
/* Output:
Signed Result -2
Unsigned Result 4294967294d */
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

Bit Magic
C/C++ Puzzles
cpp-operator
XOR

Interesting facts about Operator Precedence and Associativity in C

Operator precedence determines which operator is performed first in an expression with more than one operators with different precedence. For example 10 + 20 * 30 is calculated as 10 + (20 * 30) and not as (10 + 20) * 30.

Associativity is used when two operators of same precedence appear in an expression. Associativity can be either **L**eft **t**o **R**ight or **R**ight **t**o **L**eft. For example '*' and '/' have same precedence and their associativity is **L**eft **t**o **R**ight, so the expression "100 / 10 * 10" is treated as

“(100 / 10) * 10”.

Precedence and Associativity are two characteristics of operators that determine the evaluation order of subexpressions in absence of brackets.

1) Associativity is only used when there are two or more operators of same precedence.
The point to note is associativity doesn’t define the order in which operands of a single operator are evaluated. For example consider the following program, associativity of the + operator is left to right, but it doesn’t mean f1() is always called before f2(). The output of following program is in-fact compiler dependent. See this for details.

```
// Associativity is not used in the below program. Output
// is compiler dependent.
int x = 0;
int f1() {
  x = 5;
  return x;
}
int f2() {
  x = 10;
  return x;
}
int main() {
  int p = f1() + f2();
  printf("%d ", x);
  return 0;
}
```

2) All operators with same precedence have same associativity
This is necessary, otherwise there won’t be any way for compiler to decide evaluation order of expressions which have two operators of same precedence and different associativity. For example + and – have same associativity.

3) Precedence and associativity of postfix ++ and prefix ++ are different
Precedence of postfix ++ is more than prefix ++, their associativity is also different. Associativity of postfix ++ is left to right and associativity of prefix ++. See this for examples.

4) Comma has the least precedence among all operators and should be used carefully For example consider the following program, the output is 1. See this and this for more details.

```
#include<stdio.h>
int main()
{
    int a;
    a = 1, 2, 3; // Evaluated as (a = 1), 2, 3
    printf("%d", a);
    return 0;
}
```

5) There is no chaining of comparison operators in C
In Python, expression like “c > b > a” is treated as “a > b and b > c”, but this type of chaining doesn’t happen in C. For example consider the following program. The output of following program is “FALSE”.

```
#include <stdio.h>
int main()
{
  int a = 10, b = 20, c = 30;

  // (c > b > a) is treated as ((c > b) > a), associativity of '>'
  // is left to right. Therefore the value becomes ((30 > 20) > 10)
  // which becomes (1 > 20)
  if (c > b > a)
   printf("TRUE");
  else
    printf("FALSE");
  return 0;
}
```

Please see the following precedence and associativity table for reference.

OPERATOR	DESCRIPTION	ASSOCIATIVITY
( )	Parentheses (function call) (see Note 1)	left-to-right
[ ]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ —	Postfix increment/decrement (see Note 2)	
++ —	Prefix increment/decrement	right-to-left
+ –	Unary plus/minus	
! ~	Logical negation/bitwise complement	
(*type*)	Cast (convert value to temporary value of *type*)	
*	Dereference	
&	Address (of operand)	
sizeof	Determine size in bytes on this implementation	
* / %	Multiplication/division/modulus	left-to-right
+ –	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <=	Relational less than/less than or equal to	left-to-right
> >=	Relational greater than/greater than or equal to	
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
| |	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
=	Assignment	right-to-left
+= -=	Addition/subtraction assignment	
*= /=	Multiplication/division assignment	
%= &=	Modulus/bitwise AND assignment	
^= |=	Bitwise exclusive/inclusive OR assignment	
<<= >>=	Bitwise shift left/right assignment	
,	Comma (separate expressions)	left-to-right

It is good to know precedence and associativity rules, but the best thing is to use brackets, especially for less commonly used operators (operators other than +, -, *.. etc). Brackets increase readability of the code as the reader doesn't have to see the table to find out the order.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-operator

Evaluation order of operands

Consider the below C/C++ program.

```
#include<stdio.h>
int x = 0;

int f1()
{
  x = 5;
  return x;
}

int f2()
{
  x = 10;
  return x;
}

int main()
{
  int p = f1() + f2();
  printf("%d ", x);
  getchar();
  return 0;
}
```

What would the output of the above program – '5' or '10'?
The output is undefined as the order of evaluation of f1() + f2() is not mandated by standard. The compiler is free to first call either f1() or f2(). Only when equal level precedence operators appear in an expression, the associativity comes into picture. For example, f1() + f2() + f3() will be considered as (f1() + f2()) + f3(). But among first pair, which function (the operand) evaluated first is not defined by the standard.

Thanks to Venki for suggesting the solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

Comma in C and C++

In C and C++, comma (,) can be used in two contexts:

1) Comma as an operator:
The comma operator (represented by the token ,) is a binary operator that evaluates its first operand and discards the result, it then evaluates the second operand and returns this value (and type). The comma operator has the lowest precedence of any C operator, and acts as a sequence point.

```
/* comma as an operator */
int i = (5, 10);  /* 10 is assigned to i*/
int j = (f1(), f2());  /* f1() is called (evaluated) first followed by f2().
                   The returned value of f2() is assigned to j */
```

2) Comma as a separator:
Comma acts as a separator when used with function calls and definitions, function like macros, variable declarations, enum declarations, and similar constructs.

```

  /* comma as a separator */
  int a = 1, b = 2;
  void fun(x, y);
```

The use of comma as a separator should not be confused with the use as an operator. For example, in below statement, f1() and f2() can be called in any order.

```
/* Comma acts as a separator here and doesn't enforce any sequence.
   Therefore, either f1() or f2() can be called first */
void fun(f1(), f2());
```

See this for C vs C++ differences of using comma operator.

You can try below programs to check your understanding of comma in C.

```
// PROGRAM 1
#include<stdio.h>
int main()
{
   int x = 10;
   int y = 15;

   printf("%d", (x, y));
   getchar();
   return 0;
}
```

```
// PROGRAM 2:  Thanks to Shekhu for suggesting this program
#include<stdio.h>
```

```
int main()
{
  int x = 10;
  int y = (x++, ++x);
  printf("%d", y);
  getchar();
  return 0;
}
```

```
// PROGRAM 3:  Thanks to Venki for suggesting this program
int main()
{
   int x = 10, y;

   // The following is equavalent to y = x++
   y = (x++, printf("x = %d\n", x), ++x, printf("x = %d\n", x), x++);

   // Note that last expression is evaluated
   // but side effect is not updated to y
   printf("y = %d\n", y);
   printf("x = %d\n", x);

   return 0;
}
```

References:
http://en.wikipedia.org/wiki/Comma_operator
http://publib.boulder.ibm.com/infocenter/comphelp/v101v121/index.jsp?topic=/com.ibm.xlcpp101.aix.doc/language_ref/co.html
http://msdn.microsoft.com/en-us/library/zs06xbxh.aspx

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

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;
}
```

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 a = 0;
   double d = 10.21;
   printf("%d", sizeof(a+d));
   return 0;
}
```

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;
}
```

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));
```

References
https://en.wikipedia.org/wiki/Sizeof

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C

Operands for sizeof operator

In C, sizeof operator works on following kind of operands:

1) *type-name*: type-name must be specified in parentheses.

```
sizeof (type-name)
```

2) *expression*: expression can be specified with or without the parentheses.

```
sizeof expression
```

The expression is used only for getting the type of operand and not evaluated. For example, below code prints value of i as 5.

```
#include <stdio.h>

int main()
{
 int i = 5;
 int int_size = sizeof(i++);
 printf("\n size of i = %d", int_size);
 printf("\n Value of i = %d", i);

 getchar();
 return 0;
}
```

Output of the above program:
size of i = depends on compiler
value of i = 5

References:
http://h30097.www3.hp.com/docs/base_doc/DOCUMENTATION/V40G_HTML/AQTLTCTE/DOCU0015.HTM
http://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html#The-sizeof-Operator

C/C++ Puzzles
GFacts

A comma operator question

Consider the following C programs.

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

int main(void)
{
   int a = 1, 2, 3;
   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;
   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).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

Result of comma operator as l-value in C and C++

Using result of comma operator as l-value is not valid in C. But in C++, result of comma operator can be used as l-value if the right operand of the comma operator is l-value.

For example, if we compile the following program as a C++ program, then it works and prints b = 30. And if we compile the same program as C program, then it gives warning/error in compilation (Warning in Dev C++ and error in Code Blocks).

```
#include<stdio.h>

int main()
{
  int a = 10, b = 20;
  (a, b) = 30; // Since b is l-value, this statement is valid in C++, but not in C.
  printf("b = %d", b);
  getchar();
  return 0;
}
```

C++ Output:
b = 30

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

Order of operands for logical operators

The order of operands of logical operators &&, || are important in C/C++.

In mathematics, logical AND, OR, etc… operations are commutative. The result will not change even if we swap RHS and LHS of the operator.

In C/C++ (may be in other languages as well) even though these operators are commutative, their order is critical. For example see the following code,

```
// Traverse every alternative node
while( pTemp && pTemp->Next )
```

```
{
  // Jump over to next node
  pTemp = pTemp->Next->Next;
}
```

The first part *pTemp* will be evaluated against NULL and followed by *pTemp->Next*. If *pTemp->Next* is placed first, the pointer *pTemp* will be dereferenced and there will be runtime error when *pTemp* is NULL.

It is mandatory to follow the order. Infact, it helps in generating efficient code. When the pointer *pTemp* is NULL, the second part will not be evaluated since the outcome of AND (&&) expression is guaranteed to be 0.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
About Venki
Software Engineer
View all posts by Venki →

Increment (Decrement) operators require L-value Expression

What will be the output of the following program?

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

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

Answer: D, None – Compilation Error.

Explanation:

In C/C++ the pre-increment (decrement) and the post-increment (decrement) operators require an L-value expression as operand. Providing an R-value or a *const* qualified variable results in compilation error.

In the above program, the expression *-i* results in R-value which is operand of pre-increment operator. The pre-increment operator requires an L-value as operand, hence the compiler throws an error.

The increment/decrement operators needs to update the operand after the sequence point, so they need an L-value. The unary operators such as -, +, won't need L-value as operand. The expression *-(++i)* is valid.

In C++ the rules are little complicated because of references. We can apply these pre/post increment (decrement) operators on references variables that are not qualified by *const*. References can also be returned from functions.

Puzzle phrased by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
About Venki
Software Engineer
View all posts by Venki →

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;
  printf(" %c", *p);
  getchar();
  return 0;
}
```

Output: h

```
// Program 2
#include<stdio.h>
int main()
{
  char arr[] = "geeksforgeeks";
  char *p = arr;
  *p++;
  printf(" %c", *p);
  getchar();
  return 0;
}
```

Output: e

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

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***.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
About Venki
Software Engineer
View all posts by Venki →

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 *exp2* is considered as output type and *exp3* will be converted into *exp2* 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.

Best Practice:

It is the power of C++ type system that avoids such bugs. Make sure both the expressions *exp2* and *exp3* return same type or atleast safely convertible types. We can see other idioms like C++ *convert union* for safe conversion.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above. We will be happy to learn and update from other geeks.

Pre-increment (or pre-decrement) in C++

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;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Difference between ++*p, *p++ and *++p

Predict the output of following C programs.

```
// PROGRAM 1
#include <stdio.h>
int main(void)
{
    int arr[] = {10, 20};
    int *p = arr;
    ++*p;
    printf("arr[0] = %d, arr[1] = %d, *p = %d", arr[0], arr[1], *p);
    return 0;
}
```

```
// PROGRAM 2
#include <stdio.h>
int main(void)
{
    int arr[] = {10, 20};
    int *p = arr;
    *p++;
    printf("arr[0] = %d, arr[1] = %d, *p = %d", arr[0], arr[1], *p);
    return 0;
}
```

```
// PROGRAM 3
#include <stdio.h>
int main(void)
{
    int arr[] = {10, 20};
    int *p = arr;
    *++p;
    printf("arr[0] = %d, arr[1] = %d, *p = %d", arr[0], arr[1], *p);
    return 0;
}
```

The output of above programs and all such programs can be easily guessed by remembering following simple rules about postfix ++, prefix ++ and * (dereference) operators

1) Precedence of prefix ++ and * is same. Associativity of both is right to left.

2) Precedence of postfix ++ is higher than both * and prefix ++. Associativity of postfix ++ is left to right.

(Refer: Precedence Table)

The expression **++*p** has two operators of same precedence, so compiler looks for assoiativity. Associativity of operators is right to left. Therefore the expression is treated as ***++(*p)***. Therefore the output of first program is "*arr[0] = 11, arr[1] = 20, *p = 11*".

The expression ***p++** is treated as ***(p++)*** as the precedence of postfix ++ is higher than *. Therefore the output of second program is "*arr[0] = 10, arr[1] = 20, *p = 20*".

The expression ***++p** has two operators of same precedence, so compiler looks for assoiativity. Associativity of operators is right to left. Therefore the expression is treated as ***(++p)***. Therefore the output of second program is "*arr[0] = 10, arr[1] = 20, *p = 20*".

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-operator
cpp-pointer

Results of comparison operations in C and C++

In C, data type of result of comparison operations is int. For example, see the following program.

```
#include<stdio.h>
int main()
{
    int x = 10, y = 10;
    printf("%d \n", sizeof(x == y));
    printf("%d \n", sizeof(x < y));
    return 0;
}
```

Output:

```
4
4
```

Whereas in C++, type of results of comparison operations is bool. For example, see the following program.

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

int main()
{
    int x = 10, y = 10;
    cout << sizeof(x == y) << endl;
    cout << sizeof(x < y);
    return 0;
}
```

Output:

```
1
1
```

This article is contributed by **Rajat**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-operator

To find sum of two numbers without using any operator

Write a C program to find sum of positive integers without using any operator. Only use of printf() is allowed. No other library function can be used.

Solution

It's a trick question. We can use printf() to find sum of two numbers as printf() returns the number of characters printed. The width field in printf() can be used to find the sum of two numbers. We can use '*' which indicates the minimum width of output. For example, in the statement "printf("%*d", width, num);", the specified 'width' is substituted in place of *, and 'num' is printed within the minimum width specified. If number of digits in 'num' is smaller than the specified 'wodth', the output is padded with blank spaces. If number of digits are more, the output is printed as it is (not truncated). In the following program, add() returns sum of x and y. It prints 2 spaces within the width specified using x and y. So total characters printed is equal to sum of x and y. That is why add() returns x+y.

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

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

Output:

```
    Sum = 7
```

The output is seven spaces followed by “Sum = 7”. We can avoid the leading spaces by using carriage return. Thanks to krazyCoder and Sandeep for suggesting this. The following program prints output without any leading spaces.

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

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

Output:

```
Sum = 7
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

Sequence Points in C | Set 1

In this post, we will try to cover many ambiguous questions like following.

Guess the output of following programs.

```
// PROGRAM 1
#include <stdio.h>
int f1() { printf ("Geeks"); return 1;}
int f2() { printf ("forGeeks"); return 1;}
int main()
{
  int p = f1() + f2();
  return 0;
}

// PROGRAM 2
#include <stdio.h>
int x = 20;
int f1() { x = x+10; return x;}
int f2() { x = x-5;  return x;}
int main()
{
  int p = f1() + f2();
  printf ("p = %d", p);
  return 0;
}


// PROGRAM 3
#include <stdio.h>
int main()
{
   int i = 8;
   int p = i++*i++;
   printf("%d\n", p);
}
```

The output of all of the above programs is undefined or unspecified. The output may be different with different compilers and different machines. It is like asking the value of undefined automatic variable.

The reason for undefined behavior in PROGRAM 1 is, the operator ‘+’ doesn’t have standard defined order of evaluation for its operands. Either f1() or f2() may be executed first. So output may be either “GeeksforGeeks” or “forGeeksGeeks”.

Similar to operator '+', most of the other similar operators like '-', '/', '*', Bitwise AND &, Bitwise OR |, .. etc don't have a standard defined order for evaluation for its operands.

Evaluation of an expression may also produce side effects. For example, in the above program 2, the final values of p is ambiguous. Depending on the order of expression evaluation, if f1() executes first, the value of p will be 55, otherwise 40.

The output of program 3 is also undefined. It may be 64, 72, or may be something else. The subexpression i++ causes a side effect, it modifies i's value, which leads to undefined behavior since i is also referenced elsewhere in the same expression.

Unlike above cases, *at certain specified points in the execution sequence called sequence points, all side effects of previous evaluations are guaranteed to be complete*. A sequence point defines any point in a computer program's execution at which it is guaranteed that all side effects of previous evaluations will have been performed, and no side effects from subsequent evaluations have yet been performed. Following are the sequence points listed in the C standard:

— The end of the first operand of the following operators:
a) logical AND &&
b) logical OR ||
c) conditional ?
d) comma ,

For example, the output of following programs is guaranteed to be "GeeksforGeeks" on all compilers/machines.

```
// Following 3 lines are common in all of the below programs
#include <stdio.h>
int f1() { printf ("Geeks"); return 1;}
int f2() { printf ("forGeeks"); return 1;}

// PROGRAM 4
int main()
{
   // Since && defines a sequence point after first operand, it is
   // guaranteed that f1() is completed first.
   int p = f1() && f2();
   return 0;
}

// PROGRAM 5
int main()
{
   // Since comma operator defines a sequence point after first operand, it is
   // guaranteed that f1() is completed first.
  int p = (f1(), f2());
  return 0;
}


// PROGRAM 6
int main()
{
   // Since ? operator defines a sequence point after first operand, it is
   // guaranteed that f1() is completed first.
  int p = f1()? f2(): 3;
  return 0;
}
```

— The end of a full expression. This category includes following expression statements
a) Any full statement ended with semicolon like "a = b;"
b) return statements
c) The controlling expressions of if, switch, while, or do-while statements.
d) All three expressions in a for statement.

The above list of sequence points is partial. We will be covering all remaining sequence points in the next post on Sequence Point. We will also be covering many C questions asked in forum (See this, this, this , this and many others).

References:
http://en.wikipedia.org/wiki/Sequence_point
http://c-faq.com/expr/seqpoints.html
http://msdn.microsoft.com/en-us/library/d45c7a5d(v=vs.110).aspx

http://www.open-std.org/jtc1/sc22/wg14/www/docs/n925.htm

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
C++

Execution of printf with ++ operators

Consider below C++ program and predict its output.

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

The above invokes undefined behaviour by referencing both 'i' and 'i++' in the argument list. It is not defined which order the arguments are evaluated. Different compilers may choose different orders. A single compiler can also choose different orders at different times.

For example below three printf statements also cause undefined behavior.

```
// All three pritf() statements in this cause undefined behavior
#include<stdio.h>

int main()
{
   int a = 10;
   printf("\n %d %d", a, a++);
   a = 10;
   printf("\n %d %d", a++, a);
   a = 10;
   printf("\n %d %d %d ", a, a++, ++a);
   return 0;
}
```

Therefore, it is not recommended to do two or more than two pre or post increment operators in the same statement.

This article is contributed by **Spoorthi Aman**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-input-output
cpp-operator

Write a C macro PRINT(x) which prints x

At the first look, it seems that writing a C macro which prints its argument is child's play. Following program should work i.e. it should print *x*

```
#define PRINT(x) (x)
int main()
{
  printf("%s",PRINT(x));
  return 0;
}
```

But it would issue compile error because the data type of *x*, which is taken as variable by the compiler, is unknown. Now it doesn't look so obvious. Isn't it? Guess what, the followings also won't work

```
#define PRINT(x) ('x')
#define PRINT(x) ("x")
```

But if we know one of lesser known traits of C language, writing such a macro is really a child's play. In C, there's a # directive, also called 'Stringizing Operator', which does this magic. Basically # directive converts its argument in a string. Voila! it is so simple to do the rest. So the above program can be modified as below.

```
#define PRINT(x) (#x)
int main()
{
  printf("%s",PRINT(x));
  return 0;
}
```

Now if the input is *PRINT(x)*, it would print *x*. In fact, if the input is *PRINT(geeks)*, it would print *geeks*.

You may find the details of this directive from Microsoft portal **here**.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
C Macro
c puzzle
puzzle

Variable length arguments for Macros

Like functions, we can also pass variable length arguments to macros. For this we will use the following preprocessor identifiers.

To support variable length arguments in macro, we must include ellipses (…) in macro definition. There is also "__VA_ARGS__" preprocessing identifier which takes care of variable length argument substitutions which are provided to macro. Concatenation operator ## (aka paste operator) is used to concatenate variable arguments.

Let us see with example. Below macro takes variable length argument like "printf()" function. This macro is for error logging. The macro prints filename followed by line number, and finally it prints info/error message. First arguments "prio" determines the priority of message, i.e. whether it is information message or error, "stream" may be "standard output" or "standard error". It displays INFO messages on stdout and ERROR messages on stderr stream.

```
#include <stdio.h>

#define INFO  1
#define ERR 2
#define STD_OUT stdout
#define STD_ERR stderr

#define LOG_MESSAGE(prio, stream, msg, ...) do {\
      char *str;\
      if (prio == INFO)\
       str = "INFO";\
      else if (prio == ERR)\
       str = "ERR";\
      fprintf(stream, "[%s] : %s : %d : "msg" \n", \
                          str, __FILE__, __LINE__, ##__VA_ARGS__);\
     } while (0)

int main(void)
{
 char *s = "Hello";

        /* display normal message */
 LOG_MESSAGE(ERR, STD_ERR, "Failed to open file");

 /* provide string as argument */
 LOG_MESSAGE(INFO, STD_OUT, "%s Geeks for Geeks", s);

 /* provide integer as arguments */
 LOG_MESSAGE(INFO, STD_OUT, "%d + %d = %d", 10, 20, (10 + 20));

 return 0;
}
```

Compile and run the above program, it produces below result.

```
[narendra@/media/partition/GFG]$ ./variable_length
[ERR] : variable_length.c : 26 : Failed to open file
[INFO] : variable_length.c : 27 : Hello Geeks for Geeks
[INFO] : variable_length.c : 28 : 10 + 20 = 30
[narendra@/media/partition/GFG]$
```

This article is compiled by **Narendra Kangralkar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
CPuzzle

Multiline macros in C

In this article, we will discuss how to write a multi-line macro. We can write multi-line macro same like function, but each statement ends with "\". Let us see with example. Below is simple macro, which accepts input number from user, and prints whether entered number is even or odd.

```
#include <stdio.h>

#define MACRO(num, str) {\
  printf("%d", num);\
  printf(" is");\
  printf(" %s number", str);\
  printf("\n");\
    }

int main(void)
{
    int num;

    printf("Enter a number: ");
    scanf("%d", &num);

    if (num & 1)
        MACRO(num, "Odd");
    else
        MACRO(num, "Even");

    return 0;
}
```

At first look, the code looks OK, but when we try to compile this code, it gives compilation error.

```
[narendra@/media/partition/GFG]$ make macro
cc     macro.c   -o macro
macro.c: In function 'main':
macro.c:19:2: error: 'else' without a previous 'if'
make: *** [macro] Error 1
[narendra@/media/partition/GFG]$
```

Let us see what mistake we did while writing macro. We have enclosed macro in curly braces. According to C-language rule, each C-statement should end with semicolon. That's why we have ended MACRO with semicolon. Here is a mistake. Let us see how compile expands this macro.

```
if (num & 1)
{
    -------------------------
    ---- Macro expansion ----
    -------------------------
};    /* Semicolon at the end of MACRO, and here is ERROR */

else
```

```
{
   -------------------------
   ---- Macro expansion ----
   -------------------------

};
```

We have ended macro with semicolon. When compiler expands macro, it puts semicolon after “if” statement. Because of semicolon between “if and else statement” compiler gives compilation error. Above program will work fine, if we ignore “else” part.

To overcome this limitation, we can enclose our macro in “do-while(0)” statement. Our modified macro will look like this.

```
#include <stdio.h>

#define MACRO(num, str) do {\
   printf("%d", num);\
   printf(" is");\
   printf(" %s number", str);\
   printf("\n");\
     } while(0)

int main(void)
{
    int num;

    printf("Enter a number: ");
    scanf("%d", &num);

    if (num & 1)
        MACRO(num, "Odd");
    else
        MACRO(num, "Even");

    return 0;
}
```

Compile and run above code, now this code will work fine.

```
[narendra@/media/partition/GFG]$ make macro
cc     macro.c   -o macro
[narendra@/media/partition/GFG]$ ./macro
Enter a number: 9
9 is Odd number
[narendra@/media/partition/GFG]$ ./macro
Enter a number: 10
10 is Even number
[narendra@/media/partition/GFG]$
```

We have enclosed macro in “do – while(0)” loop and at the end of while, we have put condition as “while(0)”, that’s why this loop will execute only one time.

Similarly, instead of “do – while(0)” loop we can enclose multi-line macro in parenthesis. We can achieve the same result by using this trick. Let us see example.

```
#include <stdio.h>

#define MACRO(num, str) ({\
   printf("%d", num);\
   printf(" is");\
   printf(" %s number", str);\
   printf("\n");\
     })

int main(void)
{
    int num;

    printf("Enter a number: ");
    scanf("%d", &num);
    if (num & 1)
        MACRO(num, "Odd");
    else
        MACRO(num, "Even");

    return 0;
}
```

```
[narendra@/media/partition/GFG]$ make macro
cc     macro.c   -o macro
[narendra@/media/partition/GFG]$ ./macro
Enter a number: 10
10 is Even number
[narendra@/media/partition/GFG]$ ./macro
Enter a number: 15
15 is Odd number
[narendra@/media/partition/GFG]$
```

This article is compiled by **Narendra Kangralkar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

CRASH() macro – interpretation

Given below a small piece of code from an open source project,

```
#ifndef __cplusplus

typedef enum BoolenTag
{
   false,
   true
} bool;

#endif

#define CRASH() do { \
      ((void(*)())0)(); \
   } while(false)

int main()
{
   CRASH();
   return 0;
}
```

Can you interpret above code?

It is simple, a step by step approach is given below,

The statement *while(false)* is meant only for testing purpose. Consider the following operation,

```
((void(*)())0)();
```

It can be achieved as follows,

```
0;                    /* literal zero */
(0); ( ()0 );               /* 0 being casted to some type */
( (*) 0 );            /* 0 casted some pointer type */
( (*)() 0 );          /* 0 casted as pointer to some function */
( void (*)(void) 0 );   /* Interpret 0 as address of function
 taking nothing and returning nothing */
( void (*)(void) 0 )(); /* Invoke the function */
```

So the given code is invoking the function whose code is stored at location zero, in other words, trying to execute an instruction stored at location zero. On systems with memory protection (MMU) the OS will throw an exception (segmentation fault) and on systems without such protection (small embedded systems), it will execute and error will propagate further.

— **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
About Venki
Software Engineer
View all posts by Venki →

The OFFSETOF() macro

We know that the elements in a structure will be stored in sequential order of their declaration.

How to extract the displacement of an element in a structure? We can make use of offsetof macro.

Usually we call structure and union types (or *classes with trivial constructors*) as *plain old data* (POD) types, which will be used to *aggregate other data types*. The following non-standard macro can be used to get the displacement of an element in bytes from the base address of the structure variable.

```
#define OFFSETOF(TYPE, ELEMENT) ((size_t)&(((TYPE *)0)->ELEMENT))
```

Zero is casted to type of structure and required element's address is accessed, which is casted to *size_t*. As per standard *size_t* is of type *unsigned int*. The overall expression results in the number of bytes after which the ELEMENT being placed in the structure.

For example, the following code returns 16 bytes (padding is considered on 32 bit machine) as displacement of the character variable *c* in the structure Pod.

```
#include <stdio.h>

#define OFFSETOF(TYPE, ELEMENT) ((size_t)&(((TYPE *)0)->ELEMENT))

typedef struct PodTag
{
   int    i;
   double  d;
   char    c;
} PodType;

int main()
{
   printf("%d", OFFSETOF(PodType, c) );

   getchar();
   return 0;
}
```

In the above code, the following expression will return the displacement of element *c* in the structure *PodType*.

```
OFFSETOF(PodType, c);
```

After preprocessing stage the above macro expands to

```
((size_t)&(((PodType *)0)->c))
```

Since we are considering 0 as address of the structure variable, c will be placed after 16 bytes of its base address i.e. 0x00 + 0x10. Applying & on the structure element (in this case it is c) returns the address of the element which is 0x10. Casting the address to *unsigned int* (size_t) results in number of bytes the element is placed in the structure.

Note: We may consider the address operator & is redundant. Without address operator in macro, the code de-references the element of structure placed at NULL address. It causes an access violation exception (segmentation fault) at runtime.

Note that there are other ways to implement offsetof macro according to compiler behaviour. The ultimate goal is to extract displacement of the element. ***We will see practical usage of offsetof macro in liked lists to connect similar objects (for example thread pool) in***

another article.

Article compiled by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:

1. Linux Kernel code.

2. http://msdn.microsoft.com/en-us/library/dz4y9b9a.aspx

3. GNU C/C++ Compiler Documentation

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
About Venki
Software Engineer
View all posts by Venki →

Branch prediction macros in GCC

One of the most used optimization techniques in the Linux kernel is " __builtin_expect". When working with conditional code (if-else statements), we often know which branch is true and which is not. If compiler knows this information in advance, it can generate most optimized code.

Let us see macro definition of "likely()" and "unlikely()" macros from linux kernel code "http://lxr.linux.no/linux+v3.6.5/include/linux/compiler.h" [line no 146 and 147].

```
#define likely(x)      __builtin_expect(!!(x), 1)
#define unlikely(x)    __builtin_expect(!!(x), 0)
```

In the following example, we are marking branch as likely true:

```
const char *home_dir ;

home_dir = getenv("HOME");
if (likely(home_dir))
 printf("home directory: %s\n", home_dir);
else
 perror("getenv");
```

For above example, we have marked "if" condition as "likely()" true, so compiler will put true code immediately after branch, and false code within the branch instruction. In this way compiler can achieve optimization. But don't use "likely()" and "unlikely()" macros blindly. If prediction is correct, it means there is zero cycle of jump instruction, but if prediction is wrong, then it will take several cycles, because processor needs to flush it's pipeline which is worst than no prediction.

Accessing memory is the slowest CPU operation as compared to other CPU operations. To avoid this limitation, CPU uses "CPU caches" e.g L1-cache, L2-cache etc. The idea behind cache is, copy some part of memory into CPU itself. We can access cache memory much faster than any other memory. But the problem is, limited size of "cache memory", we can't copy entire memory into cache. So, the CPU has to guess which memory is going to be used in the near future and load that memory into the CPU cache and above macros are hint to load memory into the CPU cache.

This article is compiled by **Narendra Kangralkar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
AdvanceC
CPuzzle
GCC

Diffference between #define and const in C?

#define is a preprocessor directive. Things defined by #define are replaced by the preprocessor before compilation begins.

const variables are actual variables like other normal variable.

The big advantage of const over #define is type checking. We can also have poitners to const varaibles, we can pass them around, typecast them and any other thing that can be done with a normal variable. One disadvantage that one could think of is extra space for variable which is immaterial due to optimizations done by compilers.

In general const is a better option if we have a choice. There are situations when #define cannot be replaced by const. For example, #define can take parameters (See this for example). #define can also be used to replace some text in a program with another text.

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C

A C Programming Language Puzzle

Give a = 12 and b = 36 write a C function/macro that returns 3612 without using arithmetic, strings and predefined functions.

We strongly recommend you to minimize your browser and try this yourself first.

Below is one solution that uses String Token-Pasting Operator (##) of C macros. For example, the expression “a##b” prints concatenation of ‘a’ and ‘b’.

Below is a working C code.

```
#include <stdio.h>
#define merge(a, b) b##a
int main(void)
{
   printf("%d ", merge(12, 36));
   return 0;
}
```

Output:

```
3612
```

Thanks to an anonymous user to suggest this solution here.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-macros
cpp-puzzle

What’s difference between header files “stdio.h” and “stdlib.h” ?

These are two important header files used in C programming. While “<stdio.h>” is header file for **St**andar**d** **I**nput **O**utput, “<stdlib.h>” is header file for **St**andar**d** **Lib**rary. One easy way to differentiate these two header files is that “<stdio.h>” contains declaration of *printf()* and *scanf()* while “<stdlib.h>” contains declaration of *malloc()* and *free()*. In that sense, the main difference in these two header files can considered that, while “<stdio.h>” contains header information for ‘File related Input/Output’ functions, “<stdlib.h>” contains header information for ‘Memory Allocation/Freeing’ functions.

Wait a minute, you said “<stdio.h>” is for file related IO but *printf()* and *scanf()* don’t deal with files… or are they? As a basic principle, in C (due to its association with UNIX history), keyboard and display are also treated as ‘files’! In fact keyboard input is the default *stdin* file stream while display output is the default *stdout* file stream. Also, please note that, though “<stdlib.h>” contains declaration of other types of functions as well that aren't related to memory such as *atoi()*, *exit()*, *rand()* etc. yet for our purpose and simplicity, we can remember *malloc()* and *free()* for "<stdlib.h>".

It should be noted that a header file can contain not only function declaration but definition of constants and variables as well. Even macros and definition of new data types can also be added in a header file.

Please do Like/Tweet/G+1 if you find the above useful. Also, please do leave us comment for further clarification or info. We would love to help and learn

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Articles C Difference

How to print a variable name in C?

How to print and store a variable name in string variable?

We strongly recommend you to minimize your browser and try this yourself first

In C, there's a # directive, also called 'Stringizing Operator', which does this magic. Basically # directive converts its argument in a string.

```
#include <stdio.h>
#define getName(var)  #var

int main()
{
 int myVar;
 printf("%s", getName(myVar));
 return 0;
}
```

Output:

```
MyVar
```

We can also store variable name in a string using sprintf() in C.

```
# include <stdio.h>
# define getName(var, str)  sprintf(str, "%s", #var)

int main()
{
 int myVar;
 char str[20];
 getName(myVar, str);
 printf("%s", str);
 return 0;
}
```

Output:

```
MyVar
```

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

Arrays in C Language | Set 1 (Introduction)

An array is collection of items stored at continuous memory locations. The idea is to declare multiple items of same type together.

arrays

Array declaration:

In C, we can declare an array by specifying its and size or by initializing it or by both.

```
// Array declaration by specifying size
int arr[10];
```

```
// Array declaration by initializing elements
int arr[] = {10, 20, 30, 40}

// Compiler creates an array of size 4.
// above is same as  "int arr[4] = {10, 20, 30, 40}"
```

```
// Array declaration by specifying size and initializing
// elements
int arr[6] = {10, 20, 30, 40}

// Compiler creates an array of size 6, initializes first
// 4 elements as specified by user and rest two elements as 0.
// above is same as  "int arr[] = {10, 20, 30, 40, 0, 0}"
```

Accessing Array Elements:

Array elements are accessed by using an integer index. Array index starts with 0 and goes till size of array minus 1. Following are few examples.

```
int main()
{
  int arr[5];
  arr[0] = 5;
  arr[2] = -10;
  arr[3/2] = 2; // this is same as arr[1] = 2
  arr[3] = arr[0];

  printf("%d %d %d %d", arr[0], arr[1], arr[2], arr[3]);

  return 0;
}
```

Output:

```
5 2 -10 5
```

No Index Out of bound Checking:

There is no index out of bound checking in C, for example the following program compiles fine but may produce unexpected output when run.

```
// This C program compiles fine as index out of bound
// is not checked in C.
```

```
int main()
{
  int arr[2];

  printf("%d ", arr[3]);
  printf("%d ", arr[-2]);

  return 0;
}
```

Also, In C, it is not compiler error to initialize an array with more elements than specified size. For example the below program compiles fine.

```
int main()
{

  // Array declaration by initializing it with more
  // elements than specified size.
  int arr[2] = {10, 20, 30, 40, 50};

  return 0;
}
```

The program won't compile in C++. If we save the above program as a .cpp, the program generates compiler error "error: too many initializers for 'int [2]'"

An Example to show that array elements are stored at contiguous locations

```
// C program to demonstrate that array elements are stored
// contiguous locations
int main()
{
  // an array of 10 integers.  If arr[0] is stored at
  // address x, then arr[1] is stored at x + sizeof(int)
  // arr[2] is stored at x + sizeof(int) + sizeof(int)
  // and so on.
  int arr[5], i;

  printf("Size of integer in this compiler is %u\n", sizeof(int));

  for (i=0; i<5; i++)
    // The use of '&' before a variable name, yields
    // address of variable.
    printf("Address arr[%d] is %u\n", i, &arr[i]);

  return 0;
}
```

Output:

```
Size of integer in this compiler is 4
Address arr[0] is 2686728
Address arr[1] is 2686732
Address arr[2] is 2686736
Address arr[3] is 2686740
Address arr[4] is 2686744
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

Arrays in C Language | Set 2 (Properties)

We have introduced arrays in set 1 (Introduction to arrays in C).

In this post array properties in C are discussed.

1) In C, it is possible to have array of all types except void and functions. See this for details.

2) In C, array and pointer are different. They seem similar because array name gives address of first element and array elements are accessed using pointer arithmetic. See array vs pointer in C for details.

3) Arrays are always passed as pointer to functions. See this for details.

4) A character array initialized with double quoted string has last element as '\0′. See this for details.

5) Like other variables, arrays can be allocated memory in any of the three segments, data, heap, and stack (See this for details). Dynamically allocated arrays are allocated memory on heap, static or global arrays are allocated memory on data segment and local arrays are allocated memory on stack segment.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C

Do not use sizeof for array parameters

Consider the below program.

```
#include<stdio.h>
void fun(int arr[])
{
  int i;

  /* sizeof should not be used here to get number
    of elements in array*/
  int arr_size = sizeof(arr)/sizeof(arr[0]); /* incorrect use of siseof*/
  for (i = 0; i < arr_size; i++)
  {
    arr[i] = i;  /*executed only once */
  }
}

int main()
{
  int i;
  int arr[4] = {0, 0 ,0, 0};
  fun(arr);

  /* use of sizeof is fine here*/
  for(i = 0; i < sizeof(arr)/sizeof(arr[0]); i++)
    printf(" %d " ,arr[i]);

  getchar();
  return 0;
}
```

Output: 0 0 0 0 on a IA-32 machine.

The function fun() receives an array parameter arr[] and tries to find out number of elements in arr[] using sizeof operator.
In C, array parameters are treated as pointers (See http://geeksforgeeks.org/?p=4088 for details). So the expression sizeof(arr)/sizeof(arr[0]) becomes sizeof(int *)/sizeof(int) which results in 1 for IA 32 bit machine (size of int and int * is 4) and the for loop inside fun() is executed only once irrespective of the size of the array.

Therefore, sizeof should not be used to get number of elements in such cases. A separate parameter for array size (or length) should be passed to fun(). So the **corrected program is:**

```
#include<stdio.h>
void fun(int arr[], size_t arr_size)
{
  int i;
  for (i = 0; i < arr_size; i++)
  {
    arr[i] = i;
  }
}

int main()
{
  int i;
  int arr[4] = {0, 0 ,0, 0};
  fun(arr, 4);

  for(i = 0; i < sizeof(arr)/sizeof(arr[0]); i++)
    printf(" %d ", arr[i]);

  getchar();
  return 0;
}
```

Please write comments if you find anything incorrect in the above article or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

Articles
C/C++ Puzzles

Initialization of variables sized arrays in C

The C99 standard allows variable sized arrays (see this). But, unlike the normal arrays, variable sized arrays cannot be initialized.

For example, the following program compiles and runs fine on a C99 compatible compiler.

```
#include<stdio.h>

int main()
{
  int M = 2;
  int arr[M][M];
  int i, j;
  for (i = 0; i < M; i++)
  {
    for (j = 0; j < M; j++)
    {
       arr[i][j] = 0;
       printf ("%d ", arr[i][j]);
    }
    printf("\n");
  }
  return 0;
}
```

Output:

```
0 0
0 0
```

But the following fails with compilation error.

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

```
int main()
{
  int M = 2;
  int arr[M][M] = {0}; // Trying to initialize all values as 0
  int i, j;
  for (i = 0; i < M; i++)
  {
    for (j = 0; j < M; j++)
       printf ("%d ", arr[i][j]);
    printf("\n");
  }
  return 0;
}
```

Output:

```
Compiler Error: variable-sized object may not be initialized
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Are array members deeply copied?

In C/C++, we can assign a struct (or class in C++ only) variable to another variable of same type. When we assign a struct variable to another, all members of the variable are copied to the other struct variable. But what happens when the structure contains pointer to dynamically allocated memory and what if it contains an array?

In the following C++ program, struct variable st1 contains pointer to dynamically allocated memory. When we assign st1 to st2, str pointer of st2 also start pointing to same memory location. This kind of copying is called Shallow Copy.

```
# include <iostream>
# include <string.h>

using namespace std;

struct test
{
  char *str;
};

int main()
{
  struct test st1, st2;

  st1.str = new char[20];
  strcpy(st1.str, "GeeksforGeeks");

  st2 = st1;

  st1.str[0] = 'X';
  st1.str[1] = 'Y';

  /* Since copy was shallow, both strings are same */
  cout << "st1's str = " << st1.str << endl;
  cout << "st2's str = " << st2.str << endl;

  return 0;
}
```

Output:

```
st1’s str = XYeksforGeeks
st2’s str = XYeksforGeeks
```

Now, what about arrays? *The point to note is that the array members are not shallow copied, compiler automatically performs Deep Copy* *for array members*.. In the following program, struct test contains array member str[]. When we assign st1 to st2, st2 has a new copy of the array. So st2 is not changed when we change str[] of st1.

```
# include <iostream>
# include <string.h>

using namespace std;

struct test
{
  char str[20];
};

int main()
{
  struct test st1, st2;

  strcpy(st1.str, "GeeksforGeeks");

  st2 = st1;

  st1.str[0] = 'X';
  st1.str[1] = 'Y';

  /* Since copy was Deep, both arrays are different */
  cout << "st1's str = " << st1.str << endl;
  cout << "st2's str = " << st2.str << endl;

  return 0;
}
```

Output:

```
st1's str = XYeksforGeeks
st2's str = GeeksforGeeks
```

Therefore, for C++ classes, we don't need to write our own copy constructor and assignment operator for array members as the default behavior is Deep copy for arrays.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
pointer

What is the difference between single quoted and double quoted declaration of char array?

In C/C++, when a character array is initialized with a double quoted string and array size is not specified, compiler automatically allocates one extra space for string terminator '\0'. For example, following program prints 6 as output.

```
#include<stdio.h>
int main()
{
  char arr[] = "geeks"; // size of arr[] is 6 as it is '\0' terminated
  printf("%d", sizeof(arr));
  getchar();
  return 0;
}
```

If array size is specified as 5 in the above program then the program works without any warning/error and prints 5 in C, but causes compilation error in C++.

```
// Works in C, but compilation error in C++
#include<stdio.h>
int main()
{
  char arr[5] = "geeks";  // arr[] is not terminated with '\0'
                               // and its size is 5
  printf("%d", sizeof(arr));
  getchar();
  return 0;
}
```

When character array is initialized with comma separated list of characters and array size is not specified, compiler doesn’t create extra space for string terminator ‘\0’. For example, following program prints 5.

```
#include<stdio.h>
int main()
{
  char arr[]= {'g', 'e', 'e', 'k', 's'}; // arr[] is not terminated with '\0' and its size is 5
  printf("%d", sizeof(arr));
  getchar();
  return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

Initialization of a multidimensional arrays in C/C++

In C/C++, initialization of a multidimensional arrays can have left most dimension as optional. Except the left most dimension, all other dimensions must be specified.

For example, following program fails in compilation because two dimensions are not specified.

```
#include<stdio.h>
int main()
{
  int a[][][2] = { {{1, 2}, {3, 4}},
                   {{5, 6}, {7, 8}}
                 };  // error
  printf("%d", sizeof(a));
  getchar();
  return 0;
}
```

Following 2 programs work without any error.

```
// Program 1
#include<stdio.h>
int main()
{
  int a[][2] = {{1,2},{3,4}}; // Works
  printf("%d", sizeof(a)); // prints 4*sizeof(int)
  getchar();
  return 0;
}
```

```
// Program 2
#include<stdio.h>
int main()
{
  int a[][2][2] = { {{1, 2}, {3, 4}},
                    {{5, 6}, {7, 8}}
                  }; // Works
  printf("%d", sizeof(a)); // prints 8*sizeof(int)
  getchar();
  return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Write one line functions for strcat() and strcmp()

Recursion can be used to do both tasks in one line. Below are one line implementations for stracat() and strcmp().

```
/* my_strcat(dest, src) copies data of src to dest.  To do so, it first reaches end of the string dest using recursive calls my_strcat(++dest, src).  Once en
(*dest++ = *src++)?  my_strcat(dest, src). */
void my_strcat(char *dest, char *src)
{
  (*dest)? my_strcat(++dest, src): (*dest++ = *src++)? my_strcat(dest, src): 0 ;
}

/* driver function to test above function */
int main()
{
  char dest[100] = "geeksfor";
  char *src = "geeks";
  my_strcat(dest, src);
  printf(" %s ", dest);
  getchar();
}
```

The function my_strcmp() is simple compared to my_strcmp().

```
/* my_strcmp(a, b) returns 0 if strings a and b are same, otherwise 1.   It recursively increases a and b pointers. At any point if *a is not equal to *b the
int my_strcmp(char *a, char *b)
{
  return (*a == *b && *b == '\0')? 0 : (*a == *b)? my_strcmp(++a, ++b): 1;
}

/* driver function to test above function */
int main()
{
  char *a = "geeksforgeeks";
  char *b = "geeksforgeeks";
  if(my_strcmp(a, b) == 0)
     printf(" String are same ");
  else
     printf(" String are not same ");

  getchar();
  return 0;
}
```

The above functions do very basic string concatenation and string comparison. These functions do not provide same functionality as standard library functions.

Asked by geek4u

Please write comments if you find the above code incorrect, or find better ways to solve the same problem.

What’s difference between char s[] and char *s in C?

Consider below two statements in C. What is difference between two?

```
char s[] = "geeksquiz";
char *s  = "geeksquiz";
```

The statements '**char s[] = "geeksquiz"**' creates a character array which is like any other array and we can do all array operations. The only special thing about this array is, although we have initialized it with 9 elements, its size is 10 (Compiler automatically adds '\0')

```
#include <stdio.h>
int main()
{
    char s[] = "geeksquiz";
    printf("%lu", sizeof(s));
    s[0] = 'j';
    printf("\n%s", s);
    return 0;
}
```

Output:

```
10
jeeksquiz
```

The statement '**char *s = "geeksquiz"**' creates a string literal. The string literal is stored in read only part of memory by most of the compilers. The C and C++ standards say that string literals have static storage duration, any attempt at modifying them gives undefined behavior.
s is just a pointer and like any other pointer stores address of string literal.

```
#include <stdio.h>
int main()
{
    char *s = "geeksquiz";
    printf("%lu", sizeof(s));

    // Uncommenting below line would cause undefined behaviour
    // (Caused segmentation fault on gcc)
    //  s[0] = 'j';
    return 0;
}
```

Output:

```
8
```

Running above program may generates a warning also "warning: deprecated conversion from string constant to 'char*'". This warning occurs because s is not a const pointer, but stores address of read only location. The warning can be avoided by pointer to const.

```
#include <stdio.h>
int main()
{
    const char *s = "geeksquiz";
    printf("%lu", sizeof(s));
    return 0;
}
```

This article is contributed by Abhay Rathi. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C Difference

gets() is risky to use!

Asked by geek4u

Consider the below program.

```
void read()
{
  char str[20];
  gets(str);
  printf("%s", str);
  return;
}
```

The code looks simple, it reads string from standard input and prints the entered string, but it suffers from Buffer Overflow as gets() doesn't do any array bound testing. gets() keeps on reading until it sees a newline character.

To avoid Buffer Overflow, fgets() should be used instead of gets() as fgets() makes sure that not more than MAX_LIMIT characters are read.

```
#define MAX_LIMIT 20
void read()
{
  char str[MAX_LIMIT];
  fgets(str, MAX_LIMIT, stdin);
  printf("%s", str);

  getchar();
  return;
}
```

Please write comments if you find anything incorrect in the above article, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

Articles
C/C++ Puzzles
C-programming
gets

C function to Swap strings

Let us consider the below program.

```
#include<stdio.h>
void swap(char *str1, char *str2)
{
 char *temp = str1;
 str1 = str2;
 str2 = temp;
}

int main()
{
 char *str1 = "geeks";
 char *str2 = "forgeeks";
 swap(str1, str2);
 printf("str1 is %s, str2 is %s", str1, str2);
 getchar();
 return 0;
}
```

Output of the program is *str1 is geeks, str2 is forgeeks*. So the above swap() function doesn't swap strings. The function just changes local pointer variables and the changes are not reflected outside the function.

Let us see the correct ways for swapping strings:

Method 1(Swap Pointers)

If you are using character pointer for strings (not arrays) then change str1 and str2 to point each other's data. i.e., swap pointers. In a function, if we want to change a pointer (and obviously we want changes to be reflected outside the function) then we need to pass a pointer to the pointer.

```
#include<stdio.h>

/* Swaps strings by swapping pointers */
void swap1(char **str1_ptr, char **str2_ptr)
{
  char *temp = *str1_ptr;
  *str1_ptr = *str2_ptr;
  *str2_ptr = temp;
}

int main()
{
  char *str1 = "geeks";
  char *str2 = "forgeeks";
  swap1(&str1, &str2);
  printf("str1 is %s, str2 is %s", str1, str2);
  getchar();
  return 0;
}
```

This method cannot be applied if strings are stored using character arrays.

Method 2(Swap Data)

If you are using character arrays to store strings then preferred way is to swap the data of both arrays.

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

/* Swaps strings by swapping data*/
void swap2(char *str1, char *str2)
{
  char *temp = (char *)malloc((strlen(str1) + 1) * sizeof(char));
  strcpy(temp, str1);
  strcpy(str1, str2);
  strcpy(str2, temp);
  free(temp);
}

int main()
{
  char str1[10] = "geeks";
  char str2[10] = "forgeeks";
  swap2(str1, str2);
  printf("str1 is %s, str2 is %s", str1, str2);
  getchar();
  return 0;
}
```

This method cannot be applied for strings stored in read only block of memory.

Please write comments if you find anything incorrect in the above article, or you want to share more information about the topic discussed above.

Articles
C/C++ Puzzles

Storage for Strings in C

In C, a string can be referred either using a character pointer or as a character array.

Strings as character arrays

```
char str[4] = "GfG"; /*One extra for string terminator*/
/*    OR    */
char str[4] = {'G', 'f', 'G', '\0'}; /* '\0' is string terminator */
```

When strings are declared as character arrays, they are stored like other types of arrays in C. For example, if str[] is an auto variable then string is stored in stack segment, if it's a global or static variable then stored in data segment, etc.

Strings using character pointers
Using character pointer strings can be stored in two ways:

1) Read only string in a shared segment.
When string value is directly assigned to a pointer, in most of the compilers, it's stored in a read only block (generally in data segment) that is shared among functions.

```

  char *str  =  "GfG";
```

In the above line "GfG" is stored in a shared read only location, but pointer str is stored in a read-write memory. You can change str to point something else but cannot change value at present str. So this kind of string should only be used when we don't want to modify string at a later stage in program.

2) Dynamically allocated in heap segment.
Strings are stored like other dynamically allocated things in C and can be shared among functions.

```
  char *str;
  int size = 4; /*one extra for '\0'*/
  str = (char *)malloc(sizeof(char)*size);
  *(str+0) = 'G';
  *(str+1) = 'f';
  *(str+2) = 'G';
  *(str+3) = '\0';
```

Let us see some examples to better understand above ways to store strings.

Example 1 (Try to modify string)
The below program may crash (gives segmentation fault error) because the line *(str+1) = 'n' tries to write a read only memory.

```
int main()
{
 char *str;
 str = "GfG";     /* Stored in read only part of data segment */
 *(str+1) = 'n'; /* Problem:  trying to modify read only memory */
 getchar();
 return 0;
}
```

Below program works perfectly fine as str[] is stored in writable stack segment.

```
int main()
{
 char str[] = "GfG";  /* Stored in stack segment like other auto variables */
 *(str+1) = 'n';   /* No problem: String is now GnG */
 getchar();
 return 0;
}
```

Below program also works perfectly fine as data at str is stored in writable heap segment.

```
int main()
{
  int size = 4;

  /* Stored in heap segment like other dynamically allocated things */
  char *str = (char *)malloc(sizeof(char)*size);
  *(str+0) = 'G';
  *(str+1) = 'f';
  *(str+2) = 'G';
  *(str+3) = '\0';
  *(str+1) = 'n';  /* No problem: String is now GnG */
   getchar();
   return 0;
}
```

Example 2 (Try to return string from a function)

The below program works perfectly fine as the string is stored in a shared segment and data stored remains there even after return of getString()

```
char *getString()
{
  char *str = "GfG"; /* Stored in read only part of shared segment */

  /* No problem: remains at address str after getString() returns*/
  return str;
}

int main()
{
  printf("%s", getString());
  getchar();
  return 0;
}
```

The below program alse works perfectly fine as the string is stored in heap segment and data stored in heap segment persists even after return of getString()

```
char *getString()
{
  int size = 4;
  char *str = (char *)malloc(sizeof(char)*size); /*Stored in heap segment*/
  *(str+0) = 'G';
  *(str+1) = 'f';
  *(str+2) = 'G';
  *(str+3) = '\0';

  /* No problem: string remains at str after getString() returns */
  return str;
}
int main()
{
  printf("%s", getString());
  getchar();
  return 0;
}
```

But, the below program may print some garbage data as string is stored in stack frame of function getString() and data may not be there after getString() returns.

```
char *getString()
{
  char str[] = "GfG"; /* Stored in stack segment */

  /* Problem: string may not be present after getSting() returns */
  return str;
}
int main()
{
  printf("%s", getString());
  getchar();
  return 0;
}
```

Please write comments if you find anything incorrect in the above article, or you want to share more information about storage of strings

Difference between pointer and array in C?

Pointers are used for storing address of dynamically allocated arrays and for arrays which are passed as arguments to functions. In other contexts, arrays and pointer are two different things, see the following programs to justify this statement.

Behavior of sizeof operator

```
// 1st program to show that array and pointers are different
#include <stdio.h>
int main()
{
   int arr[] = {10, 20, 30, 40, 50, 60};
   int *ptr = arr;

   // sizof(int) * (number of element in arr[]) is printed
   printf("Size of arr[] %d\n", sizeof(arr));

   // sizeof a pointer is printed which is same for all type
   // of pointers (char *, void *, etc)
   printf("Size of ptr %d", sizeof(ptr));
   return 0;
}
```

Output:

```
Size of arr[] 24
Size of ptr 4
```

Assigning any address to an array variable is not allowed.

```
// IInd program to show that array and pointers are different
#include <stdio.h>
int main()
{
   int arr[] = {10, 20}, x = 10;
   int *ptr = &x; // This is fine
   arr = &x;  // Compiler Error
   return 0;
}
```

Output:

```
 Compiler Error: incompatible types when assigning to
            type 'int[2]' from type 'int *'
```

See the previous post on this topic for more differences.

Although array and pointer are different things, following properties of array make them look similar.

1) *Array name gives address of first element of array.*
Consider the following program for example.

```
#include <stdio.h>
int main()
{
   int arr[] = {10, 20, 30, 40, 50, 60};
   int *ptr = arr;  // Assigns address of array to ptr
   printf("Value of first element is %d", *ptr)
   return 0;
}
```

Output:

```
Value of first element is 10
```

2) *Array members are accessed using pointer arithmetic.*

Compiler uses pointer arithmetic to access array element. For example, an expression like “arr[i]” is treated as *(arr + i) by the compiler. That is why the expressions like *(arr + i) work for array arr, and expressions like ptr[i] also work for pointer ptr.

```
#include <stdio.h>
int main()
{
  int arr[] = {10, 20, 30, 40, 50, 60};
  int *ptr = arr;
  printf("arr[2] = %d\n", arr[2]);
  printf("*(arr + 2) = %d\n", *(arr + 2));
  printf("ptr[2] = %d\n", ptr[2]);
  printf("*(ptr + 2) = %d\n", *(ptr + 2));
  return 0;
}
```

Output:

```
arr[2] = 30
*(arr + 2) = 30
ptr[2] = 30
*(ptr + 2) = 30
```

3) *Array parameters are always passed as pointers, even when we use square brackets.*

```
#include <stdio.h>

int fun(int ptr[])
{
  int x = 10;

  // size of a pointer is printed
  printf("sizeof(ptr) = %d\n", sizeof(ptr));

  // This allowed because ptr is a pointer, not array
  ptr = &x;

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

  return 0;
}
int main()
{
  int arr[] = {10, 20, 30, 40, 50, 60};
  fun(arr);
  return 0;
}
```

Output:

```
sizeof(ptr) = 4
*ptr = 10
```

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-array
cpp-pointer

How to dynamically allocate a 2D array in C?

Following are different ways to create a 2D array on heap (or dynamically allocate a 2D array).

In the following examples, we have considered '**r**' as number of rows, '**c**' as number of columns and we created a 2D array with r = 3, c = 4 and following values

```
 1  2  3  4
 5  6  7  8
 9  10 11 12
```

1) Using a single pointer:

A simple way is to allocate memory block of size r*c and access elements using simple pointer arithmetic.

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

int main()
{
    int r = 3, c = 4;
    int *arr = (int *)malloc(r * c * sizeof(int));

    int i, j, count = 0;
    for (i = 0; i <  r; i++)
      for (j = 0; j < c; j++)
         *(arr + i*c + j) = ++count;

    for (i = 0; i <  r; i++)
      for (j = 0; j < c; j++)
         printf("%d ", *(arr + i*c + j));

   /* Code for further processing and free the
      dynamically allocated memory */

   return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12
```

2) Using an array of pointers

We can create an array of pointers of size r. Note that from C99, C language allows variable sized arrays. After creating an array of pointers, we can dynamically allocate memory for every row.

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

int main()
{
    int r = 3, c = 4, i, j, count;

    int *arr[r];
    for (i=0; i<r; i++)
         arr[i] = (int *)malloc(c * sizeof(int));

    // Note that arr[i][j] is same as *(*(arr+i)+j)
    count = 0;
    for (i = 0; i <  r; i++)
      for (j = 0; j < c; j++)
         arr[i][j] = ++count; // Or *(*(arr+i)+j) = ++count

    for (i = 0; i <  r; i++)
      for (j = 0; j < c; j++)
         printf("%d ", arr[i][j]);

    /* Code for further processing and free the
      dynamically allocated memory */

   return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12
```

3) Using pointer to a pointer

We can create an array of pointers also dynamically using a double pointer. Once we have an array pointers allocated dynamically, we can dynamically allocate memory and for every row like method 2.

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

int main()
{
    int r = 3, c = 4, i, j, count;

    int **arr = (int **)malloc(r * sizeof(int *));
    for (i=0; i<r; i++)
         arr[i] = (int *)malloc(c * sizeof(int));

    // Note that arr[i][j] is same as *(*(arr+i)+j)
    count = 0;
    for (i = 0; i <  r; i++)
      for (j = 0; j < c; j++)
         arr[i][j] = ++count;  // OR *(*(arr+i)+j) = ++count

    for (i = 0; i <  r; i++)
      for (j = 0; j < c; j++)
         printf("%d ", arr[i][j]);

   /* Code for further processing and free the
      dynamically allocated memory */

   return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12
```

4) Using double pointer and one malloc call for all rows

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

int main()
{
    int r=3, c=4;
    int **arr;
    int count = 0,i,j;

    arr  = (int **)malloc(sizeof(int *) * r);
    arr[0] = (int *)malloc(sizeof(int) * c * r);

    for(i = 0; i < r; i++)
        arr[i] = (*arr + c * i);

    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            arr[i][j] = ++count;  // OR *(*(arr+i)+j) = ++count

    for (i = 0; i <  r; i++)
        for (j = 0; j < c; j++)
            printf("%d ", arr[i][j]);

    return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12
```

Thanks to Trishansh Bhardwaj for suggesting this 4th method.

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-array
cpp-pointer

How to pass a 2D array as a parameter in C?

This post is an extension of How to dynamically allocate a 2D array in C?

A one dimensional array can be easily passed as a pointer, but syntax for passing a 2D array to a function can be difficult to remember. One important thing for passing multidimensional arrays is, first array dimension does not have to be specified. The second (and any subsequent) dimensions must be given

1) When both dimensions are available globally (either as a macro or as a global constant).

```
#include <stdio.h>
const int M = 3;
const int N = 3;

void print(int arr[M][N])
{
    int i, j;
    for (i = 0; i < M; i++)
      for (j = 0; j < N; j++)
        printf("%d ", arr[i][j]);
}

int main()
{
    int arr[][N] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    print(arr);
    return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9
```

2) When only second dimension is available globally (either as a macro or as a global constant).

```
#include <stdio.h>
const int N = 3;

void print(int arr[][N], int m)
{
    int i, j;
    for (i = 0; i < m; i++)
      for (j = 0; j < N; j++)
        printf("%d ", arr[i][j]);
}

int main()
{
    int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    print(arr, 3);
    return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9
```

The above method is fine if second dimension is fixed and is not user specified. The following methods handle cases when second dimension can also change.

3) If compiler is C99 compatible
From C99, C language supports variable sized arrays to be passed simply by specifying the variable dimensions (See this for an example run)

```
// The following program works only if your compiler is C99 compatible.
#include <stdio.h>

// n must be passed before the 2D array
void print(int m, int n, int arr[][n])
{
    int i, j;
    for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
        printf("%d ", arr[i][j]);
}

int main()
{
    int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int m = 3, n = 3;
    print(m, n, arr);
    return 0;
}
```

Output on a C99 compatible compiler:

```
1 2 3 4 5 6 7 8 9
```

If compiler is not C99 compatible, then we can use one of the following methods to pass a variable sized 2D array.

4) Using a single pointer
In this method, we must typecast the 2D array when passing to function.

```
#include <stdio.h>
void print(int *arr, int m, int n)
{
    int i, j;
    for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
        printf("%d ", *((arr+i*n) + j));
}

int main()
{
    int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int m = 3, n = 3;

    // We can also use "print(&arr[0][0], m, n);"
    print((int *)arr, m, n);
    return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9
```

References:
http://www.eskimo.com/~scs/cclass/int/sx9a.html

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

How to write long strings in Multi-lines C/C++?

Image a situation where we want to use or print a long long string in C or C++, how to do do this?

In C/C++, we can break a string at any point in the middle using two double quotes in the middle. Below is a simple example to demonstrate the same.

```
#include<stdio.h>
int main()
{
  // We can put two double quotes anywhere in a string
  char *str1  = "geeks""quiz";

  // We can put space line break between two double quotes
  char *str2  = "Qeeks"     "Quiz";
  char *str3  = "Qeeks"
               "Quiz";

  puts(str1);
  puts(str2);
  puts(str3);

  puts("Geeks"       // Breaking string in multiple lines
      "forGeeks");
  return 0;
}
```

Output:
geeksquiz
GeeksQuiz
GeeksQuiz
GeeksforGeeks

Below are few examples with long long strings broken using two double quotes for better readability.

```
#include<stdio.h>
int main()
{
  char *str = "These are reserved words in C language are int, float, "
            "if, else, for, while etc. An Identifier is a sequence of"
            "letters and digits, but must start with a letter. "
            "Underscore ( _ ) is treated as a letter. Identifiers are "
            "case sensitive. Identifiers are used to name variables,"
            "functions etc.";
  puts(str);
  return 0;
}
```

Output: *These are reserved words in C language are int, float, if, else, for, while etc. An Identifier is a sequence ofletters and digits, but must start with a letter. Underscore (_) is treated as a letter. Identifiers are case sensitive. Identifiers are used to name variables,functions etc.*

Similarly, we can write long strings in printf and or cout.

```
#include<stdio.h>
int main()
{
  char *str = "An Identifier is a sequence of"
            "letters and digits, but must start with a letter. "
            "Underscore ( _ ) is treated as a letter. Identifiers are "
            "case sensitive. Identifiers are used to name variables,"
            "functions etc.";
```

```
    printf ("These are reserved words in C language are int, float, "
            "if, else, for, while etc. %s ", str);
    return 0;
}
```

Output: *These are reserved words in C language are int, float, if, else, for, while etc. An Identifier is a sequence ofletters and digits, but must start with a letter. Underscore (_) is treated as a letter. Identifiers are case sensitive. Identifiers are used to name variables,functions etc.*

This article is contributed by **Ayush Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C

What are the data types for which it is not possible to create an array?

In C, it is possible to have array of all types except following.

1) void.

2) functions.

For example, below program throws compiler error

```
int main()
{
   void arr[100];
}
```

Output:

```
error: declaration of 'arr' as array of voids
```

But we can have array of void pointers and function pointers. The below program works fine.

```
int main()
{
   void *arr[100];
}
```

See examples of function pointers for details of array function pointers.

This article is contributed by **Shiva**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C

Variable Length Arrays in C and C++

Variable length arrays is a feature where we can allocate an auto array (on stack) of variable size. C supports variable sized arrays from C99 standard. For example, the below program compiles and runs fine in C.

```
void fun(int n)
{
 int arr[n];
 // ......
}
int main()
{
  fun(6);
}
```

But C++ standard (till C++11) doesn't support variable sized arrays. The C++11 standard mentions array size as a constant-expression See (See 8.3.4 on page 179 of N3337). So the above program may not be a valid C++ program. The program may work in GCC compiler, because GCC compiler provides an extension to support them.

As a side note, the latest C++14 (See 8.3.4 on page 184 of N3690) mentions array size as a simple expression (not constant-expression).

References:
http://stackoverflow.com/questions/1887097/variable-length-arrays-in-c
https://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

C/C++ Puzzles
cpp-array

What should be data type of case labels of switch statement in C?

In C switch statement, the expression of each case label must be an integer constant expression.

For example, the following program fails in compilation.

```
/* Using non-const in case label */
#include<stdio.h>
int main()
{
 int i = 10;
 int c = 10;
 switch(c)
 {
   case i: // not a "const int" expression
        printf("Value of c = %d", c);
        break;
   /*Some more cases */

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

Putting *const* before *i* makes the above program work.

```
#include<stdio.h>
int main()
{
 const int i = 10;
 int c = 10;
 switch(c)
 {
   case i:  // Works fine
        printf("Value of c = %d", c);
```

```
      break;
    /*Some more cases */

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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

For Versus While

Question: Is there any example for which the following two loops will not work same way?

```
/*Program 1 --> For loop*/
for (<init-stmnt>; <boolean-expr>; <incr-stmnt>)
{
   <body-statements>
}

/*Program 2 --> While loop*/
<init-stmnt>;
while (<boolean-expr>)
{
   <body-statements>
   <incr-stmnt>
}
```

Solution:

If the body-statements contains continue, then the two programs will work in different ways

See the below examples: Program 1 will print “loop” 3 times but Program 2 will go in an infinite loop.

Example for program 1

```
int main()
{
  int i = 0;
  for(i = 0; i < 3; i++)
  {
    printf("loop ");
    continue;
  }
  getchar();
  return 0;
}
```

Example for program 2

```
int main()
{
  int i = 0;
  while(i < 3)
  {
    printf("loop"); /* printed infinite times */
    continue;
    i++; /*This statement is never executed*/
  }
  getchar();
  return 0;
}
```

Please write comments if you want to add more solutions for the above question.

C/C++ Puzzles
c puzzle
puzzle

A nested loop puzzle

Which of the following two code segments is faster? Assume that compiler makes no optimizations.

```
/* FIRST */
for(i=0;i<10;i++)
  for(j=0;j<100;j++)
    //do somthing
```

```
/* SECOND */
for(i=0;i<100;i++)
  for(j=0;j<10;j++)
    //do something
```

Both code segments provide same functionality, and the code inside the two for loops would be executed same number of times in both code segments.
If we take a closer look then we can see that the SECOND does more operations than the FIRST. It executes all three parts (assignment, comparison and increment) of the for loop more times than the corresponding parts of FIRST

a) The SECOND executes assignment operations (j = 0 or i = 0) 101 times while FIRST executes only 11 times.
b) The SECOND does 101 + 1100 comparisons (i Dheeraj for suggesting the solution.

Please write comments if you find any of the answers/codes incorrect, or you want to share more information about the topics discussed above.

C/C++ Puzzles

Interesting facts about switch statement in C

Switch is a control statement that allows a value to change control of execution.

```
// Following is a simple program to demonstrate syntax of switch.
#include <stdio.h>
int main()
{
   int x = 2;
   switch (x)
   {
      case 1: printf("Choice is 1");
              break;
      case 2: printf("Choice is 2");
               break;
      case 3: printf("Choice is 3");
              break;
      default: printf("Choice other than 1, 2 and 3");
               break;
   }
   return 0;
}
```

Output:

```
Choice is 2
```

Following are some interesting facts about switch statement.

1) The expression used in switch must be integral type (int, char and enum). Any other type of expression is not allowed.

```
// float is not allowed in switch
#include <stdio.h>
int main()
{
  float x = 1.1;
  switch (x)
  {
     case 1.1: printf("Choice is 1");
              break;
     default: printf("Choice other than 1, 2 and 3");
             break;
  }
  return 0;
}
```

Output:

```
 Compiler Error: switch quantity not an integer
```

In Java, String is also allowed in switch (See this)

2) All the statements following a matching case execute until a break statement is reached.

```
// There is no break in all cases
#include <stdio.h>
int main()
{
  int x = 2;
  switch (x)
  {
     case 1: printf("Choice is 1\n");
     case 2: printf("Choice is 2\n");
     case 3: printf("Choice is 3\n");
     default: printf("Choice other than 1, 2 and 3\n");
  }
  return 0;
}
```

Output:

```
Choice is 2
Choice is 3
Choice other than 1, 2 and 3
```

```
// There is no break in some cases
#include <stdio.h>
int main()
{
  int x = 2;
  switch (x)
  {
     case 1: printf("Choice is 1\n");
     case 2: printf("Choice is 2\n");
     case 3: printf("Choice is 3\n");
     case 4: printf("Choice is 4\n");
            break;
     default: printf("Choice other than 1, 2, 3 and 4\n");
             break;
  }
  printf("After Switch");
  return 0;
}
```

Output:

```
Choice is 2
Choice is 3
Choice is 4
After Switch
```

3) The default block can be placed anywhere. The position of default doesn't matter, it is still executed if no match found.

```
// The default block is placed above other cases.
#include <stdio.h>
int main()
{
  int x = 4;
  switch (x)
  {
     default: printf("Choice other than 1 and 2");
            break;
     case 1: printf("Choice is 1");
           break;
     case 2: printf("Choice is 2");
            break;
  }
  return 0;
}
```

Output:

```
Choice other than 1 and 2
```

4) The integral expressions used in labels must be a constant expressions

```
// A program with variable expressions in labels
#include <stdio.h>
int main()
{
   int x = 2;
   int arr[] = {1, 2, 3};
   switch (x)
   {
      case arr[0]: printf("Choice 1\n");
      case arr[1]: printf("Choice 2\n");
      case arr[2]: printf("Choice 3\n");
   }
   return 0;
}
```

Output:

```
Compiler Error: case label does not reduce to an integer constant
```

5) The statements written above cases are never executed After the switch statement, the control transfers to the matching case, the statements executed before case are not executed.

```
// Statements before all cases are never executed
#include <stdio.h>
int main()
{
  int x = 1;
  switch (x)
  {
     x = x + 1;  // This statement is not executed
     case 1: printf("Choice is 1");
           break;
     case 2: printf("Choice is 2");
            break;
     default: printf("Choice other than 1 and 2");
            break;
  }
  return 0;
}
```

Output:

```
Choice is 1
```

6) Two case labels cannot have same value

```
// Program where two case labels have same value
#include <stdio.h>
int main()
{
  int x = 1;
  switch (x)
  {
     case 2: printf("Choice is 1");
           break;
     case 1+1: printf("Choice is 2");
            break;
  }
  return 0;
}
```

Output:

```
Compiler Error: duplicate case value
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-switch

Functions in C

A function is a set of statements that take inputs, do some specific computation and produces output.

The idea is to put some commonly or repeatedly done task together and make a function, so that instead of writing the same code again and again for different inputs, we can call the function.

Example:
Below is a simple C program to demonstrate functions in C.

```
#include <stdio.h>

// An example function that takes two parameters 'x' and 'y'
// as input and returns max of two input numbers
int max(int x, int y)
{
   if (x > y)
     return x;
   else
     return y;
}

// main function that doesn't receive any parameter and
// returns integer.
int main(void)
{
   int a = 10, b = 20;

   // Calling above function to find max of 'a' and 'b'
   int m = max(a, b);

   printf("m is %d", m);
   return 0;
}
```

Output:

```
m is 20
```

Function Declaration

Function declaration tells compiler about number of parameters function takes, data-types of parameters and return type of function. Putting parameter names in function declaration is optional in function declaration, but it is necessary to put them in definition. Below are example of function declarations. (parameter names are not there in below declarations)

```
// A function that takes two integers as parameters
// and returns an integer
int max(int, int);

// A function that takes a char and an int as parameters
// and returns an integer
int fun(char, int);
```

It is always recommended to declare a function before it is used (See this, this and this for details)

In C, we can do both declaration and definition at same place, like done in above example program.

C also allows to declare and define functions separately, this is specially needed in case of library functions. The library functions are declared in header files and defined in library files. Below is an example declaration.

Parameter Passing to functions

The parameters passed to function are called ***actual parameters***. For example, in the above program 10 and 20 are actual parameters.
The parameters received by function are called ***formal parameters***. For example, in the above program x and y are formal parameters.
There are two most popular ways to pass parameters.

Pass by Value: In this parameter passing method, values of actual parameters are copied to function's formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of caller.

Pass by Reference Both actual and formal parameters refer to same locations, so any changes made inside the function are actually reflected in actual parameters of caller.

In C, parameters are always passed by value. Parameters are always passed by value in C. For example. in the below code, value of y is not modified using the function fun().

```
#include <stdio.h>
void fun(int x)
{
  x = 30;
}

int main(void)
{
   int x = 20;
   fun(x);
   printf("x = %d", x);
   return 0;
}
```

Output:

```
x = 20
```

However, in C, we can use pointers to get the effect of pass by reference. For example, consider the below program. The function fun() expects a pointer ptr to an integer (or an address of an integer). It modifies the value at the address ptr. The dereference operator * is used to access the value at an address. In the statement '*ptr = 30', value at address ptr is changed to 30. The address operator & is used to get the address of a variable of any data type. In the function call statement 'fun(&y)', address of y is passed so that y can be modified using its address.

```
# include <stdio.h>
void fun(int *ptr)
{
   *ptr = 30;
}

int main()
{
  int x = 20;
  fun(&x);
  printf("x = %d", x);

  return 0;
}
```

Output:

```
x = 30
```

Following are some important points about functions in C.
1) Every C program has a function called main() that is called by operating system when a user runs the program.

2) Every function has a return type. If a function doesn't return any value, then void is used as return type.

3) In C, functions can return any type except arrays and functions. We can get around this limitation by returning pointer to array or pointer to function.

4) Empty parameter list in C mean that the parameter list is not specified and function can be called with any parameters. In C, it is not a good idea to declare a function like fun(). To declare a function that can only be called without any parameter, we should use "void fun(void)".
As a side note, in C++, empty list means function can only be called without any parameter. In C++, both void fun() and void fun(void) are same.

More on Functions in C:

- Quiz on function in C
- Importance of function prototype in C
- Functions that are executed before and after main() in C
- return statement vs exit() in main()
- How to Count Variable Numbers of Arguments in C?,
- What is evaluation order of function parameters in C?
- Does C support function overloading?
- How can we return multiple values from a function?
- What is the purpose of a function prototype?
- Static functions in C
- exit(), abort() and assert()
- Implicit return type int in C
- What happens when a function is called before its declaration in C?

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C

Importance of function prototype in C

Function prototype tells compiler about number of parameters function takes, data-types of parameters and return type of function. By using this information, compiler cross checks function parameters and their data-type with function definition and function call. If we ignore function prototype, program may compile with warning, and may work properly. But some times, it will give strange output and it is very hard to find such programming mistakes. Let us see with examples

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

int main(int argc, char *argv[])
{
 FILE *fp;

 fp = fopen(argv[1], "r");
 if (fp == NULL) {
  fprintf(stderr, "%s\n", strerror(errno));
  return errno;
 }

 printf("file exist\n");

 fclose(fp);

 return 0;
}
```

Above program checks existence of file, provided from command line, if given file is exist, then the program prints “file exist”, otherwise it prints appropriate error message. Let us provide a filename, which does not exist in file system, and check the output of program on x86_64 architecture.

```
[narendra@/media/partition/GFG]$ ./file_existence hello.c
Segmentation fault (core dumped)
```

Why this program crashed, instead it should show appropriate error message. This program will work fine on x86 architecture, but will crash on x86_64 architecture. Let us see what was wrong with code. Carefully go through the program, deliberately I haven’t included prototype of “strerror()” function. This function returns “pointer to character”, which will print error message which depends on errno passed to this function. Note that x86 architecture is ILP-32 model, means integer, pointers and long are 32-bit wide, that’s why program will work correctly on this architecture. But x86_64 is LP-64 model, means long and pointers are 64 bit wide. *In C language, when we don’t provide prototype of function, the compiler assumes that function returns an integer.* In our example, we haven’t included “string.h” header file (strerror’s prototype is declared in this file), that’s why compiler assumed that function returns integer. But its return type is pointer to character. In x86_64, pointers are 64-bit wide and integers are 32-bits wide, that’s why while returning from function, the returned address gets truncated (i.e. 32-bit wide address, which is size of integer on x86_64) which is invalid and when we try to dereference this address, the result is segmentation fault.

Now include the “string.h” header file and check the output, the program will work correctly.

```
[narendra@/media/partition/GFG]$ ./file_existence hello.c
No such file or directory
```

Consider one more example.

```
#include <stdio.h>

int main(void)
{
 int *p = malloc(sizeof(int));

 if (p == NULL) {
  perror("malloc()");
  return -1;
 }

 *p = 10;
 free(p);

 return 0;
}
```

Above code will work fine on IA-32 model, but will fail on IA-64 model. Reason for failure of this code is we haven't included prototype of malloc() function and returned value is truncated in IA-64 model.

This article is compiled by **Narendra Kangralkar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

Functions that are executed before and after main() in C

With GCC family of C compilers, we can mark some functions to execute before and after main(). So some startup code can be executed before main() starts, and some cleanup code can be executed after main() ends. For example, in the following program, myStartupFun() is called before main() and myCleanupFun() is called after main().

```
#include<stdio.h>

/* Apply the constructor attribute to myStartupFun() so that it
    is executed before main() */
void myStartupFun (void) __attribute__ ((constructor));


/* Apply the destructor attribute to myCleanupFun() so that it
   is executed after main() */
void myCleanupFun (void) __attribute__ ((destructor));


/* implementation of myStartupFun */
void myStartupFun (void)
{
    printf ("startup code before main()\n");
}

/* implementation of myCleanupFun */
void myCleanupFun (void)
{
    printf ("cleanup code after main()\n");
}

int main (void)
{
    printf ("hello\n");
    return 0;
}
```

Output:

```
startup code before main()
hello
cleanup code after main()
```

Like the above feature, GCC has added many other interesting features to standard C language. See this for more details.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

return statement vs exit() in main()

In C++, what is the difference between *exit(0)* and *return 0* ?

When *exit(0)* is used to exit from program, destructors for locally scoped non-static objects are not called. But destructors are called if return 0 is used.

Program 1 – – uses exit(0) to exit

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

using namespace std;

class Test {
public:
  Test() {
    printf("Inside Test's Constructor\n");
  }

  ~Test(){
    printf("Inside Test's Destructor");
    getchar();
  }
};

int main() {
  Test t1;

  // using exit(0) to exit from main
  exit(0);
}
```

Output:

Inside Test’s Constructor

Program 2 – uses return 0 to exit

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

using namespace std;

class Test {
public:
  Test() {
    printf("Inside Test's Constructor\n");
  }

  ~Test(){
    printf("Inside Test's Destructor");
  }
};

int main() {
  Test t1;

   // using return 0 to exit from main
  return 0;
}
```

Output:

Inside Test’s Constructor

Inside Test’s Destructor

Calling destructors is sometimes important, for example, if destructor has code to release resources like closing files.

Note that static objects will be cleaned up even if we call exit(). For example, see following program.

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

```
using namespace std;

class Test {
public:
  Test() {
    printf("Inside Test's Constructor\n");
  }

  ~Test(){
    printf("Inside Test's Destructor");
    getchar();
  }
};

int main() {
  static Test t1;  // Note that t1 is static

  exit(0);
}
```

Output:
Inside Test's Constructor
Inside Test's Destructor

Contributed by **indiarox**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
exit
main
return

How to Count Variable Numbers of Arguments in C?

C supports variable numbers of arguments. But there is no language provided way for finding out total number of arguments passed. User has to handle this in one of the following ways:
1) By passing first argument as count of arguments.
2) By passing last argument as NULL (or 0).
3) Using some printf (or scanf) like mechanism where first argument has placeholders for rest of the arguments.

Following is an example that uses first argument *arg_count* to hold count of other arguments.

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

// this function returns minimum of integer numbers passed.  First
// argument is count of numbers.
int min(int arg_count, ...)
{
  int i;
  int min, a;

  // va_list is a type to hold information about variable arguments
  va_list ap;

  // va_start must be called before accessing variable argument list
  va_start(ap, arg_count);

  // Now arguments can be accessed one by one using va_arg macro
  // Initialize min as first argument in list
  min = va_arg(ap, int);

  // traverse rest of the arguments to find out minimum
  for(i = 2; i <= arg_count; i++) {
    if((a = va_arg(ap, int)) < min)
      min = a;
  }
  //va_end should be executed before the function returns whenever
  // va_start has been previously used in that function
  va_end(ap);

  return min;
}

int main()
{
   int count = 5;

   // Find minimum of 5 numbers: (12, 67, 6, 7, 100)
   printf("Minimum value is %d", min(count, 12, 67, 6, 7, 100));
   getchar();
   return 0;
}
```

Output:
Minimum value is 6

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

What is evaluation order of function parameters in C?

It is compiler dependent in C. It is never safe to depend on the order of evaluation of side effects. For example, a function call like below may very well behave differently from one compiler to another:

```
void func (int, int);

int i = 2;
func (i++, i++);
```

There is no guarantee (in either the C or the C++ standard language definitions) that the increments will be evaluated in any particular order. Either increment might happen first. func might get the arguments `2, 3′, or it might get `3, 2′, or even `2, 2′.

Source: http://gcc.gnu.org/onlinedocs/gcc/Non_002dbugs.html

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
GFacts

Does C support function overloading?

First of all, what is function overloading? Function overloading is a feature of a programming language that allows one to have many functions with same name but with different signatures.
This feature is present in most of the Object Oriented Languages such as C++ and Java. But C (not Object Oriented Language) doesn't support this feature. However, one can achieve the similar functionality in C indirectly. One of the approach is as follows.

Have a void * type of pointer as an argument to the function. And another argument telling the actual data type of the first argument that is being passed.

```
int foo(void * arg1, int arg2);
```

Suppose, arg2 can be interpreted as follows. 0 = Struct1 type variable, 1 = Struct2 type variable etc. Here Struct1 and Struct2 are user defined struct types.

While calling the function foo at different places…

```
foo(arg1, 0);   /*Here, arg1 is pointer to struct type Struct1 variable*/
foo(arg1, 1);    /*Here, arg1 is pointer to struct type Struct2 variable*/
```

Since the second argument of the foo keeps track the data type of the first type, inside the function foo, one can get the actual data type of the first argument by typecast accordingly. i.e. inside the foo function

```
if(arg2 == 0)
{
  struct1PtrVar = (Struct1 *)arg1;
}
else if(arg2 == 1)
{
  struct2PtrVar = (Struct2 *)arg1;
}
else
{
  /*Error Handling*/
}
```

There can be several other ways of implementing function overloading in C. But all of them will have to use pointers – the most powerful feature of C.
In fact, it is said that without using the pointers, one can't use C efficiently & effectively in a real world program!

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
c puzzle
puzzle

How can I return multiple values from a function?

We all know that a function in C can return only one value. So how do we achieve the purpose of returning multiple values.
Well, first take a look at the declaration of a function.

```
int foo(int arg1, int arg2);
```

So we can notice here that our interface to the function is through arguments and return value only. (Unless we talk about modifying the globals inside the function)

Let us take a deeper look…Even though a function can return only one value but that value can be of pointer type. That's correct, now you're speculating right!
We can declare the function such that, it returns a structure type user defined variable or a pointer to it . And by the property of a structure, we know that a structure in C can hold multiple values of asymmetrical types (i.e. one int variable, four char variables, two float variables and so on…)

If we want the function to return multiple values of same data types, we could return the pointer to array of that data types.

We can also make the function return multiple values by using the arguments of the function. How? By providing the pointers as arguments.

Usually, when a function needs to return several values, we use one pointer in return instead of several pointers as argumentss.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
c puzzle
puzzle

What is the purpose of a function prototype?

The Function prototype serves the following purposes –

1) It tells the return type of the data that the function will return.
2) It tells the number of arguments passed to the function.

3) It tells the data types of the each of the passed arguments.
4) Also it tells the order in which the arguments are passed to the function.

Therefore essentially, function prototype specifies the input/output interlace to the function i.e. what to give to the function and what to expect from the function.

Prototype of a function is also called signature of the function.

What if one doesn't specify the function prototype?
Output of below kind of programs is generally asked at many places.

```
int main()
{
   foo();
   getchar();
   return 0;
}
void foo()
{
   printf("foo called");
}
```

If one doesn't specify the function prototype, the behavior is specific to C standard (either C90 or C99) that the compilers implement. Up to C90 standard, C compilers assumed the return type of the omitted function prototype as int. And this assumption at compiler side may lead to unspecified program behavior.

Later C99 standard specified that compilers can no longer assume return type as int. Therefore, C99 became more restrict in type checking of function prototype. But to make C99 standard backward compatible, in practice, compilers throw the warning saying that the return type is assumed as int. But they go ahead with compilation. Thus, it becomes the responsibility of programmers to make sure that the assumed function prototype and the actual function type matches.

To avoid all this implementation specifics of C standards, it is best to have function prototype.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
c puzzle
puzzle

Static functions in C

In C, functions are global by default. The "*static*" keyword before a function name makes it static. For example, below function *fun()* is static.

```
static int fun(void)
{
  printf("I am a static function ");
}
```

Unlike global functions in C, access to static functions is restricted to the file where they are declared. Therefore, when we want to restrict access to functions, we make them static. Another reason for making functions static can be reuse of the same function name in other files.

For example, if we store following program in one file *file1.c*

```
/* Inside file1.c */
static void fun1(void)
{
  puts("fun1 called");
}
```

And store following program in another file *file2.c*

```
/* Iinside file2.c  */
int main(void)
{
  fun1();
  getchar();
  return 0;
}
```

Now, if we compile the above code with command “*gcc file2.c file1.c*”, we get the error *“undefined reference to `fun1’”* . This is because *fun1()* is declared *static* in *file1.c* and cannot be used in *file2.c*.

Please write comments if you find anything incorrect in the above article, or want to share more information about static functions in C.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

exit(), abort() and assert()

exit()

```
void exit ( int status );
```

exit() terminates the process normally.
status: Status value returned to the parent process. Generally, a status value of 0 or EXIT_SUCCESS indicates success, and any other value or the constant EXIT_FAILURE is used to indicate an error. exit() performs following operations.
* Flushes unwritten buffered data.
* Closes all open files.
* Removes temporary files.
* Returns an integer exit status to the operating system.

The C standard atexit() function can be used to customize exit() to perform additional actions at program termination.

Example use of exit.

```
/* exit example */
#include <stdio.h>
#include <stdlib.h>

int main ()
{
  FILE * pFile;
  pFile = fopen ("myfile.txt", "r");
  if (pFile == NULL)
  {
    printf ("Error opening file");
    exit (1);
  }
  else
  {
    /* file operations here */
  }
  return 0;
}
```

abort()

```
 void abort ( void );
```

Unlike exit() function, abort() may not close files that are open. It may also not delete temporary files and may not flush stream buffer. Also, it does not call functions registered with atexit().

This function actually terminates the process by raising a SIGABRT signal, and your program can include a handler to intercept this signal (see this).

So programs like below might not write “Geeks for Geeks” to “tempfile.txt”

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

```
int main()
{
  FILE *fp = fopen("C:\\myfile.txt", "w");

  if(fp == NULL)
  {
    printf("\n could not open file ");
    getchar();
    exit(1);
  }

  fprintf(fp, "%s", "Geeks for Geeks");

  /* ....... */
  /* ....... */
  /* Something went wrong so terminate here */
  abort();

  getchar();
  return 0;
}
```

If we want to make sure that data is written to files and/or buffers are flushed then we should either use exit() or include a signal handler for SIGABRT.

assert()

```
void assert( int expression );
```

If expression evaluates to 0 (false), then the expression, sourcecode filename, and line number are sent to the standard error, and then abort() function is called. If the identifier NDEBUG (“no debug”) is defined with #define NDEBUG then the macro assert does nothing.

Common error outputting is in the form:

Assertion failed: expression, file filename, line line-number

```

#include<assert.h>

void open_record(char *record_name)
{
    assert(record_name != NULL);
    /* Rest of code */
}

int main(void)
{
   open_record(NULL);
}
```

This article is contributed by **Rahul Gupta**. Please write comments if you find anything incorrect in the above article or you want to share more information about the topic discussed above.
References:
http://www.cplusplus.com/reference/clibrary/cstdlib/abort/
http://www.cplusplus.com/reference/clibrary/cassert/assert/
http://www.acm.uiuc.edu/webmonkeys/book/c_guide/2.1.html
https://www.securecoding.cert.org/confluence/display/seccode/ERR06-C.+Understand+the+termination+behavior+of+assert%28%29+and+abort%28%29
https://www.securecoding.cert.org/confluence/display/seccode/ERR04-C.+Choose+an+appropriate+termination+strategy

Implicit return type int in C

Predict the output of following C program.

```
#include <stdio.h>
fun(int x)
{
   return x*x;
}
int main(void)
{
   printf("%d", fun(10));
   return 0;
}
```

Output: 100

The important thing to note is, there is no return type for fun(), the program still compiles and runs fine in most of the C compilers. In C, if we do not specify a return type, compiler assumes an implicit return type as int. However, C99 standard doesn't allow return type to be omitted even if return type is int. This was allowed in older C standard C89.

In C++, the above program is not valid except few old C++ compilers like Turbo C++. Every function should specify the return type in C++.

This article is contributed by **Pravasi Meet**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-functions

What happens when a function is called before its declaration in C?

In C, if a function is called before its declaration, the **compiler assumes return type of the function as int**.

For example, the following program fails in compilation.

```
#include <stdio.h>
int main(void)
{
   // Note that fun() is not declared
   printf("%d\n", fun());
   return 0;
}

char fun()
{
  return 'G';
}
```

The following program compiles and run fine because return type of fun() is changed to int.

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

int fun()
{
  return 10;
}
```

What about parameters? compiler assumes nothing about parameters. Therefore, the compiler will not be able to perform compile-time checking of argument types and arity when the function is applied to some arguments. This can cause problems. For example, the following program compiled fine in GCC and produced garbage value as output.

```
#include <stdio.h>

int main (void)
{
   printf("%d", sum(10, 5));
   return 0;
}
int sum (int b, int c, int a)
{
   return (a+b+c);
}
```

There is this misconception that the compiler assumes input parameters also int. Had compiler assumed input parameters int, the above program would have failed in compilation.

It is always recommended to declare a function before its use so that we don't see any surprises when the program is run (See this for more details).

Source:
http://en.wikipedia.org/wiki/Function_prototype#Uses

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-puzzle

Why C treats array parameters as pointers?

In C, array parameters are treated as pointers. The following two definitions of foo() look different, but to the compiler they mean exactly the same thing. It's preferable to use whichever syntax is more accurate for readability. If the pointer coming in really is the base address of a whole array, then we should use [ ].

```
void foo(int arr_param[])
{

  /* Silly but valid. Just changes the local pointer */
  arr_param = NULL;
}

void foo(int *arr_param)
{

  /* ditto */
  arr_param = NULL;
}
```

Array parameters treated as pointers because of efficiency. It is inefficient to copy the array data in terms of both memory and time; and most of the times, when we pass an array our intention is to just tell the array we interested in, not to create a copy of the array.

Asked by Shobhit

References:
http://cslibrary.stanford.edu/101/EssentialC.pdf

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
pointer

Output of the program | Dereference, Reference, Dereference,

Reference….

Predict the output of below program

```
int main()
{
 char *ptr = "geeksforgeeks";
 printf("%c\n", *&*&*ptr);

 getchar();
 return 0;
}
```

Output: g

Explanation: The operator * is used for dereferencing and the operator & is used to get the address. These operators cancel effect of each other when used one after another. We can apply them alternatively any no. of times. For example *ptr gives us g, &*ptr gives address of g, *&*ptr again g, &*&*ptr address of g, and finally *&*&*ptr gives 'g'

Now try below

```
int main()
{
 char *ptr = "geeksforgeeks";
 printf("%s\n", *&*&ptr);

 getchar();
 return 0;
}
```

GATE CS Corner Company Wise Coding Practice

Output
C-Output

Dangling, Void , Null and Wild Pointers

Dangling pointer

A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer. There are **three** different ways where Pointer acts as dangling pointer

1. **De-allocation of memory**

```
// Deallocating a memory pointed by ptr causes
// dangling pointer
#include <stdlib.h>
#include <stdio.h>
int main()
{
   int *ptr = (int *)malloc(sizeof(int));

   // After below free call, ptr becomes a
   // dangling pointer
   free(ptr);

   // No more a dangling pointer
   ptr = NULL;
}
```

2. **Function Call**

```
// The pointer pointing to local variable becomes
// dangling when local variable is static.
#include<stdio.h>

int *fun()
```

```
{
    // x is local variable and goes out of
    // scope after an execution of fun() is
    // over.
    int x = 5;

    return &x;
}

// Driver Code
int main()
{
    int *p = fun();
    fflush(stdin);

    // p points to something which is not
    // valid anymore
    printf("%d", *p);
    return 0;
}
```

Output:

```
A garbage Address
```

The above problem doesn’t appear (or p doesn’t become dangling) if x is a static variable.

```
// The pointer pointing to local variable doesn't
// become dangling when local variable is static.
#include<stdio.h>

int *fun()
{
    // x now has scope throughout the program
    static int x = 5;

    return &x;
}

int main()
{
    int *p = fun();
    fflush(stdin);

    // Not a dangling pointer as it points
    // to static variable.
    printf("%d",*p);
}
```

Output:

```
 5
```

3. **Variable goes out of scope**

```
void main()
{
   int *ptr;
   .....
   .....
   {
       int ch;
       ptr = &ch;
   }
   .....
   // Here ptr is dangling pointer
}
```

Void pointer

Void pointer is a specific pointer type – void * – a pointer that points to some data location in storage, which doesn't have any specific type. Void refers to the type. Basically the type of data that it points to is can be any. If we assign address of char data type to void pointer it will become char Pointer, if int data type then int pointer and so on. Any pointer type is convertible to a void pointer hence it can point to any value.

Important Points

1. void pointers **cannot be dereferenced**. It can however be done using typecasting the void pointer
2. Pointer arithmetic is not possible on pointers of void due to lack of concrete value and thus size.

Example:

```
#include<stdlib.h>

int main()
{
    int x = 4;
    float y = 5.5;

    //A void pointer
    void *ptr;
    ptr = &x;

    // (int*)ptr - does type casting of void
    // *((int*)ptr) dereferences the typecasted
    // void pointer variable.
    printf("Integer variable is = %d", *( (int*) ptr) );

    // void pointer is now float
    ptr = &y;
    printf("\nFloat variable is= %f", *( (float*) ptr) );

    return 0;
}
```

Output:

```
Integer variable is = 4
Float variable is= 5.500000
```

Refer void pointer article for details.

NULL Pointer

NULL Pointer is a pointer which is pointing to nothing. In case, if we don't have address to be assigned to a pointer, then we can simply use NULL.

```
#include <stdio.h>
int main()
{
    // Null Pointer
    int *ptr = NULL;

    printf("The value of ptr is %u", ptr);
    return 0;
}
```

Output :

```
The value of ptr is 0
```

Important Points

1. **NULL vs Uninitialized pointer –** An uninitialized pointer stores an undefined value. A null pointer stores a defined value, but one that is defined by the environment to not be a valid address for any member or object.
2. **NULL vs Void Pointer** – Null pointer is a value, while void pointer is a type

Wild pointer

A pointer which has not been initialized to anything (not even NULL) is known as wild pointer. The pointer may be initialized to a non-NULL garbage value that may not be a valid address.

```
int main()
{
   int *p;  /* wild pointer */

   int x = 10;

   // p is not a wild pointer now
   p = &x;

   return 0;
}
```

This article is contributed by **Spoorthi Arun**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-pointer

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.

Please write comments if you find anything incorrect in the above article.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

How to declare a pointer to a function?

Well, we assume that you know what does it mean by pointer in C. So how do we create a pointer to an integer in C?
Huh..it is pretty simple..

```
int * ptrInteger; /*We have put a * operator between int
                    and ptrInteger to create a pointer.*/
```

Here ptrInteger is a pointer to integer. If you understand this, then logically we should not have any problem in declaring a pointer to a function

So let us first see ..how do we declare a function? For example,

```
int foo(int);
```

Here foo is a function that returns int and takes one argument of int type. So as a logical guy will think, by putting a * operator between int and foo(int) should create a pointer to a function i.e.

```
int * foo(int);
```

But Oops..C operator precedence also plays role here ..so in this case, operator () will take priority over operator *. And the above declaration will mean – a function foo with one argument of int type and return value of int * i.e. integer pointer. So it did something that we didn't want to do.

So as a next logical step, we have to bind operator * with foo somehow. And for this, we would change the default precedence of C operators using () operator.

```
int (*foo)(int);
```

That's it. Here * operator is with foo which is a function name. And it did the same that we wanted to do.

So that wasn't as difficult as we thought earlier!

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
pointer

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*/
```

References: http://icecube.wisc.edu/~dglo/c_class/array_ptr.html

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
pointer

---

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'
```

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);
}
```

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;
}
```

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;
}
```

Output:

```
10
```

2) The C standard doesn't allow pointer arithmetic with void pointers. However, in GNU C it is allowed by considering the size of void is 1. For example the following program compiles and runs fine in gcc.

```
#include<stdio.h>
int main()
{
    int a[2] = {1, 2};
    void *ptr = &a;
    ptr = ptr + sizeof(int);
    printf("%d", *(int *)ptr);
    return 0;
}
```

Output:

```
2
```

Note that the above program may not work in other compilers.

References:
http://stackoverflow.com/questions/20967868/should-the-compiler-warn-on-pointer-arithmetic-with-a-void-pointer
http://stackoverflow.com/questions/692564/concept-of-void-pointer-in-c-programming

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C

NULL pointer in C

At the very high level, we can think of NULL as null pointer which is used in C for various purposes. Some of the most common use cases for NULL are
a) To initialize a pointer variable when that pointer variable isn't assigned any valid memory address yet.
b) To check for null pointer before accessing any pointer variable. By doing so, we can perform error handling in pointer related code e.g. dereference pointer variable only if it's not NULL.
c) To pass a null pointer to a function argument when we don't want to pass any valid memory address.

The example of a) is

```
int * pInt = NULL;
```

The example of b) is

```
if(pInt != NULL) /*We could use if(pInt) as well*/
{ /*Some code*/}
else
{ /*Some code*/}
```

The example of c) is

```
int fun(int *ptr)
{
 /*Fun specific stuff is done with ptr here*/
 return 10;
}
fun(NULL);
```

It should be noted that NULL pointer is different from uninitialized and dangling pointer. In a specific program context, all uninitialized or dangling or NULL pointers are invalid but NULL is a specific invalid pointer which is mentioned in C standard and has specific purposes. What we mean is that uninitialized and dangling pointers are invalid but they can point to some memory address that may be accessible though the memory access is unintended.

```
#include <stdio.h>
int main()
{
 int *i, *j;
 int *ii = NULL, *jj = NULL;
 if(i == j)
 {
  printf("This might get printed if both i and j are same by chance.");
 }
 if(ii == jj)
 {
  printf("This is always printed coz ii and jj are same.");
 }
 return 0;
}
```

By specifically mentioning NULL pointer, C standard gives mechanism using which a C programmer can use and check whether a given pointer is legitimate or not. But what exactly is NULL and how it's defined? Strictly speaking, NULL expands to an implementation-defined null pointer constant which is defined in many header files such as "*stdio.h*", "*stddef.h*", "*stdlib.h*" etc. Let us see what C standards say about null pointer. From C11 standard clause 6.3.2.3,

"*An integer constant expression with the value 0, or such an expression cast to type void *, is called a null pointer constant. If a null pointer constant is converted to a pointer type, the resulting pointer, called a null pointer, is guaranteed to compare unequal to a pointer to any object or function.*"

Before we proceed further on this NULL discussion :), let's mention few lines about C standard just in case you wants to refer it for further study. Please note that ISO/IEC 9899:2011 is the C language's latest standard which was published in Dec 2011. This is also called C11 standard. For completeness, let us mention that previous standards for C were C99, C90 (also known as ISO C) and C89 (also known as ANSI C). Though the actual C11 standard can be purchased from ISO, there's a draft document which is available in public domain for free.

Coming to our discussion, NULL macro is defined as *((void *)0)* in header files of most of the C compiler implementations. But C standard is saying that 0 is also a null pointer constant. It means that the following is also perfectly legal as per standard.

```
int * ptr = 0;
```

Please note that 0 in the above C statement is used in pointer-context and it's different from 0 as integer. This is one of the reason why usage of NULL is preferred because it makes it explicit in code that programmer is using null pointer not integer 0. Another important concept about NULL is that "*NULL expands to an implementation-defined null pointer constant*". This statement is also from C11 clause 7.19. It means that internal representation of the null pointer could be non-zero bit pattern to convey NULL pointer. That's why NULL always needn't be internally represented as all zeros bit pattern. A compiler implementation can choose to represent "null pointer constant" as a bit pattern for all 1s or anything else. But again, as a C programmer, we needn't to worry much on the internal value of the null pointer unless we are involved in Compiler coding or even below level of coding. Having said so, typically NULL is represented as all bits set to 0 only. To know this on a specific platform, one can use the following

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

Most likely, it's printing 0 which is the typical internal null pointer value but again it can vary depending on the C compiler/platform. You can try few other things in above program such as *printf("'%c",NULL)* or *printf("%s",NULL)* and even *printf("%f",NULL)*. The outputs of these are going to be different depending on the platform used but it'd be interesting especially usage of *%f* with NULL!

Can we use *sizeof()* operator on NULL in C? Well, usage of *sizeof(NULL)* is allowed but the exact size would depend on platform.

```
#include<stdio.h>
int main()
{
 printf("%d",sizeof(NULL));
 return 0;
}
```

Since NULL is defined as *((void*)0)*, we can think of NULL as a special pointer and its size would be equal to any pointer. If pointer size of a platform is 4 bytes, the output of above program would be 4. But if pointer size on a platform is 8 bytes, the output of above program would be 8.

What about dereferencing of NULL? What's going to happen if we use the following C code

```
#include<stdio.h>
int main()
{
 int * ptr = NULL;
 printf("%d",*ptr);
 return 0;
}
```

On some machines, the above would compile successfully but crashes when the program is run though it needn't to show the same behavior across all the machines. Again it depends on lot of factors. But the idea of mentioning the above snippet is that we should always check for NULL before accessing it.

Since NULL is typically defined as *((void*)0)*, let us discuss a little bit about *void* type as well. As per C11 standard clause 6.2.5, "*The void type comprises an empty set of values; it is an incomplete object type that cannot be completed*". Even C11 clause 6.5.3.4 mentions that "*The sizeof operator shall not be applied to an expression that has function type or an incomplete type, to the parenthesized name of such a type, or to an expression that designates a bit-field member.*" Basically, it means that *void* is an incomplete type whose size doesn't make any sense in C programs but implementations (such as gcc) can choose *sizeof(void)* as 1 so that the flat memory pointed by void pointer can be viewed as untyped memory i.e. a sequence of bytes. But the output of the following needn't to same on all platforms.

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

On gcc, the above would output 1. What about *sizeof(void *)*? Here C11 has mentioned guidelines. From clause 6.2.5, "*A pointer to void shall have the same representation and alignment requirements as a pointer to a character type*". That's why the output of the following would be same as any pointer size on a machine.

```
#include<stdio.h>
int main()
{
 printf("%d",sizeof(void *));
 return 0;
}
```

Inspite of mentioning machine dependent stuff as above, we as C programmers should always strive to make our code as portable as possible. So we can conclude on NULL as follows:

1. Always initialize pointer variables as NULL.
2. Always perform NULL check before accessing any pointer.

Please do Like/Tweet/G+1 if you find the above useful. Also, please do leave us comment for further clarification or info. We would love to help and learn

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C

Function Pointer in C

In C, like normal data pointers (int *, char *, etc), we can have pointers to functions. Following is a simple example that shows declaration and function call using function pointer.

```
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
   printf("Value of a is %d\n", a);
}

int main()
{
   // fun_ptr is a pointer to function fun()
```

```
    void (*fun_ptr)(int) = &fun;

    /* The above line is equivalent of following two
       void (*fun_ptr)(int);
       fun_ptr = &fun;
    */

    // Invoking fun() using fun_ptr
    (*fun_ptr)(10);

    return 0;
}
```

Output:

```
Value of a is 10
```

Why do we need an extra bracket around function pointers like fun_ptr in above example?
If we remove bracket, then the expression “void (*fun_ptr)(int)” becomes “void *fun_ptr(int)” which is declaration of a function that returns void pointer. See following post for details.
How to declare a pointer to a function?

Following are some interesting facts about function pointers.

1) Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.

2) Unlike normal pointers, we do not allocate de-allocate memory using function pointers.

3) A function’s name can also be used to get functions’ address. For example, in the below program, we have removed address operator ‘&’ in assignment. We have also changed function call by removing *, the program still works.

```
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
    printf("Value of a is %d\n", a);
}

int main()
{
    void (*fun_ptr)(int) = fun;  // & removed

    fun_ptr(10);  // * removed

    return 0;
}
```

Output:

```
Value of a is 10
```

4) Like normal pointers, we can have an array of function pointers. Below example in point 5 shows syntax for array of pointers.

5) Function pointer can be used in place of switch case. For example, in below program, user is asked for a choice between 0 and 2 to do different tasks.

```
#include <stdio.h>
void add(int a, int b)
{
    printf("Addition is %d\n", a+b);
}
void subtract(int a, int b)
{
    printf("Subtraction is %d\n", a-b);
}
void multiply(int a, int b)
{
    printf("Multiplication is %d\n", a*b);
}

int main()
{
    // fun_ptr_arr is an array of function pointers
    void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
    unsigned int ch, a = 15, b = 10;

    printf("Enter Choice: 0 for add, 1 for subtract and 2 "
            "for multiply\n");
    scanf("%d", &ch);

    if (ch > 2) return 0;

    (*fun_ptr_arr[ch])(a, b);

    return 0;
}
```

```
Enter Choice: 0 for add, 1 for subtract and 2 for multiply
2
Multiplication is 150
```

6) Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.
For example, consider the following C program where wrapper() receives a void fun() as parameter and calls the passed function.

```
// A simple C program to show function pointers as parameter
#include <stdio.h>

// Two simple functions
void fun1() { printf("Fun1\n"); }
void fun2() { printf("Fun2\n"); }

// A function that receives a simple function
// as parameter and calls the function
void wrapper(void (*fun)())
{
    fun();
}

int main()
{
    wrapper(fun1);
    wrapper(fun2);
    return 0;
}
```

This point in particular is very useful in C. In C, we can use function pointers to avoid code redundancy. For example a simple qsort() function can be used to sort arrays in ascending order or descending or by any other order in case of array of structures. Not only this, with function pointers and void pointers, it is possible to use qsort for any data type.

```
// An example for qsort and comparator
#include <stdio.h>
#include <stdlib.h>

// A sample comparator function that is used
// for sorting an integer array in ascending order.
// To sort any array for any other data type and/or
// criteria, all we need to do is write more compare
// functions.  And we can use the same qsort()
int compare (const void * a, const void * b)
{
  return ( *(int*)a - *(int*)b );
}

int main ()
{
  int arr[] = {10, 5, 15, 12, 90, 80};
  int n = sizeof(arr)/sizeof(arr[0]), i;

  qsort (arr, n, sizeof(int), compare);

  for (i=0; i<n; i++)
     printf ("%d ", arr[i]);
  return 0;
}
```

Output:

```
5 10 12 15 80 90
```

Similar to qsort(), we can write our own functions that can be used for any data type and can do different tasks without code redundancy. Below is an example search function that can be used for any data type. In fact we can use this search function to find close elements (below a threshold) by writing a customized compare function.

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

// A compare function that is used for searching an integer
// array
bool compare (const void * a, const void * b)
{
  return ( *(int*)a == *(int*)b );
}

// General purpose search() function that can be used
// for searching an element *x in an array arr[] of
// arr_size. Note that void pointers are used so that
// the function can be called by passing a pointer of
// any type.  ele_size is size of an array element
int search(void *arr, int arr_size, int ele_size, void *x,
           bool compare (const void * , const void *))
{
    // Since char takes one byte, we can use char pointer
    // for any type/ To get pointer arithmetic correct,
    // we need to multiply index with size of an array
    // element ele_size
    char *ptr = (char *)arr;

    int i;
    for (i=0; i<arr_size; i++)
        if (compare(ptr + i*ele_size, x))
           return i;

    // If element not found
    return -1;
}

int main()
{
    int arr[] = {2, 5, 7, 90, 70};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 7;
    printf ("Returned index is %d ", search(arr, n,
                           sizeof(int), &x, compare));
    return 0;
}
```

Output:

```
Returned index is 2
```

The above search function can be used for any data type by writing a separate customized compare().

7) Many object oriented features in C++ are implemented using function pointers in C. For example virtual functions. Class methods are another example implemented using function pointers. Refer this book for more details.

References:

http://www.cs.cmu.edu/~ab/15-123S11/AnnotatedNotes/Lecture14.pdf

http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-087-practical-programming-in-c-january-iap-2010/lecture-notes/MIT6_087IAP10_lec08.pdf

http://www.cs.cmu.edu/~guna/15-123S11/Lectures/Lecture14.pdf

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

What are near, far and huge pointers?

These are some old concepts used in 16 bit intel architectures in the days of MS DOS, not much useful anymore.

Near pointer is used to store 16 bit addresses means within current segment on a 16 bit machine. The limitation is that we can only access 64kb of data at a time.

A far pointer is typically 32 bit that can access memory outside current segment. To use this, compiler allocates a segment register to store segment address, then another register to store offset within current segment.

Like far pointer, **huge pointer** is also typically 32 bit and can access outside segment. In case of far pointers, a segment is fixed. In far pointer, the segment part cannot be modified, but in Huge it can be

See below links for more details.

http://www.answers.com/Q/What_are_near_far_and_huge_pointers_in_C

https://www.quora.com/What-is-the-difference-between-near-far-huge-pointers-in-C-C++

http://stackoverflow.com/questions/8727122/explain-the-difference-between-near-far-and-huge-pointers-in-c

Source: http://qa.geeksforgeeks.org/664/near-far-huge-pointers?show=664#q664

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Generic Linked List in C

Unlike C++ and Java, C doesn't support generics. How to create a linked list in C that can be used for any data type? In C, we can use void pointer and function pointer to implement the same functionality. The great thing about void pointer is it can be used to point to any data type. Also, size of all types of pointers is always is same, so we can always allocate a linked list node. Function pointer is needed process actual content stored at address pointed by void pointer.

Following is a sample C code to demonstrate working of generic linked list.

```
// C program for generic linked list
#include<stdio.h>
#include<stdlib.h>
```

```
/* A linked list node */
struct node
{
    // Any data type can be stored in this node
    void  *data;

    struct node *next;
};

/* Function to add a node at the beginning of Linked List.
   This function expects a pointer to the data to be added
   and size of the data type */
void push(struct node** head_ref, void *new_data, size_t data_size)
{
    // Allocate memory for node
    struct node* new_node = (struct node*)malloc(sizeof(struct node));

    new_node->data  = malloc(data_size);
    new_node->next = (*head_ref);

    // Copy contents of new_data to newly allocated memory.
    // Assumption: char takes 1 byte.
    int i;
    for (i=0; i<data_size; i++)
        *(char *)(new_node->data + i) = *(char *)(new_data + i);

    // Change head pointer as new node is added at the beginning
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list. fpitr is used
   to access the function to be used for printing current node data.
   Note that different data types need different specifier in printf() */
void printList(struct node *node, void (*fptr)(void *))
{
    while (node != NULL)
    {
        (*fptr)(node->data);
        node = node->next;
    }
}

// Function to print an integer
void printInt(void *n)
{
   printf(" %d", *(int *)n);
}

// Function to print a float
void printFloat(void *f)
{
   printf(" %f", *(float *)f);
}

/* Driver program to test above function */
int main()
{
    struct node *start = NULL;

    // Create and print an int linked list
    unsigned int_size = sizeof(int);
    int arr[] = {10, 20, 30, 40, 50}, i;
    for (i=4; i>=0; i--)
       push(&start, &arr[i], int_size);
    printf("Created integer linked list is \n");
    printList(start, printInt);

    // Create and print a float linked list
    unsigned float_size = sizeof(float);
    start = NULL;
    float arr2[] = {10.1, 20.2, 30.3, 40.4, 50.5};
    for (i=4; i>=0; i--)
       push(&start, &arr2[i], float_size);
    printf("\n\nCreated float linked list is \n");
    printList(start, printFloat);

    return 0;
}
```

Output:

```
Created integer linked list is
 10 20 30 40 50

Created float linked list is
 10.100000 20.200001 30.299999 40.400002 50.500000
```

This article is contributed by **Himanshu Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

Advanced Data Structure
C/C++ Puzzles
Linked Lists
cpp-pointer

Enumeration (or enum) in C

Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

The keyword 'enum' is used to declare new enumeration types in C and C++. Following is an example of enum declaration.

```
enum State {Working = 1, Failed = 0};
```

Following are some interesting facts about initialization of enum.

1. Two enum names can have same value. For example, in the following C program both 'Failed' and 'Freezed' have same value 0.

```
#include <stdio.h>
enum State {Working = 1, Failed = 0, Freezed = 0};

int main()
{
   printf("%d, %d, %d", Working, Failed, Freezed);
   return 0;
}
```

Output:

```
1, 0, 0
```

2. If we do not explicitly assign values to enum names, the compiler by default assigns values starting from 0. For example, in the following C program, sunday gets value 0, monday gets 1, and so on.

```
#include <stdio.h>
enum day {sunday, monday, tuesday, wednesday, thursday, friday, saturday};

int main()
{
    enum day d = thursday;
    printf("The day number stored in d is %d", d);
    return 0;
}
```

Output:

```
The day number stored in d is 4
```

3. We can assign values to some name in any order. All unassigned names get value as value of previous name plus one.

```
#include <stdio.h>
enum day {sunday = 1, monday, tuesday = 5,
        wednesday, thursday = 10, friday, saturday};

int main()
{
   printf("%d %d %d %d %d %d %d", sunday, monday, tuesday,
          wednesday, thursday, friday, saturday);
   return 0;
}
```

Output:

```
1 2 5 6 10 11 12
```

4. The value assigned to enum names must be some integeral constant, i.e., the value must be in range from minimum possible integer value to maximum possible integer value.

5. All enum constants must be unique in their scope. For example, the following program fails in compilation.

```
enum state  {working, failed};
enum result {failed, passed};

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

Output:

```
Compile Error: 'failed' has a previous declaration as 'state failed'
```

Exercise:
Predict the output of following C programs

Program 1:

```
#include <stdio.h>
enum day {sunday = 1, tuesday, wednesday, thursday, friday, saturday};

int main()
{
   enum day d = thursday;
   printf("The day number stored in d is %d", d);
   return 0;
}
```

Program 2:

```
#include <stdio.h>
enum State {WORKING = 0, FAILED, FREEZED};
enum State currState = 2;

enum State FindState() {
   return currState;
}

int main() {
  (FindState() == WORKING)? printf("WORKING"): printf("NOT WORKING");
```

```
   return 0;
}
```

Enum vs Macro

We can also use macros to define names constants. For example we can define 'Working' and 'Failed' using following macro.

```
#define Working 0
#define Failed 1
#define Freezed 2
```

There are multiple advantages of using enum over macro when many related named constants have integral values.

a) Enums follow scope rules.

b) Enum variables are automatically assigned values. Following is simpler

```
 enum state  {Working, Failed, Freezed};
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C

Structures in C

What is a structure?

A structure is a user defined data type in C/C++. A structure creates a data type that can be used to group items of possibly different types into a single type.

How to create a structure?

'struct' keyword is used to create a structure. Following is an example.

```
struct addrress
{
   char name[50];
   char street[100];
   char city[50];
   char state[20]
   int pin;
};
```

How to declare structure variables?

A structure variable can either be declared with structure declaration or as a separate declaration like basic types.

```
// A variable declaration with structure declaration.
struct Point
{
   int x, y;
} p1;  // The variable p1 is declared with 'Point'


// A variable declaration like basic data types
struct Point
{
   int x, y;
};

int main()
```

```
{
   struct Point p1;  // The variable p1 is declared like a normal variable
}
```

Note: In C++, the struct keyword is optional before in declaration of variable. In C, it is mandatory.

How to initialize structure members?

Structure members **cannot be** initialized with declaration. For example the following C program fails in compilation.

```
struct Point
{
   int x = 0;  // COMPILER ERROR:  cannot initialize members here
   int y = 0;  // COMPILER ERROR:  cannot initialize members here
};
```

The reason for above error is simple, when a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.

Structure members **can be** initialized using curly braces '{}'. For example, following is a valid initialization.

```
struct Point
{
   int x, y;
};

int main()
{
   // A valid initialization. member x gets value 0 and y
   // gets value 1.  The order of declaration is followed.
   struct Point p1 = {0, 1};
}
```

How to access structure elements?

Structure members are accessed using dot (.) operator.

```
struct Point
{
   int x, y;
};

int main()
{
   struct Point p1 = {0, 1};

   // Accesing members of point p1
   p1.x = 20;
   printf ("x = %d, y = %d", p1.x, p1.y);

   return 0;
}
```

Output:

```
20  1
```

What is designated Initialization?

Designated Initialization allows structure members to be initialized in any order. This feature has been added in C99 standard.

```
struct Point
{
   int x, y, z;
};

int main()
{
   // Examples of initializtion using designated initialization
   struct Point p1 = {.y = 0, .z = 1, .x = 2};
   struct Point p2 = {.x = 20};

   printf ("x = %d, y = %d, z = %d\n", p1.x, p1.y, p1.z);
   printf ("x = %d", p2.x);
   return 0;
}
```

Output:

```
x = 2, y = 0, z = 1
x = 20
```

This feature is not available in C++ and works only in C.

What is an array of structures?

Like other primitive data types, we can create an array of structures.

```
struct Point
{
    int x, y;
};

int main()
{
    // Create an array of structures
    struct Point arr[10];

    // Access array members
    arr[0].x = 10;
    arr[0].y = 20;

    printf("%d %d", arr[0].x, arr[0].y);
    return 0;
}
```

Output:

```
10 20
```

What is a structure pointer?

Like primitive types, we can have pointer to a structure. If we have a pointer to structure, members are accessed using arrow (->) operator.

```
struct Point
{
    int x, y;
};

int main()
{
    struct Point p1 = {1, 2};

    // p2 is a pointer to structure p1
    struct Point *p2 = &p1;

    // Accessing structure members using structure pointer
    printf("%d %d", p2->x, p2->y);
    return 0;
}
```

Output:

```
1 2
```

What is structure member alignment?

See http://www.geeksforgeeks.org/structure-member-alignment-padding-and-data-packing/

We will soon be discussing union and other struct related topics in C. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C

Union in C

Like Structures, union is a user defined data type. In union, all members share the same memory location. For example in the following C program, both x and y share the same location. If we change x, we can see the changes being reflected in y.

```
#include <stdio.h>

// Declaration of union is same as structures
union test
{
  int x, y;
};

int main()
{
   // A union variable t
   union test t;

   t.x = 2; // t.y also gets value 2
   printf ("After making x = 2:\n x = %d, y = %d\n\n",
          t.x, t.y);

   t.y = 10;  // t.x is also updated to 10
   printf ("After making Y = 'A':\n x = %d, y = %d\n\n",
          t.x, t.y);
   return 0;
}
```

Output:

```
After making x = 2:
 x = 2, y = 2

After making Y = 'A':
 x = 10, y = 10
```

How is the size of union decided by compiler?

Size of a union is taken according the size of largest member in union.

```
#include <stdio.h>

union test1
{
  int x;
  int y;
};

union test2
{
  int x;
  char y;
};

union test3
{
  int arr[10];
  char y;
};

int main()
{
   printf ("sizeof(test1) = %d, sizeof(test2) = %d,"
          "sizeof(test3) =  %d", sizeof(test1),
          sizeof(test2), sizeof(test3));
   return 0;
}
```

Output

```
sizeof(test1) = 4, sizeof(test2) = 4,sizeof(test3) =  40
```

Pointers to unions?
Like structures, we can have pointers to unions and can access members using arrow operator (->). The following example demonstrates the same.

```
union test
{
  int x;
  char y;
};

int main()
{
  union test p1;
  p1.x = 65;

  // p2 is a pointer to union p1
  union test *p2 = &p1;

  // Accessing union members using pointer
  printf("%d %c", p2->x, p2->y);
  return 0;
}
```

```
65 A
```

What are applications of union?
Unions can be useful in many situations where we want to use same memory for two ore more members. For example, suppose we want to implement a binary tree data structure where each leaf node has a double data value, while each internal node has pointers to two children, but no data. If we declare this as:

```
struct NODE {
 struct NODE *left;
 struct NODE *right;
 double data;
};
```

then every node requires 16 bytes, with half the bytes wasted for each type of node. On the other hand, if we declare a node as following, then we can save space.

```
struct NODE
{
   bool is_leaf;
   union
   {
        struct
        {
            struct NODE *left;
            struct NODE *right;
        } internal;
        double data;
    } info;
};
```

The above example is taken from Computer Systems : A Programmer's Perspective (English) 2nd Edition book.

References:

http://en.wikipedia.org/wiki/Union_type

Computer Systems : A Programmer's Perspective (English) 2nd Edition

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C

Struct Hack

What will be the size of following structure?

```
struct employee
{
    int  emp_id;
    int  name_len;
    char name[0];
};
```

4 + 4 + 0 = 8 bytes.

And what about size of "name[0]". In gcc, when we create an array of zero length, it is considered as array of incomplete type that's why gcc reports its size as "0" bytes. This technique is known as "Stuct Hack". When we create array of zero length inside structure, it must be (and only) last member of structure. Shortly we will see how to use it.
"Struct Hack" technique is used to create variable length member in a structure. In the above structure, string length of "name" is not fixed, so we can use "name" as variable length array.

Let us see below memory allocation.

```
struct employee *e = malloc(sizeof(*e) + sizeof(char) * 128);
```

is equivalent to

```
struct employee
{
 int  emp_id;
 int  name_len;
 char name[128]; /* character array of size 128 */
};
```

And below memory allocation

```
struct employee *e = malloc(sizeof(*e) + sizeof(char) * 1024);
```

is equivalent to

```
struct employee
{
 int emp_id;
 int name_len;
 char name[1024]; /* character array of size 1024 */
};
```

Note: since name is character array, in malloc instead of “sizeof(char) * 128”, we can use “128” directly. sizeof is used to avoid confusion.

Now we can use “name” same as pointer. e.g.

```
e->emp_id  = 100;
e->name_len = strlen("Geeks For Geeks");
strncpy(e->name, "Geeks For Geeks", e->name_len);
```

When we allocate memory as given above, compiler will allocate memory to store “emp_id” and “name_len” plus contiguous memory to store “name”. When we use this technique, gcc guaranties that, “name” will get contiguous memory.
Obviously there are other ways to solve problem, one is we can use character pointer. But there is no guarantee that character pointer will get contiguous memory, and we can take advantage of this contiguous memory. For example, by using this technique, we can allocate and deallocate memory by using single malloc and free call (because memory is contagious). Other advantage of this is, suppose if we want to write data, we can write whole data by using single “write()” call. e.g.

```
write(fd, e, sizeof(*e) + name_len); /* write emp_id + name_len + name */
```

If we use character pointer, then we need 2 write calls to write data. e.g.

```
write(fd, e, sizeof(*e));   /* write emp_id + name_len */
write(fd, e->name, e->name_len); /* write name */
```

Note: In C99, there is feature called “flexible array members”, which works same as “Struct Hack”

This article is compiled by **Narendra Kangralkar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-structure

Structure Member Alignment, Padding and Data Packing

What do we mean by data alignment, structure packing and padding?

Predict the output of following program.

```
#include <stdio.h>

// Alignment requirements
// (typical 32 bit machine)

// char         1 byte
// short int    2 bytes
// int          4 bytes
// double       8 bytes

// structure A
typedef struct structa_tag
{
   char       c;
   short int   s;
} structa_t;

// structure B
typedef struct structb_tag
{
   short int   s;
   char       c;
   int        i;
} structb_t;
```

```
// structure C
typedef struct structc_tag
{
  char      c;
  double    d;
  int       s;
} structc_t;

// structure D
typedef struct structd_tag
{
  double    d;
  int       s;
  char      c;
} structd_t;

int main()
{
  printf("sizeof(structa_t) = %d\n", sizeof(structa_t));
  printf("sizeof(structb_t) = %d\n", sizeof(structb_t));
  printf("sizeof(structc_t) = %d\n", sizeof(structc_t));
  printf("sizeof(structd_t) = %d\n", sizeof(structd_t));

  return 0;
}
```

Before moving further, write down your answer on a paper, and read on. If you urge to see explanation, you may miss to understand any lacuna in your analogy. Also read the post by Kartik.

Data Alignment:

Every data type in C/C++ will have alignment requirement (infact it is mandated by processor architecture, not by language). A processor will have processing word length as that of data bus size. On a 32 bit machine, the processing word size will be 4 bytes.

Historically memory is byte addressable and arranged sequentially. If the memory is arranged as single bank of one byte width, the processor needs to issue 4 memory read cycles to fetch an integer. It is more economical to read all 4 bytes of integer in one memory cycle. To take such advantage, the memory will be arranged as group of 4 banks as shown in the above figure.

The memory addressing still be sequential. If bank 0 occupies an address X, bank 1, bank 2 and bank 3 will be at (X + 1), (X + 2) and (X + 3) addresses. If an integer of 4 bytes is allocated on X address (X is multiple of 4), the processor needs only one memory cycle to read entire integer.

Where as, if the integer is allocated at an address other than multiple of 4, it spans across two rows of the banks as shown in the below figure. Such an integer requires two memory read cycle to fetch the data.

A variable's ***data alignment*** deals with the way the data stored in these banks. For example, the natural alignment of ***int*** on 32-bit machine is 4 bytes. When a data type is naturally aligned, the CPU fetches it in minimum read cycles.

Similarly, the natural alignment of ***short int*** is 2 bytes. It means, a ***short int*** can be stored in bank 0 – bank 1 pair or bank 2 – bank 3 pair. A ***double*** requires 8 bytes, and occupies two rows in the memory banks. Any misalignment of ***double*** will force more than two read cycles to fetch ***double*** data.

Note that a **double** variable will be allocated on 8 byte boundary on 32 bit machine and requires two memory read cycles. On a 64 bit machine, based on number of banks, **double** variable will be allocated on 8 byte boundary and requires only one memory read cycle.

Structure Padding:

In C/C++ a structures are used as data pack. It doesn't provide any data encapsulation or data hiding features (C++ case is an exception due to its semantic similarity with classes).

Because of the alignment requirements of various data types, every member of structure should be naturally aligned. The members of structure allocated sequentially increasing order. Let us analyze each struct declared in the above program.

Output of Above Program:

For the sake of convenience, assume every structure type variable is allocated on 4 byte boundary (say 0x0000), i.e. the base address of structure is multiple of 4 (need not necessary always, see explanation of structc_t).

structure A

The *structa_t* first element is *char* which is one byte aligned, followed by *short int*. short int is 2 byte aligned. If the the short int element is immediately allocated after the char element, it will start at an odd address boundary. The compiler will insert a padding byte after the char to ensure short int will have an address multiple of 2 (i.e. 2 byte aligned). The total size of structa_t will be sizeof(char) + 1 (padding) + sizeof(short), 1 + 1 + 2 = 4 bytes.

structure B

The first member of *structb_t* is short int followed by char. Since char can be on any byte boundary no padding required in between short int and char, on total they occupy 3 bytes. The next member is int. If the int is allocated immediately, it will start at an odd byte boundary. We need 1 byte padding after the char member to make the address of next int member is 4 byte aligned. On total, the *structb_t* requires 2 + 1 + 1 (padding) + 4 = 8 bytes.

structure C – Every structure will also have alignment requirements

Applying same analysis, *structc_t* needs sizeof(char) + 7 byte padding + sizeof(double) + sizeof(int) = 1 + 7 + 8 + 4 = 20 bytes. However, the sizeof(structc_t) will be 24 bytes. It is because, along with structure members, structure type variables will also have natural alignment. Let us understand it by an example. Say, we declared an array of structc_t as shown below

```
structc_t structc_array[3];
```

Assume, the base address of *structc_array* is 0x0000 for easy calculations. If the structc_t occupies 20 (0x14) bytes as we calculated, the second structc_t array element (indexed at 1) will be at 0x0000 + 0x0014 = 0x0014. It is the start address of index 1 element of array. The double member of this structc_t will be allocated on 0x0014 + 0x1 + 0x7 = 0x001C (decimal 28) which is not multiple of 8 and conflicting with the alignment requirements of double. As we mentioned on the top, the alignment requirement of double is 8 bytes.

Inorder to avoid such misalignment, compiler will introduce alignment requirement to every structure. It will be as that of the largest member of the structure. In our case alignment of structa_t is 2, structb_t is 4 and structc_t is 8. If we need nested structures, the size of largest inner structure will be the alignment of immediate larger structure.

In structc_t of the above program, there will be padding of 4 bytes after int member to make the structure size multiple of its alignment. Thus the sizeof (structc_t) is 24 bytes. It guarantees correct alignment even in arrays. You can cross check.

structure D – How to Reduce Padding?

By now, it may be clear that padding is unavoidable. There is a way to minimize padding. The programmer should declare the structure members in their increasing/decreasing order of size. An example is structd_t given in our code, whose size is 16 bytes in lieu of 24 bytes of structc_t.

What is structure packing?

Some times it is mandatory to avoid padded bytes among the members of structure. For example, reading contents of ELF file header or BMP or JPEG file header. We need to define a structure similar to that of the header layout and map it. However, care should be exercised in accessing such members. Typically reading byte by byte is an option to avoid misaligned exceptions. There will be hit on performance.

Most of the compilers provide non standard extensions to switch off the default padding like pragmas or command line switches. Consult the documentation of respective compiler for more details.

Pointer Mishaps:

There is possibility of potential error while dealing with pointer arithmetic. For example, dereferencing a generic pointer (void *) as shown below can cause misaligned exception,

```
// Deferencing a generic pointer (not safe)
// There is no guarantee that pGeneric is integer aligned
*(int *)pGeneric;
```

It is possible above type of code in programming. If the pointer *pGeneric* is not aligned as per the requirements of casted data type, there is possibility to get misaligned exception.

Infact few processors will not have the last two bits of address decoding, and there is no way to access *misaligned* address. The processor generates misaligned exception, if the programmer tries to access such address.

A note on malloc() returned pointer

The pointer returned by malloc() is *void **. It can be converted to any data type as per the need of programmer. The implementer of malloc() should return a pointer that is aligned to maximum size of primitive data types (those defined by compiler). It is usually aligned to 8 byte boundary on 32 bit machines.

Object File Alignment, Section Alignment, Page Alignment

These are specific to operating system implementer, compiler writers and are beyond the scope of this article. Infact, I don't have much information.

General Questions:

1. Is alignment applied for stack?

Yes. The stack is also memory. The system programmer should load the stack pointer with a memory address that is properly aligned. Generally, the processor won't check stack alignment, it is the programmer's responsibility to ensure proper alignment of stack memory. Any misalignment will cause run time surprises.

For example, if the processor word length is 32 bit, stack pointer also should be aligned to be multiple of 4 bytes.

2. If *char* data is placed in a bank other bank 0, it will be placed on wrong data lines during memory read. How the processor handles *char* type?

Usually, the processor will recognize the data type based on instruction (e.g. LDRB on ARM processor). Depending on the bank it is stored, the processor shifts the byte onto least significant data lines.

3. When arguments passed on stack, are they subjected to alignment?

Yes. The compiler helps programmer in making proper alignment. For example, if a 16-bit value is pushed onto a 32-bit wide stack, the value is automatically padded with zeros out to 32 bits. Consider the following program.

```
void argument_alignment_check( char c1, char c2 )
{
  // Considering downward stack
  // (on upward stack the output will be negative)
  printf("Displacement %d\n", (int)&c2 - (int)&c1);
}
```

The output will be 4 on a 32 bit machine. It is because each character occupies 4 bytes due to alignment requirements.

4. What will happen if we try to access a misaligned data?

It depends on processor architecture. If the access is misaligned, the processor automatically issues sufficient memory read cycles and packs the data properly onto the data bus. The penalty is on performance. Where as few processors will not have last two address lines, which means there is no-way to access odd byte boundary. Every data access must be aligned (4 bytes) properly. A misaligned access is critical exception on such processors. If the exception is ignored, read data will be incorrect and hence the results.

5. Is there any way to query alignment requirements of a data type.

Yes. Compilers provide non standard extensions for such needs. For example, __alignof() in Visual Studio helps in getting the alignment requirements of data type. Read MSDN for details.

6. When memory reading is efficient in reading 4 bytes at a time on 32 bit machine, why should a **double** type be aligned on 8 byte boundary?

It is important to note that most of the processors will have math co-processor, called Floating Point Unit (FPU). Any floating point operation in the code will be translated into FPU instructions. The main processor is nothing to do with floating point execution. All this will be done behind the scenes.

As per standard, double type will occupy 8 bytes. And, every floating point operation performed in FPU will be of 64 bit length. Even float types will be promoted to 64 bit prior to execution.

The 64 bit length of FPU registers forces double type to be allocated on 8 byte boundary. I am assuming (I don't have concrete information) in case of FPU operations, data fetch might be different, I mean the data bus, since it goes to FPU. Hence, the address decoding will be different for double types (which is expected to be on 8 byte boundary). It means, *the address decoding circuits of floating point unit will not have last 3 pins*.

Answers:

```
sizeof(structa_t) = 4
sizeof(structb_t) = 8
sizeof(structc_t) = 24
sizeof(structd_t) = 16
```

Update: 1-May-2013

It is observed that on latest processors we are getting size of struct_c as 16 bytes. I yet to read relevant documentation. I will update once I got proper information (written to few experts in hardware).

On older processors (AMD Athlon X2) using same set of tools (GCC 4.7) I got struct_c size as 24 bytes. The size depends on how memory banking organized at the hardware level.

– – – by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

Articles
C/C++ Puzzles
About Venki
Software Engineer
View all posts by Venki →

Operations on struct variables in C

In C, the only operation that can be applied to *struct* variables is assignment. Any other operation (e.g. equality check) is not allowed on *struct* variables.
For example, program 1 works without any error and program 2 fails in compilation.

Program 1

```
#include <stdio.h>

struct Point {
  int x;
  int y;
};

int main()
{
  struct Point p1 = {10, 20};
  struct Point p2 = p1; // works: contents of p1 are copied to p1
  printf(" p2.x = %d, p2.y = %d", p2.x, p2.y);
  getchar();
  return 0;
}
```

Program 2

```
#include <stdio.h>

struct Point {
  int x;
  int y;
};

int main()
{
  struct Point p1 = {10, 20};
  struct Point p2 = p1; // works: contents of p1 are copied to p1
  if (p1 == p2)  // compiler error: cannot do equality check for
                // whole structures
  {
    printf("p1 and p2 are same ");
  }
  getchar();
  return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Bit Fields in C

In C, we can specify size (in bits) of structure and union members. The idea is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is withing a small range.

For example, consider the following declaration of date without use of bit fields.

```
#include <stdio.h>

// A simple representation of date
struct date
{
   unsigned int d;
   unsigned int m;
   unsigned int y;
};

int main()
{
   printf("Size of date is %d bytes\n", sizeof(struct date));
   struct date dt = {31, 12, 2014};
   printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
}
```

Output:

```
Size of date is 12 bytes
Date is 31/12/2014
```

The above representation of ‘date’ takes 12 bytes on a compiler where an unsigned int takes 4 bytes. Since we know that the value of d is always from 1 to 31, value of m is from 1 to 12, we can optimize the space using bit fields.

```
#include <stdio.h>

// A space optimized representation of date
struct date
{
   // d has value between 1 and 31, so 5 bits
   // are sufficient
   unsigned int d: 5;

   // m has value between 1 and 12, so 4 bits
   // are sufficient
   unsigned int m: 4;

   unsigned int y;
};
```

```
int main()
{
   printf("Size of date is %d bytes\n", sizeof(struct date));
   struct date dt = {31, 12, 2014};
   printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
   return 0;
}
```

Output:

```
Size of date is 8 bytes
Date is 31/12/2014
```

Following are some interesting facts about bit fields in C.

1) A special unnamed bit field of size 0 is used to force alignment on next boundary. For example consider the following program.

```
#include <stdio.h>

// A structure without forced alignment
struct test1
{
   unsigned int x: 5;
   unsigned int y: 8;
};

// A structure with forced alignment
struct test2
{
   unsigned int x: 5;
   unsigned int: 0;
   unsigned int y: 8;
};

int main()
{
   printf("Size of test1 is %d bytes\n", sizeof(struct test1));
   printf("Size of test2 is %d bytes\n", sizeof(struct test2));
   return 0;
}
```

Output:

```
Size of test1 is 4 bytes
Size of test2 is 8 bytes
```

2) We cannot have pointers to bit field members as they may not start at a byte boundary.

```
#include <stdio.h>
struct test
{
   unsigned int x: 5;
   unsigned int y: 5;
   unsigned int z;
};
int main()
{
   test t;

   // Uncommenting the following line will make
   // the program compile and run
   printf("Address of t.x is %p", &t.x);

   // The below line works fine as z is not a
   // bit field member
   printf("Address of t.z is %p", &t.z);
   return 0;
}
```

Output:

```
error: attempt to take address of bit-field structure member 'test::x'
```

3) It is implementation defined to assign an out-of-range value to a bit field member.

```
#include <stdio.h>
struct test
{
   unsigned int x: 2;
   unsigned int y: 2;
   unsigned int z: 2;
};
int main()
{
   test t;
   t.x = 5;
   printf("%d", t.x);
   return 0;
}
```

Output:

```
Implementation-Dependent
```

4) In C++, we can have static members in a structure/class, but bit fields cannot be static.

```
// The below C++ program compiles and runs fine
struct test1 {
   static unsigned int x;
};
int main() {  }


// But below C++ program fails in compilation as bit fields
// cannot be static
struct test1 {
   static unsigned int x: 5;
};
int main() {  }
// error: static member 'x' cannot be a bit-field
```

5) Array of bit fields is not allowed. For example, the below program fails in compilation.

```
struct test
{
  unsigned int x[10]: 5;
};

int main()
{

}
```

Output:

```
error: bit-field 'x' has invalid type
```

Exercise:

Predict the output of following programs. Assume that unsigned int takes 4 bytes and long int takes 8 bytes.

1)

```
#include <stdio.h>
struct test
{
  unsigned int x;
  unsigned int y: 33;
  unsigned int z;
};
int main()
{
  printf("%d", sizeof(struct test));
  return 0;
}
```

2)

```
#include <stdio.h>
struct test
{
  unsigned int x;
  long int y: 33;
  unsigned int z;
};
int main()
{
  struct test t;
  unsigned int *ptr1 = &t.x;
  unsigned int *ptr2 = &t.z;
  printf("%d", ptr2 - ptr1);
  return 0;
}
```

3)

```
union test
{
 unsigned int x: 3;
 unsigned int y: 3;
 int z;
};

int main()
{
  union test t;
  t.x = 5;
  t.y = 4;
  t.z = 1;
  printf("t.x = %d, t.y = %d, t.z = %d",
         t.x, t.y, t.z);
  return 0;
}
```

4) Use bit fields in C to figure out a way whether a machine is little endian or big endian.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

Bit Magic
C/C++ Puzzles
cpp-structure

Structure Sorting (By Multiple Rules) in C++

Prerequisite : Structures in C

Name and marks in different subjects (physics, chemistry and maths) are given for all students. The task is to compute total marks and ranks of all students. And finally display all students sorted by rank.

Rank of student is computed using below rules.

1. If total marks are different, then students with higher marks gets better rank.
2. If total marks are same, then students with higher marks in Maths gets better rank.
3. If total marks are same and marks in Maths are also same, then students with better marks in Physics gets better rank.
4. If all marks (total, Maths, Physics and Chemistry) are same, then any student can be assigned bedtter rank.

We use below structure to store details of students.

```
struct Student
{
   string name; // Given
   int math;  // Marks in math (Given)
   int phy;   // Marks in Physics (Given)
   int che;   // Marks in Chemistry (Given)
   int total; // Total marks (To be filled)
   int rank;  // Rank of student (To be filled)
};
```

We use std::sort() for **Structure Sorting**. In Structure sorting, all the respective properties possessed by the structure object are sorted on the basis of one (or more) property of the object.

In this example, marks of students in different subjects are provided by user. These marks in individual subjects are added to calculate the total marks of the student, which is then used to sort different students on the basis of their ranks (as explained above).

```
// C++ program to demonstrate structure sorting in C++
#include <bits/stdc++.h>
using namespace std;

struct Student
{
   string name; // Given
   int math;  // Marks in math (Given)
   int phy;   // Marks in Physics (Given)
   int che;   // Marks in Chemistry (Given)
   int total; // Total marks (To be filled)
   int rank;  // Rank of student (To be filled)
};

// Function for comparing two students according
// to given rules
bool compareTwoStudents(Student a, Student b)
{
   // If total marks are not same then
   // returns true for higher total
   if (a.total != b.total )
      return a.total > b.total;

   // If marks in Maths are not same then
   // returns true for higher marks
   if (a.math != b.math)
         return a.math > b.math;

   return (a.phy > b.phy);
}

// Fills total marks and ranks of all Students
void computeRanks(Student a[], int n)
{
   // To calculate total marks for all Students
   for (int i=0; i<n; i++)
      a[i].total = a[i].math + a[i].phy + a[i].che;
```

```
    // Sort structure array using user defined
    // function compareTwoStudents()
    sort(a, a+5, compareTwoStudents);

    // Assigning ranks after sorting
    for (int i=0; i<n; i++)
        a[i].rank = i+1;
}

// Driver code
int main()
{
    int n = 5;

    // array of structure objects
    Student a[n];

    // Details of Student 1
    a[0].name = "Bryan" ;
    a[0].math = 80 ;
    a[0].phy = 95 ;
    a[0].che = 85 ;

    // Details of Student 2
    a[1].name= "Kevin" ;
    a[1].math= 95 ;
    a[1].phy= 85 ;
    a[1].che= 99 ;

    // Details of Student 3
    a[2].name = "Nick" ;
    a[2].math = 95 ;
    a[2].phy = 85 ;
    a[2].che = 80 ;

    // Details of Student 4
    a[3].name = "AJ" ;
    a[3].math = 80 ;
    a[3].phy = 70 ;
    a[3].che = 90 ;

    // Details of Student 5
    a[4].name = "Howie" ;
    a[4].math = 80 ;
    a[4].phy = 80 ;
    a[4].che = 80 ;

    computeRanks(a, n);

    //Column names for displaying data
    cout << "Rank" <<"\t" << "Name" << "\t";
    cout << "Maths" <<"\t" <<"Physics" <<"\t"
         << "Chemistry";
    cout << "\t" << "Total\n";

    // Display details of Students
    for (int i=0; i<n; i++)
    {
        cout << a[i].rank << "\t";
        cout << a[i].name << "\t";
        cout << a[i].math << "\t"
             << a[i].phy << "\t"
             << a[i].che << "\t\t";
        cout << a[i].total <<"\t";
        cout <<"\n";
    }

    return 0;
}
```

Output:

```
Rank Name Maths Physics Chemistry Total
1 Kevin 95 85 99  279
2 Nick 95 85 80  260
3 Bryan 80 95 85  260
4 Howie 80 80 80  240
5 AJ 80 70 90  240
```

Related Articles:

sort() in C++ STL

Comparator function of qsort() in C

C qsort() vs C++ sort()

Sort an array according to count of set bits

This article is contributed by **Abhinav Tiwari** . If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
Sorting
cpp-structure

Memory Layout of C Programs

A typical memory representation of C program consists of following sections.

1. Text segment
2. Initialized data segment
3. Uninitialized data segment
4. Stack
5. Heap

A typical memory layout of a running process

1. Text Segment:

A text segment , also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions.

As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.

Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

2. Initialized Data Segment:

Initialized data segment, usually called simply the Data Segment. A data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

Note that, data segment is not read-only, since the values of the variables can be altered at run time.

This segment can be further classified into initialized read-only area and initialized read-write area.

For instance the global string defined by char s[] = “hello world” in C and a C statement like int debug=1 outside the main (i.e. global) would be stored in initialized read-write area. And a global C statement like const char* string = “hello world” makes the string literal “hello world” to be stored in initialized read-only area and the character pointer variable string in initialized read-write area.

Ex: static int i = 10 will be stored in data segment and global int i = 10 will also be stored in data segment

3. Uninitialized Data Segment:
Uninitialized data segment, often called the “bss” segment, named after an ancient assembler operator that stood for “block started by symbol.” Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing

uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

For instance a variable declared static int i; would be contained in the BSS segment.
For instance a global variable declared int j; would be contained in the BSS segment.

4. Stack:
The stack area traditionally adjoined the heap area and grew the opposite direction; when the stack pointer met the heap pointer, free memory was exhausted. (With modern large address spaces and virtual memory techniques they may be placed almost anywhere, but they still typically grow opposite directions.)

The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. On the standard PC x86 computer architecture it grows toward address zero; on some other architectures it grows the opposite direction. A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack. The set of values pushed for one function call is termed a “stack frame”; A stack frame consists at minimum of a return address.

Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn’t interfere with the variables from another instance of the function.

5. Heap:
Heap is the segment where dynamic memory allocation usually takes place.

The heap area begins at the end of the BSS segment and grows to larger addresses from there.The Heap area is managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size (note that the use of brk/sbrk and a single “heap area” is not required to fulfill the contract of malloc/realloc/free; they may also be implemented using mmap to reserve potentially non-contiguous regions of virtual memory into the process’ virtual address space). The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

Examples.

The size(1) command reports the sizes (in bytes) of the text, data, and bss segments. (for more details please refer man page of size(1))

1. Check the following simple C program

```
#include <stdio.h>

int main(void)
{
    return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text     data      bss      dec      hex   filename
960       248        8      1216      4c0   memory-layout
```

2. Let us add one global variable in program, now check the size of bss (highlighted in red color).

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

```
int global; /* Uninitialized variable stored in bss*/

int main(void)
{
   return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text     data      bss      dec      hex   filename
 960      248       12      1220      4c4   memory-layout
```

3. Let us add one static variable which is also stored in bss.

```
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
   static int i; /* Uninitialized static variable stored in bss */
   return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text     data      bss      dec      hex   filename
 960      248       16      1224      4c8   memory-layout
```

4. Let us initialize the static variable which will then be stored in Data Segment (DS)

```
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
   static int i = 100; /* Initialized static variable stored in DS*/
   return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text     data      bss      dec      hex   filename
960       252       12      1224      4c8   memory-layout
```

5. Let us initialize the global variable which will then be stored in Data Segment (DS)

```
#include <stdio.h>

int global = 10; /* initialized global variable stored in DS*/

int main(void)
{
   static int i = 100; /* Initialized static variable stored in DS*/
   return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text     data      bss      dec      hex   filename
960       256       8      1224      4c8   memory-layout
```

This article is compiled by **Narendra Kangralkar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source:

http://en.wikipedia.org/wiki/Data_segment

http://en.wikipedia.org/wiki/Code_segment

http://en.wikipedia.org/wiki/.bss

http://www.amazon.com/Advanced-Programming-UNIX-Environment-2nd/dp/0201433079

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
Memory Layout Of C Program
About Narendra
A Computer Engineer from Pune. My ares of interests includes C, Algorithms and Data Structure, Linux Kernel Development. I am currently working as Software Engineer in storage domain.
View all posts by Narendra →

How to deallocate memory without using free() in C?

Question: How to deallocate dynamically allocate memory without using "free()" function.

Solution: Standard library function realloc() can be used to deallocate previously allocated memory. Below is function declaration of "realloc()" from "stdlib.h"

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

If "size" is zero, then call to realloc is equivalent to "free(ptr)". And if "ptr" is NULL and size is non-zero then call to realloc is equivalent to "malloc(size)".

Let us check with simple example.

```
/* code with memory leak */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int *ptr = (int*)malloc(10);

   return 0;
}
```

Check the leak summary with valgrind tool. It shows memory leak of 10 bytes, which is highlighed in red colour.

```
 [narendra@ubuntu]$ valgrind –leak-check=full ./free
 ==1238== LEAK SUMMARY:
 ==1238==    definitely lost: 10 bytes in 1 blocks.
 ==1238==      possibly lost: 0 bytes in 0 blocks.
 ==1238==    still reachable: 0 bytes in 0 blocks.
 ==1238==         suppressed: 0 bytes in 0 blocks.
[narendra@ubuntu]$
```

Let us modify the above code.

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

int main(void)
{
   int *ptr = (int*) malloc(10);

   /* we are calling realloc with size = 0 */
   realloc(ptr, 0);


   return 0;
}
```

Check the valgrind's output. It shows no memory leaks are possible, highlighted in red color.

```
[narendra@ubuntu]$ valgrind –leak-check=full ./a.out
==1435== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 11 from 1)
==1435== malloc/free: in use at exit: 0 bytes in 0 blocks.
==1435== malloc/free: 1 allocs, 1 frees, 10 bytes allocated.
==1435== For counts of detected errors, rerun with: -v
==1435== All heap blocks were freed — no leaks are possible.
[narendra@ubuntu]$
```

This article is compiled by "Narendra Kangralkar" and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

---

calloc() versus malloc()

malloc() allocates memory block of given size (in bytes) and returns a pointer to the beginning of the block.

```
 void * malloc( size_t size );
```

malloc() doesn't initialize the allocated memory.

calloc() allocates the memory and also initializes the allocates memory to zero.

```
void * calloc( size_t num, size_t size );
```

Unlike malloc(), calloc() takes two arguments: 1) number of blocks to be allocated 2) size of each block.

We can achieve same functionality as calloc() by using malloc() followed by memset(),

```
ptr = malloc(size);
memset(ptr, 0, size);
```

If we do not want to initialize memory then malloc() is the obvious choice.

Please write comments if you find anything incorrect in the above article or you want to share more information about malloc() and calloc() functions.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

How does free() know the size of memory to be deallocated?

Consider the following prototype of *free()* function which is used to free memory allocated using *malloc()* or *calloc()* or *realloc()*.

```
void free(void *ptr);
```

Note that the free function does not accept size as a parameter. How does free() function know how much memory to free given just a pointer?

Following is the most common way to store size of memory so that free() knows the size of memory to be deallocated.
When memory allocation is done, the actual heap space allocated is one word larger than the requested memory. The extra word is used to store the size of the allocation and is later used by free( )

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:
http://www.cs.cmu.edu/afs/cs/academic/class/15213-f10/www/lectures/17-allocation-basic.pptx
http://en.wikipedia.org/wiki/Malloc

Use of realloc()

Size of dynamically allocated memory can be changed by using realloc().

As per the C99 standard:

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

realloc deallocates the old object pointed to by ptr and returns a pointer to a new object that has the size specified by size. The contents of the new object is identical to that of the old object prior to deallocation, up to the lesser of the new and old sizes. Any bytes in the new object beyond the size of the old object have indeterminate values.

The point to note is that **realloc() should only be used for dynamically allocated memory**. If the memory is not dynamically allocated, then behavior is undefined.
For example, program 1 demonstrates incorrect use of realloc() and program 2 demonstrates correct use of realloc().

Program 1:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int arr[2], i;
   int *ptr = arr;
   int *ptr_new;

   arr[0] = 10;
   arr[1] = 20;

   // incorrect use of new_ptr: undefined behaviour
   ptr_new = (int *)realloc(ptr, sizeof(int)*3);
   *(ptr_new + 2) = 30;

   for(i = 0; i < 3; i++)
     printf("%d ", *(ptr_new + i));

   getchar();
   return 0;
}
```

Output:
Undefined Behavior

Program 2:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int *ptr = (int *)malloc(sizeof(int)*2);
   int i;
   int *ptr_new;

   *ptr = 10;
   *(ptr + 1) = 20;

   ptr_new = (int *)realloc(ptr, sizeof(int)*3);
   *(ptr_new + 2) = 30;
   for(i = 0; i < 3; i++)
       printf("%d ", *(ptr_new + i));

   getchar();
```

```
    return 0;
}
```

Output:

10 20 30

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

What is Memory Leak? How can we avoid?

Memory leak occurs when programmers create a memory in heap and forget to delete it.

Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminate.

```
/* Function with memory leak */
#include <stdlib.h>

void f()
{
   int *ptr = (int *) malloc(sizeof(int));

   /* Do some work */

   return; /* Return without freeing ptr*/
}
```

To avoid memory leaks, memory allocated on heap should always be freed when no longer needed.

```
/* Function without memory leak */
#include <stdlib.h>;

void f()
{
   int *ptr = (int *) malloc(sizeof(int));

   /* Do some work */

   free(ptr);
   return;
}
```

fseek() vs rewind() in C

In C, fseek() should be preferred over rewind().

Note the following text C99 standard:

The rewind function sets the file position indicator for the stream pointed to by stream to the beginning of the file. It is equivalent to

```
    (void)fseek(stream, 0L, SEEK_SET)
```

except that the error indicator for the stream is also cleared.

This following code example sets the file position indicator of an input stream back to the beginning using rewind(). But there is no way to check whether the rewind() was successful.

```
int main()
```

```
{
  FILE *fp = fopen("test.txt", "r");

  if ( fp == NULL ) {
    /* Handle open error */
  }

  /* Do some processing with file*/

  rewind(fp);  /* no way to check if rewind is successful */

  /* Do some more precessing with file */

  return 0;
}
```

In the above code, fseek() can be used instead of rewind() to see if the operation succeeded. Following lines of code can be used in place of rewind(fp);

```
if ( fseek(fp, 0L, SEEK_SET) != 0 ) {
  /* Handle repositioning error */
}
```

Source:
https://www.securecoding.cert.org/confluence/display/seccode/FIO07-C.+Prefer+fseek%28%29+to+rewind%28%29

This article is contributed by **Rahul Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
GFacts

EOF, getc() and feof() in C

In C/C++, getc() returns EOF when end of file is reached. getc() also returns EOF when it fails. So, only comparing the value returned by getc() with EOF is not sufficient to check for actual end of file. To solve this problem, C provides feof() which returns non-zero value only if end of file has reached, otherwise it returns 0.
For example, consider the following C program to print contents of file *test.txt* on screen. In the program, returned value of getc() is compared with EOF first, then there is another check using feof(). By putting this check, we make sure that the program prints *"End of file reached"* only if end of file is reached. And if getc() returns EOF due to any other reason, then the program prints *"Something went wrong"*

```
#include <stdio.h>

int main()
{
  FILE *fp = fopen("test.txt", "r");
  int ch = getc(fp);
  while (ch != EOF)
  {
    /* display contents of file on screen */
    putchar(ch);

    ch = getc(fp);
  }

  if (feof(fp))
     printf("\n End of file reached.");
  else
     printf("\n Something went wrong.");
  fclose(fp);

  getchar();
  return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

fopen() for an existing file in write mode

In C, fopen() is used to open a file in different modes. To open a file in write mode, “w” is specified. When mode “w” is specified, it creates an empty file for output operations.

What if the file already exists?

If a file with the same name already exists, its contents are discarded and the file is treated as a new empty file. For example, in the following program, if “test.txt” already exists, its contents are removed and “GeeksforGeeks” is written to it.

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

int main()
{
    FILE *fp = fopen("test.txt", "w");
    if (fp == NULL)
    {
        puts("Couldn't open file");
        exit(0);
    }
    else
    {
        fputs("GeeksforGeeks", fp);
        puts("Done");
        fclose(fp);
    }
    return 0;
}
```

The above behavior may lead to unexpected results. If programmer’s intention was to create a new file and a file with same name already exists, the existing file’s contents are overwritten.

The latest C standard C11 provides a new mode “x” which is exclusive create-and-open mode. Mode “x” can be used with any “w” specifier, like “wx”, “wbx”. **When x is used with w, fopen() returns NULL if file already exists or could not open.** Following is modified C11 program that doesn’t overwrite an existing file.

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

int main()
{
    FILE *fp = fopen("test.txt", "wx");
    if (fp == NULL)
    {
        puts("Couldn't open file or file already exists");
        exit(0);
    }
    else
    {
        fputs("GeeksforGeeks", fp);
        puts("Done");
        fclose(fp);
    }
    return 0;
}
```

References:

Do not make assumptions about fopen() and file creation
http://en.wikipedia.org/wiki/C11_(C_standard_revision)
http://www.cplusplus.com/reference/cstdio/freopen/

This article is compiled by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

C Program to print numbers from 1 to N without using semicolon?

How to print numbers from 1 to N without using any semicolon in C.

```
#include<stdio.h>
#define N 100

// Add your code here to print numbers from 1
// to N without using any semicolon
```

What code to add in above snippet such that it doesn’t contain semicolon and prints numbers from 1 to N?

We strongly recommend you to minimize your browser and try this yourself first

Method 1 (Recursive)

```
// A recursive C program to print all numbers from 1
// to N without semicoolon
#include<stdio.h>
#define N 10

int main(int num)
{
   if (num <= N && printf("%d ", num) && main(num + 1))
   {
   }
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

See this for complete run. Thanks to Utkarsh Trivedi for suggesting this solution.

Method 2 (Iterative)

```
// An iterative C program to print all numbers from 1
// to N without semicoolon
#include<stdio.h>
#define N 10

int main(int num, char *argv[])
{
while (num <= N && printf("%d ", num) && num++)
{
}
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

See this for complete run. Thanks to Rahul Huria for suggesting this solution.

How do these solutions work?

main() function can receive arguments. The first argument is argument count whose value is 1 if no argument is passed to it. The first argument is always program name.

```
#include<stdio.h>

int main(int num, char *argv[])
{
   printf("num = %d\n", num);
   printf("argv[0] = %s ", argv[0]);
}
```

Output:

```
num = 1
argv[0] = <file_name>
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C

To find sum of two numbers without using any operator

Write a C program to find sum of positive integers without using any operator. Only use of printf() is allowed. No other library function can be used.

Solution

It's a trick question. We can use printf() to find sum of two numbers as printf() returns the number of characters printed. The width field in printf() can be used to find the sum of two numbers. We can use '*' which indicates the minimum width of output. For example, in the statement "printf("%*d", width, num);", the specified 'width' is substituted in place of *, and 'num' is printed within the minimum width specified. If number of digits in 'num' is smaller than the specified 'wodth', the output is padded with blank spaces. If number of digits are more, the output is printed as it is (not truncated). In the following program, add() returns sum of x and y. It prints 2 spaces within the width specified using x and y. So total characters printed is equal to sum of x and y. That is why add() returns x+y.

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

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

Output:

```
     Sum = 7
```

The output is seven spaces followed by "Sum = 7". We can avoid the leading spaces by using carriage return. Thanks to krazyCoder and Sandeep for suggesting this. The following program prints output without any leading spaces.

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

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

Output:

```
Sum = 7
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

How will you show memory representation of C variables?

Write a C program to show memory representation of C variables like int, float, pointer, etc.

Algorithm:

Get the address and size of the variable. Typecast the address to char pointer. Now loop for size of the variable and print the value at the typecasted pointer.

Program:

```
#include <stdio.h>
typedef unsigned char *byte_pointer;

/*show bytes takes byte pointer as an argument
  and prints memory contents from byte_pointer
  to byte_pointer + len */
void show_bytes(byte_pointer start, int len)
{
    int i;
    for (i = 0; i < len; i++)
         printf(" %.2x", start[i]);
    printf("\n");
}

void show_int(int x)
{
    show_bytes((byte_pointer) &x, sizeof(int));
}

void show_float(float x)
{
    show_bytes((byte_pointer) &x, sizeof(float));
}

void show_pointer(void *x)
{
    show_bytes((byte_pointer) &x, sizeof(void *));
}

/* Drover program to test above functions */
int main()
{
   int i = 1;
   float f = 1.0;
   int *p = &i;
   show_float(f);
   show_int(i);
   show_pointer(p);
   show_int(i);
   getchar();
   return 0;
}
```

Condition To Print “HelloWord”

What should be the “condition” so that the following code snippet prints both HelloWorld !

```
  if  "condition"
    printf ("Hello");
  else
    printf("World");
```

Solution:

```
#include<stdio.h>
int main()
{
   if(!printf("Hello"))
      printf("Hello");
   else
      printf("World");
   getchar();
}
```

Explanation: Printf returns the number of character it has printed successfully. So, following solutions will also work

if (printf(“Hello”)

Change/add only one character and print ‘*’ exactly 20 times

In the below code, change/add only one character and print ‘*’ exactly 20 times.

```
int main()
{
   int i, n = 20;
   for (i = 0; i
```

Solutions:

1. Replace i by n in for loop's third expression

```
#include <stdio.h>
int main()
{
   int i, n = 20;
   for (i = 0; i < n; n--)
      printf("*");
   getchar();
   return 0;
}
```

2. Put '-' before i in for loop's second expression

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

```
{
   int i, n = 20;
   for (i = 0; -i < n; i--)
      printf("*");
   getchar();
   return 0;
}
```

3. Replace

```
#include <stdio.h>
int main()
{
   int i, n = 20;
   for (i = 0; i + n; i--)
     printf("*");
   getchar();
   return 0;
}
```

Let's extend the problem little.

Change/add only one character and print '*' exactly 21 times.

Solution: Put negation operator before i in for loop's second expression.

Explanation: Negation operator converts the number into its one's complement.

```
     No.           One's complement
 0 (00000..00)          -1 (1111..11)
-1 (11..1111)          0 (00..0000)
-2 (11..1110)          1 (00..0001)
-3 (11..1101)          2 (00..0010)
..............................................
-20 (11..01100)         19 (00..10011)
```

```
#include <stdio.h>
int main()
{
   int i, n = 20;
   for (i = 0; ~i < n; i--)
      printf("*");
   getchar();
   return 0;
}
```

Please comment if you find more solutions of above problems.

How can we sum the digits of a given number in single statement?

We strongly recommend that you click here and practice it, before moving on to the solution.

Below are the solutions to get sum of the digits.

1. Iterative:

The function has three lines instead of one line but it calculates sum in line. It can be made one line function if we pass pointer to sum.

```
# include<stdio.h>
int main()
{
  int n = 687;
  printf(" %d ", getSum(n));

  getchar();
  return 0;
}

/* Function to get sum of digits */
int getSum(int n)
{
    int sum;
    /*Single line that calculates sum*/
    for(sum=0; n > 0; sum+=n%10,n/=10);
    return sum;
}
```

2. Recursive

Thanks to ayesha for providing the below recursive solution.

```
int sumDigits(int no)
{
  return no == 0 ? 0 : no%10 + sumDigits(no/10) ;
}

int main(void)
{
  printf("%d", sumDigits(1352));
  getchar();
  return 0;
}
```

Please write comments if you find the above codes/algorithms incorrect, or find better ways to solve the same problem.

C/C++ Puzzles
number-digits

What is the best way in C to convert a number to a string?

Solution: Use sprintf() function.

```
#include<stdio.h>
int main()
{
   char result[50];
   float num = 23.34;
   sprintf(result, "%f", num);
   printf("\n The string for the num is %s", result);
   getchar();
}
```

You can also write your own function using ASCII values of numbers.

C/C++ Puzzles
c puzzle
puzzle

Calculate Logn in one line

Write a one line C function that calculates and returns . For example, if n = 64, then your function should return 6, and if n = 129, then your function should return 7.

Following is a solution using recursion.

```
#include<stdio.h>

unsigned int Log2n(unsigned int n)
{
  return (n > 1)? 1 + Log2n(n/2): 0;
}

int main()
{
 unsigned int n = 32;
 printf("%u", Log2n(n));
 getchar();
 return 0;
}
```

Let us try an extended verision of the problem. Write a one line function Logn(n ,r) which returns . Following is the solution for the extended problem.

```
#include<stdio.h>

unsigned int Logn(unsigned int n, unsigned int r)
{
  return (n > r-1)? 1 + Logn(n/r, r): 0;
}

int main()
{
 unsigned int n = 256;
 unsigned int r = 4;
 printf("%u", Logn(n, r));
 getchar();
 return 0;
}
```

Please write comments if you find any of the above codes incorrect, or find other ways to solve the same problem.

Print “Even” or “Odd” without using conditional statement

Write a C/C++ program that accepts a number from the user and prints “Even” if the entered number is even and prints “Odd” if the number is odd. Your are not allowed to use any comparison (==, ..etc) or conditional (if, else, switch, ternary operator,..etc) statement.

Method 1

Below is a tricky code can be used to print “Even” or “Odd” accordingly.

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

using namespace std;

int main()
{
  char arr[2][5] = {"Even", "Odd"};
  int no;
  cout << "Enter a number: ";
  cin >> no;
  cout << arr[no%2];
  getch();
  return 0;
}
```

Method 2

Below is another tricky code can be used to print “Even” or “Odd” accordingly. Thanks to student for suggesting this method.

```
#include<stdio.h>
int main()
{
    int no;
    printf("Enter a no: ");
    scanf("%d", &no);
    (no & 1 && printf("odd"))|| printf("even");
    return 0;
}
```

Please write comments if you find the above code incorrect, or find better ways to solve the same problem

How will you print numbers from 1 to 100 without using loop?

If we take a look at this problem carefully, we can see that the idea of “loop” is to track some counter value e.g. “i=0” till “i

Well, one possibility is the use of ‘recursion’ provided we use the terminating condition carefully. Here is a solution that prints numbers using recursion.

```
#include <stdio.h>

/* Prints numbers from 1 to n */
void printNos(unsigned int n)
{
  if(n > 0)
  {
    printNos(n-1);
    printf("%d ",  n);
  }
  return;
}

/* Driver program to test printNos */
int main()
{
  printNos(100);
  getchar();
  return 0;
}
```

Time Complexity: O(n)

Now try writing a program that does the same but without any “if” construct.
Hint — use some operator which can be used instead of “if”.

Please note that recursion technique is good but every call to the function creates one “stack-frame” in program stack. So if there’s constraint to the limited memory and we need to print large set of numbers, “recursion” might not be a good idea. So what could be the other alternative?

Another alternative is “goto” statement. Though use of “goto” is not suggestible as a general programming practice as “goto” statement changes the normal program execution sequence yet in some cases, use of “goto” is the best working solution.

So please give a try printing numbers from 1 to 100 with “goto” statement. You can use GfG IDE!

Print 1 to 100 in C++, without loop and recursion

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

How can we sum the digits of a given number in single statement?

We strongly recommend that you click here and practice it, before moving on to the solution.

Below are the solutions to get sum of the digits.

1. Iterative:
The function has three lines instead of one line but it calculates sum in line. It can be made one line function if we pass pointer to sum.

```
# include<stdio.h>
int main()
{
  int n = 687;
  printf(" %d ", getSum(n));

  getchar();
  return 0;
}

/* Function to get sum of digits */
int getSum(int n)
{
    int sum;
    /*Single line that calculates sum*/
    for(sum=0; n > 0; sum+=n%10,n/=10);
    return sum;
}
```

2. Recursive

Thanks to ayesha for providing the below recursive solution.

```
int sumDigits(int no)
{
  return no == 0 ? 0 : no%10 + sumDigits(no/10) ;
}

int main(void)
{
  printf("%d", sumDigits(1352));
  getchar();
  return 0;
}
```

Please write comments if you find the above codes/algorithms incorrect, or find better ways to solve the same problem.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
number-digits

Write a C program to print “Geeks for Geeks” without using a semicolon

Use printf statement inside the if condition

```
#include<stdio.h>
int main()
{
    if( printf( "Geeks for Geeks" ) )
    {    }
}
```

One trivial extension of the above problem: Write a C program to print “;” without using a semicolon

```
#include<stdio.h>
int main()
{
   if(printf("%c",59))
   {
   }
}
```

Please comment if you know more solutions to this problem.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

Write a one line C function to round floating point numbers

Algorithm: roundNo(num)

1. If num is positive then add 0.5.
2. Else subtract 0.5.
3. Type cast the result to int and return.

Example:

num = 1.67, (int) num + 0.5 = (int)2.17 = 2

num = -1.67, (int) num – 0.5 = -(int)2.17 = -2

Implementation:

```
/* Program for rounding floating point numbers */
```

```
# include<stdio.h>

int roundNo(float num)
{
   return num < 0 ? num - 0.5 : num + 0.5;
}

int main()
{
   printf("%d", roundNo(-1.777));
   getchar();
   return 0;
}
```

Output: -2

Time complexity: O(1)
Space complexity: O(1)

Now try rounding for a given precision. i.e., if given precision is 2 then function should return 1.63 for 1.63322 and -1.63 for 1.6332.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
c puzzle
puzzle

Implement Your Own sizeof

Here is an implementation.

```
#define my_sizeof(type) (char *)(&type+1)-(char*)(&type)
int main()
{
   double x;
   printf("%d", my_sizeof(x));
   getchar();
   return 0;
}
```

You can also implement using function instead of macro, but function implementation cannot be done in C as C doesn't support function overloading and sizeof() is supposed to receive parameters of all data types.

Note that above implementation assumes that size of character is one byte.

Time Complexity: O(1)
Space Complexity: O(1)

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
c puzzle
puzzle

How to count set bits in a floating point number in C?

Given a floating point number, write a function to count set bits in its binary representation.

For example, floating point representation of 0.15625 has 6 set bits (See this). A typical C compiler uses single precision floating point format.

We can use the idea discussed here. The idea is to take address of the given floating point number in a pointer variable, typecast the pointer to char * type and process individual bytes one by one. We can easily count set bits in a char using the techniques discussed here.

Following is C implementation of the above idea.

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

```
// A utility function to count set bits in a char.
// Refer http://goo.gl/eHF6Y8 for details of this function.
unsigned int countSetBitsChar(char n)
{
    unsigned int count = 0;
    while (n)
    {
      n &= (n-1);
      count++;
    }
    return count;
}

// Returns set bits in binary representation of x
unsigned int countSetBitsFloat(float x)
{
    // Count number of chars (or bytes) in binary representation of float
    unsigned int n = sizeof(float)/sizeof(char);

    // typcast address of x to a char pointer
    char *ptr = (char *)&x;

    int count = 0;   // To store the result
    for (int i = 0; i < n; i++)
    {
         count += countSetBitsChar(*ptr);
         ptr++;
    }
    return count;
}

// Driver program to test above function
int main()
{
    float x = 0.15625;
    printf ("Binary representation of %f has %u set bits ", x,
             countSetBitsFloat(x));
    return 0;
}
```

Output:

```
Binary representation of 0.156250 has 6 set bits
```

This article is contrbuted by **Vineet Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-puzzle

How to change the output of printf() in main() ?

Consider the following program. Change the program so that the output of printf is always 10. It is not allowed to change main(). Only fun() can be changed.

```
void fun()
{
    // Add something here so that the printf in main prints 10
}

int main()
{
    int i = 10;
    fun();
    i = 20;
    printf("%d", i);
    return 0;
}
```

We can use Macro Arguments to change the output of printf.

```
#include <stdio.h>

void fun()
{
  #define printf(x, y) printf(x, 10);
}

int main()
{
    int i = 10;
    fun();
    i = 20;
    printf("%d", i);
    return 0;
}
```

Output:

```
10
```

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-input-output
cpp-puzzle

How to find length of a string without string.h and loop in C?

Find the length of a string without using any loops and string.h in C. Your program is supposed to behave in following way:

```
Enter a string: GeeksforGeeks (Say user enters GeeksforGeeks)
Entered string is: GeeksforGeeks
Length is: 13
```

You may assume that the length of entered string is always less than 100.

The following is solution.

```
#include <stdio.h>

int main()
{
     char str[100];
     printf("Enter a string: ");
     printf( "\rLength is: %d",
             printf("Entered string is: %s\n", gets(str)) - 20
          );

     return 0;
}
```

Output:

```
Enter a string: GeeksforGeeks
Entered string is: GeeksforGeeks
Length is: 13
```

The idea is to use return values of printf() and gets().
gets() returns the enereed string.

printf() returns the number of characters successfully written on output.

In the above program, gets() returns the entered string. We print the length using the first printf. The second printf() calls gets() and prints the entered string using returned value of gets(), it also prints 20 extra characters for printing “Entered string is: ” and “\n”. That is why we subtract 20 from the returned value of second printf and get the length.

This article is contributed by **Umesh Sharma**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-string

Implement your own itoa()

itoa function converts integer into null-terminated string. It can convert negative numbers too. The standard definition of itoa function is give below:-

```
char* itoa(int num, char* buffer, int base)
```

The third parameter base specify the conversion base. For example:- if base is 2, then it will convert the integer into its binary compatible string or if base is 16, then it will create hexadecimal converted string form of integer number.

If base is 10 and value is negative, the resulting string is preceded with a minus sign (-). With any other base, value is always considered unsigned.

Reference: http://www.cplusplus.com/reference/cstdlib/itoa/?kw=itoa

Examples:

```
itoa(1567, str, 10) should return string "1567"
itoa(-1567, str, 10) should return string "-1567"
itoa(1567, str, 2) should return string "11000011111"
itoa(1567, str, 16) should return string "61f"
```

Individual digits of the given number must be processed and their corresponding characters must be put in the given string. Using repeated division by the given base, we get individual digits from least significant to most significant digit. But in the output, these digits are needed in reverse order. Therefore, we reverse the string obtained after repeated division and return it.

```
/* A C++ program to implement itoa() */
#include <iostream>
using namespace std;

/* A utility function to reverse a string  */
void reverse(char str[], int length)
{
    int start = 0;
    int end = length -1;
    while (start < end)
    {
        swap(*(str+start), *(str+end));
        start++;
        end--;
    }
}

// Implementation of itoa()
char* itoa(int num, char* str, int base)
{
    int i = 0;
    bool isNegative = false;

    /* Handle 0 explicitely, otherwise empty string is printed for 0 */
    if (num == 0)
    {
        str[i++] = '0';
        str[i] = '\0';
        return str;
    }

    // In standard itoa(), negative numbers are handled only with
    // base 10. Otherwise numbers are considered unsigned.
    if (num < 0 && base == 10)
    {
        isNegative = true;
        num = -num;
    }

    // Process individual digits
    while (num != 0)
    {
        int rem = num % base;
        str[i++] = (rem > 9)? (rem-10) + 'a' : rem + '0';
        num = num/base;
    }

    // If number is negative, append '-'
    if (isNegative)
        str[i++] = '-';

    str[i] = '\0'; // Append string terminator

    // Reverse the string
    reverse(str, i);

    return str;
}

// Driver program to test implementation of itoa()
int main()
{
    char str[100];
    cout << "Base:10 " << itoa(1567, str, 10) << endl;
    cout << "Base:10 " << itoa(-1567, str, 10) << endl;
    cout << "Base:2 " << itoa(1567, str, 2) << endl;
    cout << "Base:8 " << itoa(1567, str, 8) << endl;
    cout << "Base:16 " << itoa(1567, str, 16) << endl;
    return 0;
}
```

Output:

```
Base:10 1567
Base:10 -1567
Base:2 11000011111
Base:8 3037
Base:16 61f
```

This article is contributed by **Neha Mahajan**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-library

Write a C program that does not terminate when Ctrl+C is pressed

Write a C program that doesn't terminate when Ctrl+C is pressed. It prints a message "Cannot be terminated using Ctrl+c" and continues execution.

We can use signal handling in C for this. When Ctrl+C is pressed, SIGINT signal is generated, we can catch this signal and run our defined signal handler. C standard defines following 6 signals in signal.h header file.

SIGABRT – abnormal termination.

SIGFPE – floating point exception.

SIGILL – invalid instruction.

SIGINT – interactive attention request sent to the program.

SIGSEGV – invalid memory access.

SIGTERM – termination request sent to the program.

Additional signals are specified Unix and Unix-like operating systems (such as Linux) defines more than 15 additional signals. See http://en.wikipedia.org/wiki/Unix_signal#POSIX_signals

The standard C library function signal() can be used to set up a handler for any of the above signals.

```
/* A C program that does not terminate when Ctrl+C is pressed */
#include <stdio.h>
#include <signal.h>

/* Signal Handler for SIGINT */
void sigintHandler(int sig_num)
{
   /* Reset handler to catch SIGINT next time.
     Refer http://en.cppreference.com/w/c/program/signal */
   signal(SIGINT, sigintHandler);
   printf("\n Cannot be terminated using Ctrl+C \n");
   fflush(stdout);
}

int main ()
{
   /* Set the SIGINT (Ctrl-C) signal handler to sigintHandler
     Refer http://en.cppreference.com/w/c/program/signal */
   signal(SIGINT, sigintHandler);

   /* Infinite loop */
   while(1)
   {
   }
   return 0;
}
```

Ouput: When Ctrl+C was pressed two times

```

 Cannot be terminated using Ctrl+C

 Cannot be terminated using Ctrl+C
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-puzzle

How to measure time taken by a function in C?

To calculate time taken by a process, we can use clock() function which is available *time.h*. We can call the clock function at the beginning and end of the code for which we measure time, subtract the values, and then divide by CLOCKS_PER_SEC (the number of clock ticks per second) to get processor time, like following.

```
    #include <time.h>

    clock_t start, end;
    double cpu_time_used;

    start = clock();
    ... /* Do the work. */
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

Following is a sample C program where we measure time taken by fun(). The function fun() waits for enter key press to terminate.

```
/* Program to demonstrate time taken by function fun() */
#include <stdio.h>
#include <time.h>

// A function that terminates when enter key is pressed
void fun()
{
   printf("fun() starts \n");
   printf("Press enter to stop fun \n");
   while(1)
   {
      if (getchar())
         break;
   }
   printf("fun() ends \n");
}

// The main program calls fun() and measures time taken by fun()
int main()
{
   // Calculate the time taken by fun()
   clock_t t;
   t = clock();
   fun();
   t = clock() - t;
   double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds

   printf("fun() took %f seconds to execute \n", time_taken);
   return 0;
}
```

Output: The following output is obtained after waiting for around 4 seconds and then hitting enter key.

```
fun() starts
Press enter to stop fun

fun() ends
fun() took 4.017000 seconds to execute
```

How to find time taken by a command/program on Linux Shell?

References:

http://www.gnu.org/software/libc/manual/html_node/CPU-Time.html

http://www.cplusplus.com/reference/ctime/clock/?kw=clock

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Print a long int in C using putchar() only

Write a C function *print(n)* that takes a long int number *n* as argument, and prints it on console. The only allowed library function is *putchar()*, no other function like *itoa()* or *printf()* is allowed. Use of loops is also not allowed.

We strongly recommend to minimize the browser and try this yourself first.

This is a simple trick question. Since putchar() prints a character, we need to call putchar() for all digits. Recursion can always be used to replace iteration, so using recursion we can print all digits one by one. Now the question is putchar() prints a character, how to print digits using putchar(). We need to convert every digit to its corresponding ASCII value, this can be done by using ASCII value of ‘0’. Following is complete C program.

```
/* C program to print a long int number using putchar() only*/
```

```
#include <stdio.h>
void print(long n)
{
    // If number is smaller than 0, put a - sign and
    // change number to positive
    if (n < 0) {
        putchar('-');
        n = -n;
    }

    // If number is 0
    if (n == 0)
        putchar('0');

    // Remove the last digit and recur
    if (n/10)
        print(n/10);

    // Print the last digit
    putchar(n%10 + '0');
}

// Driver program to test abvoe function
int main()
{
    long int n = 12045;
    print(n);
    return 0;
}
```

Output:

```
12045
```

One important thing to note is the sequence of putchar() and recursive call print(n/10). Since the digits should be printed left to right, the recursive call must appear before putchar() (The rightmost digit should be printed at the end, all other digits must be printed before it).

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-library
cpp-puzzle

Convert a floating point number to string in C

Write a C function ftoa() that converts a given floating point number to string. Use of standard library functions for direct conversion is not allowed. The following is prototype of ftoa().

```
ftoa(n, res, afterpoint)
n          --> Input Number
res[]      --> Array where output string to be stored
afterpoint --> Number of digits to be considered after point.

For example ftoa(1.555, str, 2) should store "1.55" in res and
ftoa(1.555, str, 0) should store "1" in res.
```

We strongly recommend to minimize the browser and try this yourself first.

A simple way is to use sprintf(), but use of standard library functions for direct conversion is not allowed.
The idea is to separate integral and fractional parts and convert them to strings separately. Following are the detailed steps.
1) Extract integer part from floating point.
2) First convert integer part to string.
3) Extract fraction part by exacted integer part from n.

4) If d is non-zero, then do following.
….a) Convert fraction part to integer by multiplying it with pow(10, d)
….b) Convert the integer value to string and append to the result.

Following is C implementation of the above approach.

```
// C program for implementation of ftoa()
#include<stdio.h>
#include<math.h>

// reverses a string 'str' of length 'len'
void reverse(char *str, int len)
{
    int i=0, j=len-1, temp;
    while (i<j)
    {
        temp = str[i];
        str[i] = str[j];
        str[j] = temp;
        i++; j--;
    }
}

 // Converts a given integer x to string str[].  d is the number
 // of digits required in output. If d is more than the number
 // of digits in x, then 0s are added at the beginning.
int intToStr(int x, char str[], int d)
{
    int i = 0;
    while (x)
    {
        str[i++] = (x%10) + '0';
        x = x/10;
    }

    // If number of digits required is more, then
    // add 0s at the beginning
    while (i < d)
        str[i++] = '0';

    reverse(str, i);
    str[i] = '\0';
    return i;
}

// Converts a floating point number to string.
void ftoa(float n, char *res, int afterpoint)
{
    // Extract integer part
    int ipart = (int)n;

    // Extract floating part
    float fpart = n - (float)ipart;

    // convert integer part to string
    int i = intToStr(ipart, res, 0);

    // check for display option after point
    if (afterpoint != 0)
    {
        res[i] = '.';  // add dot

        // Get the value of fraction part upto given no.
        // of points after dot. The third parameter is needed
        // to handle cases like 233.007
        fpart = fpart * pow(10, afterpoint);

        intToStr((int)fpart, res + i + 1, afterpoint);
    }
}

// driver program to test above funtion
```

```
int main()
{
   char res[20];
   float n = 233.007;
   ftoa(n, res, 4);
   printf("\n\"%s\"\n", res);
   return 0;
}
```

Output:

```
"233.0070"
```

This article is contributed by Jayasantosh Samal. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-data-types
cpp-string

How to write a running C code without main()?

Write a C code that prints “GeeksforGeeks” without any main function. This is a trick question and can be solved in following ways.

1) Using a macro that defines main

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

Output:

```
Geeksforgeeks
```

2) Using Token-Pasting Operator
The above solution has word ‘main’ in it. If we are not allowed to even write main, we ca use token-pasting operator (see this for details)

```
#include<stdio.h>
#define fun m##a##i##n
int fun()
{
   printf("Geeksforgeeks");
   return 0;
}
```

Output:

```
Geeksforgeeks
```

References:
Macros and Preprocessors in C

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-puzzle

Write your own memcpy() and memmove()

The memcpy function is used to copy a block of data from a source address to a destination address. Below is its prototype.

```
void * memcpy(void * destination, const void * source, size_t num);
```

The idea is to simply typecast given addresses to char *(char takes 1 byte). Then one by one copy data from source to destination. Below is implementation of this idea.

```
// A C implementation of memcpy()
#include<stdio.h>
#include<string.h>

void myMemCpy(void *dest, void *src, size_t n)
{
   // Typecast src and dest addresses to (char *)
   char *csrc = (char *)src;
   char *cdest = (char *)dest;

   // Copy contents of src[] to dest[]
   for (int i=0; i<n; i++)
       cdest[i] = csrc[i];
}

// Driver program
int main()
{
   char csrc[] = "GeeksforGeeks";
   char cdest[100];
   myMemCpy(cdest, csrc, strlen(csrc)+1);
   printf("Copied string is %s", cdest);

   int isrc[] = {10, 20, 30, 40, 50};
   int n = sizeof(isrc)/sizeof(isrc[0]);
   int idest[n], i;
   myMemCpy(idest, isrc,  sizeof(isrc));
   printf("\nCopied array is ");
   for (i=0; i<n; i++)
     printf("%d ", idest[i]);
   return 0;
}
```

Output:

```
Copied string is GeeksforGeeks
Copied array is 10 20 30 40 50
```

What is memmove()?

memmove() is similar to memcpy() as it also copies data from a source to destination. memcpy() leads to problems when source and destination addresses overlap as memcpy() simply copies data one by one from one location to another. For example consider below program.

```
// Sample program to show that memcpy() can loose data.
#include <stdio.h>
#include <string.h>
int main()
{
   char csrc[100] = "Geeksfor";
   memcpy(csrc+5, csrc, strlen(csrc)+1);
   printf("%s", csrc);
   return 0;
}
```

Output:

```
GeeksGeeksGeek
```

Since the input addresses are overlapping, the above program overwrites the original string and causes data loss.

```
// Sample program to show that memmove() is better than memcpy()
// when addresses overlap.
#include <stdio.h>
#include <string.h>
int main()
{
  char csrc[100] = "Geeksfor";
  memmove(csrc+5, csrc, strlen(csrc)+1);
  printf("%s", csrc);
  return 0;
}
```

Output:

```
GeeksGeeksfor
```

How to implement memmove()?

The trick here is to use a temp array instead of directly copying from src to dest. The use of temp array is important to handle cases when source and destination addresses are overlapping.

```
//C++ program to demonstrate implementation of memmove()
#include<stdio.h>
#include<string.h>

// A function to copy block of 'n' bytes from source
// address 'src' to destination address 'dest'.
void myMemMove(void *dest, void *src, size_t n)
{
  // Typecast src and dest addresses to (char *)
  char *csrc = (char *)src;
  char *cdest = (char *)dest;

  // Create a temporary array to hold data of src
  char *temp = new char[n];

  // Copy data from csrc[] to temp[]
  for (int i=0; i<n; i++)
     temp[i] = csrc[i];

  // Copy data from temp[] to cdest[]
  for (int i=0; i<n; i++)
     cdest[i] = temp[i];

  delete [] temp;
}

// Driver program
int main()
{
  char csrc[100] = "Geeksfor";
  myMemMove(csrc+5, csrc, strlen(csrc)+1);
  printf("%s", csrc);
  return 0;
}
```

Output:

```
GeeksGeeksfor
```

Optimizations:
The algorithm is inefficient (and honestly double the time if you use a temporary array). Double copies should be avoided unless if it is really impossible.

In this case though it is easily possible to avoid double copies by picking a direction of copy. In fact this is what the memmove() library function does.

By comparing the src and the dst addresses you should be able to find if they overlap.

– If they do not overlap, you can copy in any direction
– If they do overlap, find which end of dest overlaps with the source and choose the direction of copy accordingly.
– If the beginning of dest overlaps, copy from end to beginning
– If the end of dest overlaps, copy from beginning to end
– Another optimization would be to copy by word size. Just be careful to handle the boundary conditions.
– A further optimization would be to use vector instructions for the copy since they're contiguous.

This article is contributed by Saurabh Jain. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-library
cpp-pointer

Quine – A self-reproducing program

A quine is a program which prints a copy of its own as the only output. A quine takes no input. Quines are named after the American mathematician and logician Willard Van Orman Quine (1908–2000). The interesting thing is you are not allowed to use open and then print file of the program.

To the best of our knowledge, below is the shortest quine in C.

```
main() { char *s="main() { char *s=%c%s%c; printf(s,34,s,34); }"; printf(s,34,s,34); }
```

This program uses the printf function without including its corresponding header (#include), which can result in undefined behavior. Also, the return type declaration for main has been left off to reduce the length of the program. Two 34s are used to print double quotes around the string s.

Following is a shorter version of the above program suggested by Narendra.

```
main(a){printf(a="main(a){printf(a=%c%s%c,34,a,34);}",34,a,34);}
```

If you find a shorter C quine or you want to share quine in other programming languages, then please do write in the comment section.

Quine in Python

Source:
http://en.wikipedia.org/wiki/Quine_%28computing%29

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

Complicated declarations in C

Most of the times declarations are simple to read, but it is hard to read some declarations which involve pointer to functions. For example, consider the following declaration from "signal.h".

```
void (*bsd_signal(int, void (*)(int)))(int);
```

Let us see the steps to read complicated declarations.

1) Convert C declaration to postfix format and read from left to right.
2) To convert experssion to postfix, start from innermost parenthesis, If innermost parenthesis is not present then start from declarations name and go right first. When first ending parenthesis encounters then go left. Once whole parenthesis is parsed then come out from parenthesis.
3) Continue until complete declaration has been parsed.

Let us start with simple example. Below examples are from “K & R” book.

```
1)  int (*fp) ();
```

Let us convert above expression to postfix format. For the above example, there is no innermost parenthesis, that’s why, we will print declaration name i.e. “fp”. Next step is, go to right side of expression, but there is nothing on right side of “fp” to parse, that’s why go to left side. On left side we found “*”, now print “*” and come out of parenthesis. We will get postfix expression as below.

```
  fp  *  ()  int
```

Now read postfix expression from left to right. e.g. fp is pointer to function returning int

Let us see some more examples.

```
2) int (*daytab)[13]
```

Postfix : daytab * [13] int
Meaning : daytab is pointer to array of 13 integers.

```
3) void (*f[10]) (int, int)
```

Postfix : f[10] * (int, int) void
Meaning : f is an array of 10 of pointer to function(which takes 2 arguments of type int) returning void

```
4) char (*(*x())[]) ()
```

Postfix : x () * [] * () char
Meaning : x is a function returning pointer to array of pointers to function returnging char

```
5) char (*(*x[3])())[5]
```

Postfix : x[3] * () * [5] char
Meaning : x is an array of 3 pointers to function returning pointer to array of 5 char’s

```
6) int *(*(*arr[5])()) ()
```

Postfix : arr[5] * () * () * int
Meaning : arr is an array of 5 pointers to functions returning pointer to function returning pointer to integer

```
7) void (*bsd_signal(int sig, void (*func)(int)))(int);
```

Postfix : bsd_signal(int sig, void(*func)(int)) * (int) void
Meaning : bsd_signal is a function that takes integer & a pointer to a function(that takes integer as argument and returns void) and returns pointer to a function(that take integer as argument and returns void)

This article is compiled by “Narendra Kangralkar“ and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
pointer

Use of bool in C

The C99 standard for C language supports bool variables. Unlike C++, where no header file is needed to use bool, a header file “stdbool.h” must be included to use bool in C. If we save the below program as .c, it will not compile, but if we save it as .cpp, it will work fine.

```
int main()
{
  bool arr[2] = {true, false};
  return 0;
}
```

If we include the header file "stdbool.h" in the above program, it will work fine as a C program.

```
#include <stdbool.h>
int main()
{
  bool arr[2] = {true, false};
  return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles

Sequence Points in C | Set 1

In this post, we will try to cover many ambiguous questions like following.

Guess the output of following programs.

```
// PROGRAM 1
#include <stdio.h>
int f1() { printf ("Geeks"); return 1;}
int f2() { printf ("forGeeks"); return 1;}
int main()
{
  int p = f1() + f2();
  return 0;
}

// PROGRAM 2
#include <stdio.h>
int x = 20;
int f1() { x = x+10; return x;}
int f2() { x = x-5;  return x;}
int main()
{
  int p = f1() + f2();
  printf ("p = %d", p);
  return 0;
}


// PROGRAM 3
#include <stdio.h>
int main()
{
   int i = 8;
   int p = i++*i++;
   printf("%d\n", p);
}
```

The output of all of the above programs is undefined or unspecified. The output may be different with different compilers and different machines. It is like asking the value of undefined automatic variable.

The reason for undefined behavior in PROGRAM 1 is, the operator '+' doesn't have standard defined order of evaluation for its operands. Either f1() or f2() may be executed first. So output may be either "GeeksforGeeks" or "forGeeksGeeks".
Similar to operator '+', most of the other similar operators like '-', '/', '*', Bitwise AND &, Bitwise OR |, .. etc don't have a standard defined order for evaluation for its operands.

Evaluation of an expression may also produce side effects. For example, in the above program 2, the final values of p is ambiguous. Depending on the order of expression evaluation, if f1() executes first, the value of p will be 55, otherwise 40.

The output of program 3 is also undefined. It may be 64, 72, or may be something else. The subexpression i++ causes a side effect, it modifies i's value, which leads to undefined behavior since i is also referenced elsewhere in the same expression.

Unlike above cases, *at certain specified points in the execution sequence called sequence points, all side effects of previous evaluations are guaranteed to be complete*. A sequence point defines any point in a computer program's execution at which it is guaranteed that all side effects of previous evaluations will have been performed, and no side effects from subsequent evaluations have yet been performed. Following are the sequence points listed in the C standard:

— The end of the first operand of the following operators:

a) logical AND &&

b) logical OR ||

c) conditional ?

d) comma ,

For example, the output of following programs is guaranteed to be "GeeksforGeeks" on all compilers/machines.

```
// Following 3 lines are common in all of the below programs
#include <stdio.h>
int f1() { printf ("Geeks"); return 1;}
int f2() { printf ("forGeeks"); return 1;}

// PROGRAM 4
int main()
{
   // Since && defines a sequence point after first operand, it is
   // guaranteed that f1() is completed first.
   int p = f1() && f2();
   return 0;
}

// PROGRAM 5
int main()
{
   // Since comma operator defines a sequence point after first operand, it is
   // guaranteed that f1() is completed first.
  int p = (f1(), f2());
  return 0;
}


// PROGRAM 6
int main()
{
   // Since ? operator defines a sequence point after first operand, it is
   // guaranteed that f1() is completed first.
  int p = f1()? f2(): 3;
  return 0;
}
```

— The end of a full expression. This category includes following expression statements

a) Any full statement ended with semicolon like "a = b;"

b) return statements

c) The controlling expressions of if, switch, while, or do-while statements.

d) All three expressions in a for statement.

The above list of sequence points is partial. We will be covering all remaining sequence points in the next post on Sequence Point. We will also be covering many C questions asked in forum (See this, this, this , this and many others).

References:

http://en.wikipedia.org/wiki/Sequence_point

http://c-faq.com/expr/seqpoints.html

http://msdn.microsoft.com/en-us/library/d45c7a5d(v=vs.110).aspx

http://www.open-std.org/jtc1/sc22/wg14/www/docs/n925.htm

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Optimization Techniques | Set 2 (swapping)

How to swap two variables?

The question may look silly, neither geeky. See the following piece of code to swap two integers (XOR swapping),

```
void myswap(int *x, int *y)
{
  if (x != y)
  {
    *x^=*y^=*x^=*y;
  }
}
```

At first glance, we may think nothing wrong with the code. However, when prompted for reason behind opting for XOR swap logic, the person was clue less. Perhaps any ***commutative*** operation can fulfill the need with some corner cases.

Avoid using fancy code snippets in production software. They create runtime surprises. We can observe the following notes on above code

1. The code behavior is undefined. The statement *x^=*y^=*x^=*y; modifying a variable more than once in without any sequence point.
2. It creates pipeline stalls when executed on a processor with pipeline architecture.
3. The compiler can't take advantage in optimizing the swapping operation. Some processors will provide single instruction to swap two variables. When we opted for standard library functions, there are more chances that the library would have been optimized. Even the compiler can recognize such standard function and generates optimum code.
4. Code readability is poor. It is very much important to write maintainable code.

Thanks to **Venki** for writing the above article. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
XOR
About Venki
Software Engineer
View all posts by Venki →

ASCII NUL, ASCII 0 ('0') and Numeric literal 0

The ASCII NUL and zero are represented as 0x00 and 0x30 respectively. An ASCII NUL character serves as sentinel characters of strings in C/C++. When the programmer uses '0' in his code, it will be represented as 0x30 in hex form. What will be filled in the binary representation of 'integer' in the following program?

```
char charNUL = '\0';

unsigned int integer = 0;

char charBinary = '0';
```

The binary form of *charNUL* will have all its bits set to logic 0. The binary form of *integer* will have all its bits set to logic 0, which means each byte will be filled with NUL character (\ 0). The binary form of *charBinary* will be set to binary equivalent of hex 0x30.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
About Venki
Software Engineer
View all posts by Venki →

Little and Big Endian Mystery

What are these?

Little and big endian are two ways of storing multibyte data-types (int, float, etc). In little endian machines, last byte of binary representation of the multibyte data-type is stored first. On the other hand, in big endian machines, first byte of binary representation of the multibyte data-type is stored first.

Suppose integer is stored as 4 bytes (For those who are using DOS based compilers such as C++ 3.0 , integer is 2 bytes) then a variable x with value 0x01234567 will be stored as following.

Memory representation of integer ox01234567 inside Big and little endian machines

How to see memory representation of multibyte data types on your machine?
Here is a sample C code that shows the byte representation of int, float and pointer.

```
#include <stdio.h>

/* function to show bytes in memory, from location start to start+n*/
void show_mem_rep(char *start, int n)
{
    int i;
    for (i = 0; i < n; i++)
         printf(" %.2x", start[i]);
    printf("\n");
}

/*Main function to call above function for 0x01234567*/
int main()
{
   int i = 0x01234567;
   show_mem_rep((char *)&i, sizeof(i));
   getchar();
   return 0;
}
```

When above program is run on little endian machine, gives “67 45 23 01” as output , while if it is run on endian machine, gives “01 23 45 67” as output.

Is there a quick way to determine endianness of your machine?
There are n no. of ways for determining endianness of your machine. Here is one quick way of doing the same.

```
#include <stdio.h>
int main()
{
   unsigned int i = 1;
   char *c = (char*)&i;
   if (*c)
       printf("Little endian");
   else
       printf("Big endian");
   getchar();
   return 0;
}
```

In the above program, a character pointer c is pointing to an integer i. Since size of character is 1 byte when the character pointer is de-referenced it will contain only first byte of integer. If machine is little endian then *c will be 1 (because last byte is stored first) and if machine is big endian then *c will be 0.

Does endianness matter for programmers?
Most of the times compiler takes care of endianness, however, endianness becomes an issue in following cases.

It matters in network programming: Suppose you write integers to file on a little endian machine and you transfer this file to a big endian machine. Unless there is little andian to big endian transformation, big endian machine will read the file in reverse order. You can find such a practical example here.

Standard byte order for networks is big endian, also known as network byte order. Before transferring data on network, data is first converted to network byte order (big endian).

Sometimes it matters when you are using type casting, below program is an example.

```
#include <stdio.h>
int main()
{
   unsigned char arr[2] = {0x01, 0x00};
   unsigned short int x = *(unsigned short int *) arr;
   printf("%d", x);
   getchar();
   return 0;
}
```

In the above program, a char array is typecasted to an unsigned short integer type. When I run above program on little endian machine, I get 1 as output, while if I run it on a big endian machine I get 256. To make programs endianness independent, above programming style should be avoided.

What are bi-endians?
Bi-endian processors can run in both modes little and big endian.

What are the examples of little, big endian and bi-endian machines ?
Intel based processors are little endians. ARM processors were little endians. Current generation ARM processors are bi-endian.

Motorola 68K processors are big endians. PowerPC (by Motorola) and SPARK (by Sun) processors were big endian. Current version of these processors are bi-endians.

Does endianness effects file formats?
File formats which have 1 byte as a basic unit are independent of endianness e..g., ASCII files . Other file formats use some fixed endianness forrmat e.g, JPEG files are stored in big endian format.

Which one is better — little endian or big endian
The term little and big endian came from Gulliver's Travels by Jonathan Swift. Two groups could not agree by which end a egg should be opened -a-the little or the big. Just like the egg issue, there is no technological reason to choose one byte ordering convention over the other, hence the arguments degenerate into bickering about sociopolitical issues. As long as one of the conventions is selected and adhered to consistently, the choice is arbitrary.

GATE CS Corner   Company Wise Coding Practice

Bit Magic
Articles
Big Endian
Bit Magic
Endianness
Little Endian
Tutorial

Comparator function of qsort() in C

Standard C library provides qsort() that can be used for sorting an array. As the name suggests, the function uses QuickSort algorithm to sort the given array. Following is prototype of qsort()

```
void qsort (void* base, size_t num, size_t size,
            int (*comparator)(const void*,const void*));
```

The key point about qsort() is comparator function *comparator.* The comparator function takes two arguments and contains logic to decide their relative order in sorted output. The idea is to provide flexibility so that qsort() can be used for any type (including user defined types) and can be used to obtain any desired order (increasing, decreasing or any other).

The comparator function takes two pointers as arguments (both type-casted to const void*) and defines the order of the elements by returning (in a stable and transitive manner

```
int comparator(const void* p1, const void* p2);
Return value meaning
0 The element pointed by p1 goes after the element pointed by p2

Source: http://www.cplusplus.com/reference/cstdlib/qsort/
```

For example, let there be an array of students where following is type of student.

```
struct Student
{
    int age, marks;
    char name[20];
};
```

Lets say we need to sort the students based on marks in ascending order. The comparator function will look like:

```
int comparator(const void *p, const void *q)
{
    int l = ((struct Student *)p)->marks;
    int r = ((struct Student *)q)->marks;
    return (l - r);
}
```

See following posts for more sample uses of qsort().
Given a sequence of words, print all anagrams together
Box Stacking Problem
Closest Pair of Points

Following is an interesting problem that can be easily solved with the help of *qsort()* and comparator function.
Given an array of integers, sort it in such a way that the odd numbers appear first and the even numbers appear later. The odd numbers should be sorted in descending order and the even numbers should be sorted in ascending order.

The simple approach is to first modify the input array such that the even and odd numbers are segregated followed by applying some sorting algorithm on both parts(odd and even) separately.

However, there exists an interesting approach with a little modification in comparator function of Quick Sort. The idea is to write a comparator function that takes two addresses p and q as arguments. Let l and r be the number pointed by p and q. The function uses following logic:
1) If both (l and r) are odd, put the greater of two first.
2) If both (l and r) are even, put the smaller of two first.
3) If one of them is even and other is odd, put the odd number first.

Following is C implementation of the above approach.

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

// This function is used in qsort to decide the relative order
// of elements at addresses p and q.
int comparator(const void *p, const void *q)
{
    // Get the values at given addresses
    int l = *(const int *)p;
    int r = *(const int *)q;

    // both odd, put the greater of two first.
    if ((l&1) && (r&1))
        return (r-l);

    // both even, put the smaller of two first
    if ( !(l&1) && !(r&1) )
        return (l-r);

    // l is even, put r first
    if (!(l&1))
        return 1;

    // l is odd, put l first
    return -1;
}

// A utility function to print an array
void printArr(int arr[], int n)
{
    int i;
    for (i = 0; i < n; ++i)
        printf("%d ", arr[i]);
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 6, 5, 2, 3, 9, 4, 7, 8};

    int size = sizeof(arr) / sizeof(arr[0]);
    qsort((void*)arr, size, sizeof(arr[0]), comparator);

    printf("Output array is\n");
    printArr(arr, size);

    return 0;
}
```

Output:

```
Output array is
9 7 5 3 1 2 4 6 8
```

Exercise:
Given an array of integers, sort it in alternate fashion. Alternate fashion means that the elements at even indices are sorted separately and elements at odd indices are sorted separately.

This article is compiled by Aashish Barnwal. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
Sorting
Quick Sort
Sorting

Program to validate an IP address

Write a program to Validate an IPv4 Address.

According to Wikipedia, IPv4 addresses are canonically represented in dot-decimal notation, which consists of four decimal numbers, each ranging from 0 to 255, separated by dots, e.g., 172.16.254.1

Following are steps to check whether a given string is valid IPv4 address or not:

step 1) Parse string with "." as delimiter using "strtok()" function.

```
e.g. ptr = strtok(str, DELIM);
```

step 2)
……..a) If ptr contains any character which is not digit then return 0
……..b) Convert "ptr" to decimal number say 'NUM'
……..c) If NUM is not in range of 0-255 return 0
……..d) If NUM is in range of 0-255 and ptr is non-NULL increment "dot_counter" by 1
……..e) if ptr is NULL goto step 3 else goto step 1

step 3) if dot_counter != 3 return 0 else return 1.

```
// Program to check if a given string is valid IPv4 address or not
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define DELIM "."

/* return 1 if string contain only digits, else return 0 */
int valid_digit(char *ip_str)
{
    while (*ip_str) {
        if (*ip_str >= '0' && *ip_str <= '9')
            ++ip_str;
        else
            return 0;
    }
    return 1;
}

/* return 1 if IP string is valid, else return 0 */
int is_valid_ip(char *ip_str)
{
    int i, num, dots = 0;
    char *ptr;

    if (ip_str == NULL)
        return 0;

    // See following link for strtok()
    // http://pubs.opengroup.org/onlinepubs/009695399/functions/strtok_r.html
    ptr = strtok(ip_str, DELIM);

    if (ptr == NULL)
        return 0;

    while (ptr) {

        /* after parsing string, it must contain only digits */
        if (!valid_digit(ptr))
            return 0;

        num = atoi(ptr);

        /* check for valid IP */
        if (num >= 0 && num <= 255) {
            /* parse remaining string */
            ptr = strtok(NULL, DELIM);
            if (ptr != NULL)
                ++dots;
        } else
            return 0;
    }

    /* valid IP string must contain 3 dots */
    if (dots != 3)
        return 0;
    return 1;
}

// Driver program to test above functions
int main()
{
    char ip1[] = "128.0.0.1";
    char ip2[] = "125.16.100.1";
    char ip3[] = "125.512.100.1";
    char ip4[] = "125.512.100.abc";
    is_valid_ip(ip1)? printf("Valid\n"): printf("Not valid\n");
    is_valid_ip(ip2)? printf("Valid\n"): printf("Not valid\n");
    is_valid_ip(ip3)? printf("Valid\n"): printf("Not valid\n");
    is_valid_ip(ip4)? printf("Valid\n"): printf("Not valid\n");
    return 0;
}
```

Output:

```
Valid
Valid
Not valid
Not valid
```

This article is compiled by **Narendra Kangralkar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Multithreading in C

What is a Thread?

A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called *lightweight processes*.

What are the differences between process and thread?

Threads are not independent of one other like processes as a result threads shares with other threads their code section, data section and OS resources like open files and signals. But, like process, a thread has its own program counter (PC), a register set, and a stack space.

Why Multithreading?

Threads are popular way to improve application through parallelism. For example, in a browser, multiple tabs can be different threads. MS word uses multiple threads, one thread to format the text, other thread to process inputs, etc.
Threads operate faster than processes due to following reasons:
1) Thread creation is much faster.
2) Context switching between threads is much faster.
3) Threads can be terminated easily
4) Communication between threads is faster.

See http://www.personal.kent.edu/~rmuhamma/OpSystems/Myos/threads.htm for more details.

Can we write multithreading programs in C?

Unlike Java, multithreading is not supported by the language standard. POSIX Threads (or Pthreads) is a POSIX standard for threads. Implementation of pthread is available with gcc compiler.

A simple C program to demonstrate use of pthread basic functions

Please not that the below program may compile only with C compilers with pthread library.

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

// A normal C function that is executed as a thread when its name
// is specified in pthread_create()
void *myThreadFun(void *vargp)
{
   sleep(1);
   printf("Printing GeeksQuiz from Thread \n");
   return NULL;
}

int main()
{
   pthread_t tid;
   printf("Before Thread\n");
   pthread_create(&tid, NULL, myThreadFun, NULL);
   pthread_join(tid, NULL);
   printf("After Thread\n");
   exit(0);
}
```

In main() we declare a variable called thread_id, which is of type pthread_t, which is an integer used to identify the thread in the system. After declaring thread_id, we call pthread_create() function to create a thread.
pthread_create() takes 4 arguments.
The first argument is a pointer to thread_id which is set by this function.
The third argument is name of function to be executed for the thread to be created.
The fourth argument is used to pass arguments to thread.
The pthread_join() function for threads is the equivalent of wait() for processes. A call to pthread_join blocks the calling thread until the thread with identifier equal to the first argument terminates.

How to compile above program?

To compile a multithreaded program using gcc, we need to link it with the pthreads library. Following is the command used to compile the program.

```
gfg@ubuntu:~/$ gcc multithread.c -lpthread
gfg@ubuntu:~/$ ./a.out
Before Thread
Printing GeeksQuiz from Thread
After Thread
gfg@ubuntu:~/$
```

A C program to show multiple threads with global and static variables

As mentioned above, all threads share data segment. Global and static variables are stored in data segment. Therefore, they are shared by all threads. The following example program demonstrates the same.

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

// Let us create a global variable to change it in threads
int g = 0;

// The function to be executed by all threads
void *myThreadFun(void *vargp)
{
    // Store the value argument passed to this thread
    int myid = (int)vargp;

    // Let us create a static variable to observe its changes
    static int s = 0;

    // Change static and global variables
    ++s; ++g;

    // Print the argument, static and global variables
    printf("Thread ID: %d, Static: %d, Global: %d\n", myid, ++s, ++g);
}

int main()
{
    int i;
    pthread_t tid;

    // Let us create three threads
    for (i = 0; i < 3; i++)
        pthread_create(&tid, NULL, myThreadFun, (void *)i);

    pthread_exit(NULL);
    return 0;
}
```

```
gfg@ubuntu:~/$ gcc multithread.c -lpthread
gfg@ubuntu:~/$ ./a.out
Thread ID: 1, Static: 1, Global: 1
Thread ID: 0, Static: 2, Global: 2
Thread ID: 2, Static: 3, Global: 3
gfg@ubuntu:~/$
```

Please note that above is simple example to show how threads work. Accessing a global variable in a thread is generally a bad idea. What if thread 2 has priority over thread 1 and thread 1 needs to change the variable. In practice, if it is required to access global variable by multiple threads, then they should be accessed using a mutex.

References:

http://www.csc.villanova.edu/~mdamian/threads/posixthreads.html
Computer Systems : A Programmer

This article is contributed by **Rahul Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Assertions in C/C++

Assertions are statements used to test assumptions made by programmer. For example, we may use assertion to check if pointer returned by malloc() is NULL or not.

Following is syntax for assertion.

```
void assert( int expression );
```

If expression evaluates to 0 (false), then the expression, sourcecode filename, and line number are sent to the standard error, and then abort() function is called.

For example, consider the following program.

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

int main()
{
    int x = 7;

    /*  Some big code in between and let's say x
       is accidentally changed to 9  */
    x = 9;

    // Programmer assumes x to be 7 in rest of the code
    assert(x==7);

    /* Rest of the code */

    return 0;
}
```

Output

```
Assertion failed: x==5, file test.cpp, line 13
This application has requested the Runtime to terminate it in an unusual
way. Please contact the application's support team for more information.
```

Assertion Vs Normal Error Handling

Assertions are mainly used to check logically impossible situations. For example, they can be used to check the state a code expects before it starts running or state after it finishes running. Unlike normal error handling, assertions are generally disabled at run-time. Therefore, it is not a good idea to write statements in asser() that can cause side effects. For example writing something like assert(x = 5) is not a good ideas as x is changed and this change won't happen when assertions are disabled. See this for more details.

Ignoring Assertions

In C/C++, we can completely remove assertions at compile time using the preprocessor NODEBUG.

```
// The below program runs fine because NDEBUG is defined
# define NDEBUG
# include <assert.h>

int main()
{
    int x = 7;
    assert (x==5);
    return 0;
}
```

The above program compiles and runs fine.

In Java, assertions are not enabled by default and we must pass an option to run-time engine to enable them.

Reference:

http://en.wikipedia.org/wiki/Assertion_%28software_development%29

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C

fork() in C

fork() system call is used to create a new process. Newly created process becomes child of the caller process. It take no parameters and return integer value. Below are different values returned by fork().

- ***Negative Value:*** creation of a child process was unsuccessful.
- ***Zero:*** Returned to the newly created child process.
- ***Positive value:*** Returned to parent or caller. The value contains process ID of newly created child process.

Examples:

1) Output of below program.

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
   pid_t pid = fork();
   if (pid == 0)
      printf("Child process created\n");
   else
      printf("Parent process created\n");
   return 0;
}
```

Output:

```
Parent process created
Child process created
```

In the above code, a child process is created, fork() returns 0 in the child process and positive integer to the parent process.

2) Calculate number of times hello is printed.

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
   fork();
   fork();
   fork();
   printf("hello\n");
   return 0;
}
```

Output:

```
hello
hello
hello
hello
hello
hello
hello
hello
```

Number of times hello printed is equal to number of process created. Total Number of Processes = 2^n where n is number of fork system calls. So here n=3, 2^3 = 8

Let us put some label names for the three lines:

```
 fork ();   // Line 1
 fork ();  // Line 2
 fork ();  // Line 3

     L1      // There will be 1 child process created by line 1
   /    \
 L2     L2    // There will be 2 child processes created by line 2
 / \   / \
L3  L3  L3  L3  // There will be 4 child processes created by line 3
```

So there are total eight processes (new child processes and one original process).

Please note that the above programs don't compile in Windows environment.

fork() vs exec()
The fork system call creates a new process. The new process created by fork() is copy of the current process except the returned value. The exex system call replaces the current process with a new program.

Exercise:
1) A process executes the following code

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

The total number of child processes created is: (GATE CS 2008)
(A) n
(B) 2^n – 1
(C) 2^n
(D) 2^(n+1) – 1;
See this for solution.

2) Consider the following code fragment:

```
if (fork() == 0)
{
  a = a + 5;
  printf("%d,%d\n", a, &a);
}
else
{
   a = a –5;
   printf("%d, %d\n", a, &a);
}
```

Let u, v be the values printed by the parent process, and x, y be the values printed by the child process. Which one of the following is TRUE? (GATE-CS-2005)
(A) u = x + 10 and v = y
(B) u = x + 10 and v != y
(C) u + 10 = x and v = y
(D) u + 10 = x and v != y
See this for solution.

3) Predict output of below program.

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

```
{
  fork();
  fork() && fork() || fork();
  fork();

  printf("forked\n");
  return 0;
}
```

See this for solution

References:

http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C Operating Systems

Interesting Facts in C Programming

Below are some interesting facts about C programming:

1) The case labels of a switch statement can occur inside if-else statements.

```
#include <stdio.h>

int main()
{
   int a = 2, b = 2;
   switch(a)
   {
   case 1:
      ;

      if (b==5)
      {
      case 2:
         printf("GeeksforGeeks");
      }
   else case 3:
   {

   }
   }
}
```

Output :

```
GeeksforGeeks
```

2) arr[index] is same as index[arr]
The reason for this to work is, array elements are accessed using pointer arithmetic.

```
// C program to demonstrate that arr[0] and
// 0[arr]
#include<stdio.h>
int main()
{
   int arr[10];
   arr[0] = 1;
```

```
    printf("%d", 0[arr] );

    return 0;
}
```

Output :

```
1
```

3) We can use ‘<:, :>’ in place of ‘[,]’ and ‘<%, %>’ in place of ‘{,}’

```
#include<stdio.h>
int main()
<%
    int arr <:10:>;
    arr<:0:> = 1;
    printf("%d", arr<:0:>);

    return 0;
%>
```

Output :

```
1
```

4) Using #include in strange places.
Let “a.txt” contains (“GeeksforGeeks”);

```
#include<stdio.h>
int main()
{
    printf
    #include "a.txt"
    ;
}
```

Output :

```
GeeksforGeeks
```

5) We can ignore input in scanf() by using an ‘*’ after ‘%’ in format specifiers

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

    // Let we input 10 20, we get output as 20
    // (First input is ignored)
    // If we remove * from below line, we get 10.
    scanf("%*d%d", &a);

    printf( "%d ",  a);

    return 0;
}
```

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
cpp-array
cpp-input-output

Precision of floating point numbers in C++ (floor(), ceil(), trunc(), round() and setprecision())

Decimal equivalent of 1/3 is 0.33333333333333…. An infinite length number would require infinite memory to store, and we typically have 4 or 8 bytes. Therefore, Floating point numbers store only a certain number of significant digits, and the rest are lost. The **precision** of a floating point number defines how many significant digits it can represent without information loss. When outputting floating point numbers, cout has a default precision of 6 and it truncates anything after that.

Given below are few libraries and methods which are used to provide precision to floating point numbers in C++:

floor():

Floor rounds off the given value to the closest integer which is less than the given value.

```
// C++ program to demonstrate working of floor()
// in C/C++
#include<bits/stdc++.h>
using namespace std;

int main()
{
    double x =1.411, y =1.500, z =1.711;
    cout << floor(x) << endl;
    cout << floor(y) << endl;
    cout << floor(z) << endl;

    double a =-1.411, b =-1.500, c =-1.611;
    cout << floor(a) << endl;
    cout << floor(b) << endl;
    cout << floor(c) << endl;
    return 0;
}
```

Output:

```
1
1
1
-2
-2
-2
```

ceil():

Ceil rounds off the given value to the closest integer which is more than the given value.

```
// C++ program to demonstrate working of ceil()
// in C/C++
#include<bits/stdc++.h>
using namespace std;

int main()
{
    double x = 1.411, y = 1.500, z = 1.611;
    cout << ceil(x) << endl;
    cout << ceil(y) << endl;
    cout << ceil(z) << endl;

    double a = -1.411, b = -1.500, c = -1.611;
    cout << ceil(a) << endl;
    cout << ceil(b) << endl;
    cout << ceil(c) << endl;
    return 0;
}
```

Output:

```
2
2
```

```
2
-1
-1
-1
```

trunc():

Trunc rounds removes digits after decimal point.

```
// C++ program to demonstrate working of trunc()
// in C/C++
#include<bits/stdc++.h>
using namespace std;

int main()
{
   double x = 1.411, y = 1.500, z = 1.611;
   cout << trunc(x) << endl;
   cout << trunc(y) << endl;
   cout << trunc(z) <<endl;

   double a = -1.411, b = -1.500, c = -1.611;
   cout << trunc(a) <<endl;
   cout << trunc(b) <<endl;
   cout << trunc(c) <<endl;
   return 0;
}
```

Output:

```
1
1
1
-1
-1
-1
```

round():

Rounds given number to the closest integer.

```
// C++ program to demonstrate working of round()
// in C/C++
#include<bits/stdc++.h>
using namespace std;

int main()
{
   double x = 1.411, y = 1.500, z = 1.611;

   cout << round(x) << endl;
   cout << round(y) << endl;
   cout << round(z) << endl;

   double a = -1.411, b = -1.500, c = -1.611;
   cout << round(a) << endl;
   cout << round(b) << endl;
   cout << round(c) << endl;
   return 0;
}
```

Output:

```
1
2
2
-1
-2
-2
```

setprecision():

Setprecision when used along with 'fixed' provides precision to floating point numbers correct to decimal numbers mentioned in the brackets of the setprecison.

```
// C++ program to demonstrate working of setprecision()
// in C/C++
#include<bits/stdc++.h>
using namespace std;

int main()
{
    double pi = 3.14159, npi = -3.14159;
    cout << fixed << setprecision(0) << pi <<" "<<npi<<endl;
    cout << fixed << setprecision(1) << pi <<" "<<npi<<endl;
    cout << fixed << setprecision(2) << pi <<" "<<npi<<endl;
    cout << fixed << setprecision(3) << pi <<" "<<npi<<endl;
    cout << fixed << setprecision(4) << pi <<" "<<npi<<endl;
    cout << fixed << setprecision(5) << pi <<" "<<npi<<endl;
    cout << fixed << setprecision(6) << pi <<" "<<npi<<endl;
}
```

Output:

```
3 -3
3.1 -3.1
3.14 -3.14
3.142 -3.142
3.1416 -3.1416
3.14159 -3.14159
3.141590 -3.141590
```

Note: When the value mentioned in the setprecision() exceeds the number of floating point digits in the original number then 0 is appended to floating point digit to match the precision mentioned by the user.

There exists other methods too to provide precision to floating point numbers. The above mentioned are few of the most commonly used methods to provide precision to floating point numbers during competitive coding.

C Language | Set 1

Following questions have been asked in GATE CS exam.

1. Consider the following three C functions :

```
[PI] int * g (void)
{
  int x = 10;
  return (&x);
}

[P2] int * g (void)
{
  int * px;
  *px = 10;
  return px;
}

[P3] int *g (void)
{
  int *px;
  px = (int *) malloc (sizeof(int));
  *px = 10;
  return px;
}
```

Which of the above three functions are likely to cause problems with pointers? (GATE 2001)

(a) Only P3

(b) Only P1 and P3

(c) Only P1 and P2

(d) P1, P2 and P3

Answer: (c)

Eplaination: In P1, pointer variable x is a local variable to g(), and g() returns pointer to this variable. x may vanish after g() has returned as x exists on stack. So, &x may become invalid.

In P2, pointer variable px is being assigned a value without allocating memory to it.

P3 works perfectly fine. Memory is allocated to pointer variable px using malloc(). So, px exists on heap, it's existence will remain in memory even after return of g() as it is on heap.

2. The value of j at the end of the execution of the following C program. (GATE CS 2000)

```
int incr (int i)
{
   static int count = 0;
   count = count + i;
   return (count);
}
main ()
{
   int i,j;
   for (i = 0; i <=4; i++)
      j = incr(i);
}
```

(a) 10

(b) 4

(c) 6

(d) 7

Answer (a)

Eplaination: count is static variable in incr(). Statement static int count = 0 will assign count to 0 only in first call. Other calls to this function will take the old values of count.

Count will become 0 after the call incr(0)

Count will become 1 after the call incr(1)

Count will become 3 after the call incr(2)

Count will become 6 after the call incr(3)

Count will become 10 after the call incr(4)

3. Consider the following C declaration

```
struct {
    short s [5]
    union {
        float y;
        long z;
    }u;
} t;
```

Assume that objects of the type short, float and long occupy 2 bytes, 4 bytes and 8 bytes, respectively. The memory requirement

for variable t, ignoring alignment considerations, is (GATE CS 2000)
(a) 22 bytes
(b) 14 bytes
(c) 18 bytes
(d) 10 bytes

Answer: (c)
Explanation: Short array s[5] will take 10 bytes as size of short is 2 bytes. Since u is a union, memory allocated to u will be max of float y(4 bytes) and long z(8 bytes). So, total size will be 18 bytes (10 + 8).

4. The number of tokens in the following C statement.

```
printf("i = %d, &i = %x", i, &i);
```

is (GATE 2000)
(a) 3
(b) 26
(c) 10
(d) 21

Answer (c)
Explanation: In a C source program, the basic element recognized by the compiler is the "token." A token is source-program text that the compiler does not break down into component elements.
There are 6 types of C tokens : identifiers, keywords, constants, operators, string literals and other separators. There are total 10 tokens in the above printf statement.

5. The following C declarations

```
struct node
{
  int i;
  float j;
};
struct node *s[10] ;
```

define s to be (GATE CS 2000)

(a) An array, each element of which is a pointer to a structure of type node
(b) A structure of 2 fields, each field being a pointer to an array of 10 elements
(c) A structure of 3 fields: an integer, a float, and an array of 10 elements
(d) An array, each element of which is a structure of type node.

Answer: (a)

GATE CS Corner   Company Wise Coding Practice

MCQ
GATE
GATE-CS-2000
GATE-CS-2001
GATE-CS-C-Language
GATE-CS-Older

C Language | Set 2

Following questions have been asked in GATE CS exam.

1. Consider the following C program segment:

```
char p[20];
char *s = "string";
int length = strlen(s);
int i;
for (i = 0; i < length; i++)
    p[i] = s[length — i];
printf("%s",p);
```

The output of the program is (GATE CS 2004)

a) gnirts

b) gnirt

c) string

d) no output is printed

Answer(d)

Let us consider below line inside the for loop

p[i] = s[length — i];

For i = 0, p[i] will be s[6 — 0] and s[6] is ‘\0’

So p[0] becomes ‘\0’. It doesn’t matter what comes in p[1], p[2]….. as P[0] will not change for i >0. Nothing is printed if we print a string with first character ‘\0’

2. Consider the following C function

```
void swap (int a, int b)
{
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

In order to exchange the values of two variables x and y. (GATE CS 2004)

a) call swap (x, y)

b) call swap (&x, &y)

c) swap (x,y) cannot be used as it does not return any value

d) swap (x,y) cannot be used as the parameters are passed by value

Answer(d)

Why a, b and c are incorrect?

a) call swap (x, y) will not cause any effect on x and y as parameters are passed by value.

b) call swap (&x, &y) will no work as function swap() expects values not addresses (or pointers).

c) swap (x, y) cannot be used but reason given is not correct.

3. Consider the following C function:

```
int f(int n)
{
  static int i = 1;
  if (n >= 5)
    return n;
  n = n+i;
  i++;
  return f(n);
}
```

The value returned by f(1) is (GATE CS 2004)

a) 5

b) 6

c) 7

d) 8

Answer (c)

Since i is static, first line of f() is executed only once.

```
Execution of f(1)
   i = 1
   n = 2
   i = 2
 Call f(2)
   i = 2
   n = 4
   i = 3
 Call f(4)
  i = 3
  n = 7
  i = 4
 Call f(7)
  since n >= 5 return n(7)
```

4. Consider the following program fragment for reversing the digits in a given integer to obtain a new integer. Let n = D1D2…Dm

```
int n, rev;
rev = 0;
while (n > 0)
{
   rev = rev*10 + n%10;
   n = n/10;
}
```

The loop invariant condition at the end of the ith iteration is:(GATE CS 2004)

a) n = D1D2….Dm-i and rev = DmDm-1…Dm-i+1

b) n = Dm-i+1…Dm-1Dm and rev = Dm-1….D2D1

c) n ≠ rev

d) n = D1D2….Dm and rev = DmDm-1…D2D1

Answer (a)

5. Consider the following C program

```
main()
{
   int x, y, m, n;
   scanf ("%d %d", &x, &y);
   /* x > 0 and y > 0 */
   m = x; n = y;
   while (m != n)
   {
     if(m>n)
       m = m - n;
     else
       n = n - m;
   }
   printf("%d", n);
}
```

The program computes (GATE CS 2004)

a) x + y using repeated subtraction

b) x mod y using repeated subtraction

c) the greatest common divisor of x and y

d) the least common multiple of x and y

Answer(c)

This is an implementation of Euclid's algorithm to find GCD

GATE CS Corner   Company Wise Coding Practice

MCQ
GATE
GATE-CS-2004
GATE-CS-C-Language

C Language | Set 3

Following questions have been asked in GATE CS exam.

1.Assume the following C variable declaration

```
int *A [10], B[10][10];
```

Of the following expressions
I A[2]
II A[2][3]
III B[1]
IV B[2][3]
which will not give compile-time errors if used as left hand sides of assignment statements in a C program (GATE CS 2003)?

a) I, II, and IV only
b) II, III, and IV only
c) II and IV only
d) IV only

Answer (a)
See below program

```
int main()
{
  int *A[10], B[10][10];
  int C[] = {12, 11, 13, 14};

  /* No problem with below statement as A[2] is a pointer
     and we are assigning a value to pointer */
  A[2] = C;

  /* No problem with below statement also as array style indexing
      can be done with pointers*/
  A[2][3] = 15;

  /* Simple assignment to an element of a 2D array*/
  B[2][3]  = 15;

  printf("%d %d", A[2][0], A[2][3]);
  getchar();
}
```

2. Consider the following declaration of a ‘two-dimensional array in C:

```
char a[100][100];
```

Assuming that the main memory is byte-addressable and that the array is stored starting from memory address 0, the address of a[40][50] is (GATE CS 2002)
a) 4040
b) 4050
c) 5040
d) 5050

Answer(b)

```
Address of a[40][50] =
     Base address + 40*100*element_size + 50*element_size
     0 + 4000*1 + 50*1
     4050
```

3. The C language is. (GATE CS 2002)

a) A context free language

b) A context sensitive language

c) A regular language

d) Parsable fully only by a Turing machine

Answer (a)

Most programming languages including C, C++, Java and Pascal can be approximated by context-free grammar and compilers for them have been developed based on properties of context-free languages.

References:

http://acm.pku.edu.cn/JudgeOnline/problem?id=3220

http://www.cs.odu.edu/~toida/nerzic/390teched/cfl/cfg.html

4 The most appropriate matching for the following pairs (GATE CS 2000)

```
X: m=malloc(5); m= NULL;       1: using dangling pointers
Y: free(n); n->value=5;        2: using uninitialized pointers
Z: char *p; *p = 'a';          3. lost memory is:
```

(a) X—1 Y—3 Z-2

(b) X—2 Y—1 Z-3

(C) X—3 Y—2 Z-1

(d) X—3 Y—1 Z-2

Answer (d)

X -> A pointer is assigned to NULL without freeing memory so a clear example of memory leak

Y -> Trying to retrieve value after freeing it so dangling pointer.

Z -> Using uninitialized pointers

5. Consider the following C-program:

```
void foo(int n, int sum)
{
  int k = 0, j = 0;
  if (n == 0) return;
    k = n % 10;
  j = n / 10;
  sum = sum + k;
  foo (j, sum);
  printf ("%d,", k);
}

int main ()
{
  int a = 2048, sum = 0;
  foo (a, sum);
  printf ("%d\n", sum);

  getchar();
}
```

What does the above program print? (GATE CS 2005)

(a) 8, 4, 0, 2, 14

(b) 8, 4, 0, 2, 0

(C) 2, 0, 4, 8, 14

(d) 2, 0, 4, 8, 0

Answer (d)

sum has no use in foo(), it is there just to confuse. Function foo() just prints all digits of a number. In main, there is one more printf statement after foo(), so one more 0 is printed after all digits of n.

Please write comments if you find any of the above answers/explanations incorrect or you want to add some more information about questions.

C Language | Set 4

Following questions have been asked in GATE CS exam.

1. In the C language (GATE CS 2002)

a) At most one activation record exists between the current activation record and the activation record for the main

b) The number of activation records between the current activation record and the activation record for the main depends on the actual function calling sequence.

c) The visibility of global variables depends on the actual function calling sequence.

d) Recursion requires the activation record for the recursive function to be saved on a different stack before the recursive function can be called.

Answer(b)

a) –> There is no such restriction in C language

b) –> True

c) –> False. In C, variables are statically scoped, not dynamically.

c) –> False. The activation records are stored on the same stack.

2. Consider the C program shown below.

```
# include <stdio.h>
# define print(x)  printf ("%d", x)
int x;
void Q(int z)
{
  z += x;
  print(z);
}
void P(int *y)
{
  int x = *y+2;
  Q(x);
  *y = x-1;
  print(x);
}

main(void)
{
  x=5;
  P(&x);
  print(x);
  getchar();
}
```

The output of this program is (GATE CS 2003)

a) 1276

b) 22 12 11

c) 14 6 6

d) 766

Answer (a)

Note that main() and Q() are accessing the global variable x. Inside P(), pointer variable y also holds address of global variable x, but x in P() is its P's own local variable.

Please see GATE Corner for all previous year paper/solutions/explanations, syllabus, important dates, notes, etc.

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above.

C Language | Set 5

Following questions have been asked in GATE CS 2008 exam.

1. What is printed by the following C program?

```
int f(int x, int *py, int **ppz)
{
  int y, z;
  **ppz += 1;
   z  = **ppz;
  *py += 2;
   y = *py;
   x += 3;
   return x + y + z;
}

void main()
{
   int c, *b, **a;
   c = 4;
   b = &c;
   a = &b;
   printf( "%d", f(c,b,a));
   getchar();
}
```

(A) 18

(B) 19

(C) 21

(D) 22

Answer (B)

```
/* Explanation for the answer */

  /*below line changes value of c to 5. Note that x remains unaffected
    by this change as x is a copy of c and address of x is different from c*/
  **ppz += 1

  /* z is changed to 5*/
  z  = **ppz;

  /* changes c to 7, x is not changed */
  *py += 2;

   /* y is changed to 7*/
  y = *py;

  /* x is incremented by 3 */
   x += 3;

  /* return 7 + 7 + 5*/
  return x + y + z;
```

2. Choose the correct option to fill ?1 and ?2 so that the program below prints an input string in reverse order. Assume that the input string is terminated by a newline character.

```
void reverse(void)
 {
  int c;
  if (?1) reverse() ;
  ?2
}
main()
{
  printf ("Enter Text ") ;
  printf ("\n") ;
  reverse();
  printf ("\n") ;
}
```

(A) ?1 is (getchar() != '\n')
?2 is getchar(c);
(B) ?1 is (c = getchar()) != '\n')
?2 is getchar(c);
(C) ?1 is (c != '\n')
?2 is putchar(c);
(D) ?1 is ((c = getchar()) != '\n')
?2 is putchar(c);

Answer(D)
getchar() is used to get the input character from the user and putchar() to print the entered character, but before printing reverse is called again and again until '\n' is entered. When '\n' is entered the functions from the function stack run putchar() statements one by one. Therefore, last entered character is printed first.
You can try running below program

```
void reverse(void); /* function prototype */

void reverse(void)
 {
  int c;
  if (((c = getchar()) != '\n'))
    reverse();
  putchar(c);
}
main()
{
  printf ("Enter Text ") ;
  printf ("\n") ;
  reverse();
  printf ("\n") ;
  getchar();
}
```

For questions 3 & 4, consider the following C functions:

```
int f1(int n)
{
  if(n == 0 || n == 1)
    return n;
  else
    return (2*f1(n-1) + 3*f1(n-2));
}

int f2(int n)
{
  int i;
  int X[N], Y[N], Z[N] ;
  X[0] = Y[0] = Z[0] = 0;
  X[1] = 1; Y[1] = 2; Z[1] = 3;
  for(i = 2; i <= n; i++)
  {
    X[i] = Y[i-1] + Z[i-2];
    Y[i] = 2*X[i];
    Z[i] = 3*X[i];
  }
  return X[n] ;
}
```

3. The running time of f1(n) and f2(n) are
(A) Θ(n) and Θ(n)
(B) Θ(2^n) and Θ(n)
(C) Θ(n) and Θ(2^n)
(D) Θ(2^n) and Θ(2^n)

Answer (B)
For f1(), let T(n) be the function for time complexity.

```
T(n) = T(n-1) + T(n-2)
```

Above recursion is a standard one for Fibonacci Numbers. After solving the recursion, we get

```
T(n) = 1/sqrt(5)[(1 + sqrt(5))/2]^n - 1/sqrt(5)[(1 - sqrt(5))/2]^n
```

Above recursion can also be written as Θ(1.618.^n)
(Please see this).

In f2(), there is a single loop, so time complexity is Θ(n)

Among all the 4 given choices, (B) looks closest.

4. f1(8) and f2(8) return the values
(A) 1661 and 1640
(B) 59 and 59
(C) 1640 and 1640
(D) 1640 and 1661

Both functions perform same operation, so output is same, means either (B) or (C) is correct.
f1(2) = 2*f1(1) + 3*f1(0) = 2
f1(3) = 2*f1(2) + 3*f1(1) = 2*2 + 3*1 = 7
f1(4) = 2*f1(3) + 3*f1(2) = 2*7 + 3*2 = 20
f1(5) = 2*f1(4) + 3*f1(3) = 2*20 + 3*7 = 40 + 21 = 61

We can skip after this as the only remaining choice is (C)
f1(6) = 2*f1(5) + 3*f1(4) = 2*61 + 3*20 = 122 + 60 = 182
f1(7) = 2*f1(6) + 3*f1(5) = 2*182 + 3*61 = 364 + 183 = 547
f1(8) = 2*f1(7) + 3*f1(6) = 2*547 + 3*182 = 1094 + 546 = 1640

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above.

GATE CS Corner   Company Wise Coding Practice

MCQ
GATE
GATE-CS-2008
GATE-CS-C-Language

C Language | Set 6

Following questions have been asked in GATE CS 2006 exam.

1. Consider the following C-program fragment in which i, j and n are integer variables.

```
for (i = n, j = 0; i >0; i /= 2, j += i);
```

Let val(j) denote the value stored in the variable j after termination of the for loop. Which one of the following is true?

(A) val(j) = Θ(logn)
(B) val(j) = Θ(sqrt(n))
(C) val(j) = Θ(n)
(D) val(j) = Θ(nlogn)

Answer (C)

Note the semicolon after the for loop, so there is nothing in the body. The variable j is initially 0 and value of j is sum of values of i. i is initialized as n and is reduced to half in each iteration.

j = n/2 + n/4 + n/8 + .. + 1 = Θ(n)

2. Consider the following C-function in which a[n] and b[m] are two sorted integer arrays and c[n + m] be another integer array.

```
void xyz(int a[], int b [], int c[])
{
  int i, j, k;
  i = j = k = O;
  while ((i<n) && (j<m))
     if (a[i] < b[j]) c[k++] = a[i++];
     else c[k++] = b[j++];
}
```

Which of the following condition(s) hold(s) after the termination of the while loop?
(i) j
(A) only (i)
(B) only (ii)
(C) either (i) or (ii) but not both
(D) neither (i) nor (ii)

Answer (C)
The condition (i) is true if the last inserted element in c[] is from a[] and condition (ii) is true if the last inserted element is from b[].

3. Consider this C code to swap two integers and these five statements: the code

```
void swap(int *px, int *py)
{
   *px = *px - *py;
   *py = *px + *py;
   *px = *py - *px;
}
```

S1: will generate a compilation error
S2: may generate a segmentation fault at runtime depending on the arguments passed
S3: correctly implements the swap procedure for all input pointers referring to integers stored in memory locations accessible to the process
S4: implements the swap procedure correctly for some but not all valid input pointers
S5: may add or subtract integers and pointers.
(A) S1
(B) S2 and S3
(C) S2 and S4
(D) S2 and S5

Answer (C)
S2: May generate segmentation fault if value at pointers px or py is constant or px or py points to a memory location that is invalid
S4: May not work for all inputs as arithmetic overflow can occur.

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above.

C Language | Set 7

Following questions have been asked in GATE CS 2010 exam.

1. What does the following program print?

```
#include<stdio.h>
void f(int *p, int *q)
{
  p = q;
 *p = 2;
}
int i = 0, j = 1;
int main()
{
  f(&i, &j);
  printf("%d %d \n", i, j);
  getchar();
  return 0;
}
```

(A) 2 2
(B) 2 1
(C) 0 1
(D) 0 2

Answer (D)

See below f() with comments for explanation.

```
/* p points to i and q points to j */
void f(int *p, int *q)
{
  p = q;    /* p also points to j now */
 *p = 2;   /* Value of j is changed to 2 now */
}
```

2. What is the value printed by the following C program?

```
#include<stdio.h>
int f(int *a, int n)
{
  if(n <= 0) return 0;
  else if(*a % 2 == 0) return *a + f(a+1, n-1);
  else return *a - f(a+1, n-1);
}

int main()
{
  int a[] = {12, 7, 13, 4, 11, 6};
  printf("%d", f(a, 6));
  getchar();
  return 0;
}
```

(A) -9
(B) 5
(C) 15
(D) 19

Answer (C)

f() is a recursive function which adds f(a+1, n-1) to *a if *a is even. If *a is odd then f() subtracts f(a+1, n-1) from *a. See below recursion tree for execution of f(a, 6).

.

```
f(add(12), 6) /*Since 12 is first element. a contains address of 12 */
  |
  |
12 + f(add(7), 5) /* Since 7 is the next element, a+1 contains address of 7 */
     |
     |
   7 - f(add(13), 4)
          |
          |
        13 - f(add(4), 3)
               |
               |
             4 + f(add(11), 2)
                    |
                    |
                  11 - f(add(6), 1)
                         |
                         |
                       6 + 0
```

So, the final returned value is 12 + (7 – (13 – (4 + (11 – (6 + 0))))) = 15

Please see GATE Corner for all previous year paper/solutions/explanations, syllabus, important dates, notes, etc.

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above.

GATE CS Corner   Company Wise Coding Practice

MCQ
GATE
GATE-CS-2010
GATE-CS-C-Language

C Language | Set 8

Following questions have been asked in GATE CS 2011 exam.

1) What does the following fragment of C-program print?

```
char c[] = "GATE2011";
char *p =c;
printf("%s", p + p[3] - p[1]) ;
```

(A) GATE2011

(B) E2011

(C) 2011

(D) 011

Answer: (C)
See comments for explanation.

```
char c[] = "GATE2011";

 // p now has the base address string "GATE2011"
char *p =c;

// p[3] is 'E' and p[1] is 'A'.
// p[3] - p[1] = ASCII value of 'E' - ASCII value of 'A' = 4
// So the expression  p + p[3] - p[1] becomes p + 4 which is
// base address of string "2011"
printf("%s", p + p[3] - p[1]) ;
```

2) Consider the following recursive C function that takes two arguments

```
unsigned int foo(unsigned int n, unsigned int r) {
  if (n  > 0) return (n%r +  foo (n/r, r ));
  else return 0;
}
```

What is the return value of the function foo when it is called as foo(513, 2)?

(A) 9

(B) 8

(C) 5

(D) 2

Answer: (D)

foo(513, 2) will return 1 + foo(256, 2). All subsequent recursive calls (including foo(256, 2)) will return 0 + foo(n/2, 2) except the last call foo(1, 2) . The last call foo(1, 2) returns 1. So, the value returned by foo(513, 2) is 1 + 0 + 0…. + 0 + 1.

The function foo(n, 2) basically returns sum of bits (or count of set bits) in the number n.

3) What is the return value of the function foo when it is called as foo(345, 10) ?

(A) 345

(B) 12

(C) 5

(D) 3

Answer: (B)

The call foo(345, 10) returns sum of decimal digits (because r is 10) in the number n. Sum of digits for 345 is 3 + 4 + 5 = 12.

Please see GATE Corner for all previous year paper/solutions/explanations, syllabus, important dates, notes, etc.

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above.

GATE CS Corner   Company Wise Coding Practice

MCQ
GATE
GATE-CS-2011
GATE-CS-C-Language

C Language | Set 9

Following questions have been asked in GATE 2012 exam.

1. What will be the output of the following C program segment?

```
char inchar = 'A';
switch (inchar)
{
case 'A' :
   printf ("choice A \n") ;
case 'B' :
   printf ("choice B ") ;
case 'C' :
case 'D' :
case 'E' :
default:
   printf ("No Choice") ;
}
```

(A) No choice

(B) Choice A

(C) Choice A
Choice B No choice

(D) Program gives no output as it is erroneous

Answer (C)

There is no break statement in case ‘A’. If a case is executed and it doesn’t contain break, then all the subsequent cases are executed until a break statement is found. That is why everything inside the switch is printed.

Try following program as an exercise.

```
int main()
{
   char inchar = 'A';
   switch (inchar)
   {
   case 'A' :
      printf ("choice A \n") ;
   case 'B' :
   {
      printf ("choice B") ;
      break;
   }
   case 'C' :
   case 'D' :
   case 'E' :
   default:
      printf ("No Choice") ;
   }
}
```

2. Consider the following C program

```
int a, b, c = 0;
void prtFun (void);
int main ()
{
   static int a = 1; /* line 1 */
   prtFun();
   a += 1;
   prtFun();
   printf ( "\n %d %d " , a, b) ;
}

void prtFun (void)
{
   static int a = 2; /* line 2 */
   int b = 1;
   a += ++b;
   printf (" \n %d %d " , a, b);
}
```

What output will be generated by the given code segment?

(A) 3 1

4 1

4 2

(B) 4 2

6 1

6 1

(C) 4 2

6 2

2 0

(D) 3 1

5 2

5 2

Answer (C)

‘a’ and ‘b’ are global variable. prtFun() also has ‘a’ and ‘b’ as local variables. The local variables hide the globals (See Scope rules in C). When prtFun() is called first time, the local ‘b’ becomes 2 and local ‘a’ becomes 4.

When prtFun() is called second time, same instance of local static ‘a’ is used and a new instance of ‘b’ is created because ‘a’ is static and ‘b’ is non-static. So ‘b’ becomes 2 again and ‘a’ becomes 6.

main() also has its own local static variable named ‘a’ that hides the global ‘a’ in main. The printf() statement in main() accesses the local ‘a’ and prints its value. The same printf() statement accesses the global ‘b’ as there is no local variable named ‘b’ in main. Also, the defaut value of static and global int variables is 0. That is why the printf statement in main() prints 0 as value of b.

3. What output will be generated by the given code d\segment if:
Line 1 is replaced by "auto int a = 1;"
Line 2 is replaced by "register int a = 2;"

(A) 3 1
4 1
4 2
(B) 4 2
6 1
6 1
(C) 4 2
6 2
2 0
(D) 4 2
4 2
2 0

Answer (D)
If we replace line 1 by "auto int a = 1;" and line 2 by "register int a = 2;", then 'a' becomes non-static in prtFun(). The output of first prtFun() remains same. But, the output of second prtFun() call is changed as a new instance of 'a' is created in second call. So "4 2" is printed again. Finally, the printf() in main will print "2 0". Making 'a' a register variable won't change anything in output.

Please see GATE Corner for all previous year paper/solutions/explanations, syllabus, important dates, notes, etc.

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above.

GATE CS Corner   Company Wise Coding Practice

MCQ
GATE
GATE-CS-2012
GATE-CS-C-Language

Output of C Programs | Set 1

Predict the output of below programs.

Question 1

```
#include<stdio.h>
int main()
{
  int n;
  for(n = 7; n!=0; n--)
    printf("n = %d", n--);
  getchar();
  return 0;
}
```

Output:Above program goes in infinite loop because n is never zero when loop condition (n != 0) is checked.

Question 2

```
#include<stdio.h>
int main()
{
  printf("%x", -1<<1);
  getchar();
  return 0;
}
```

Output is dependent on the compiler. For 32 bit compiler it would be fffffffe and for 16 bit it would be fffe.

Question 3

```
# include <stdio.h>
# define scanf  "%s Geeks For Geeks "
main()
{
  printf(scanf, scanf);
  getchar();
  return 0;
}
```

Output: %s Geeks For Geeks Geeks For Geeks
Explanation: After pre-processing phase of compilation, printf statement will become.

```
  printf("%s Geeks For Geeks ",  "%s Geeks For Geeks ");
```

Now you can easily guess why output is %s Geeks For Geeks Geeks For Geeks.

Question 4

```
#include <stdlib.h>
#include <stdio.h>
enum {false, true};
int main()
{
  int i = 1;
  do
  {
    printf("%d\n", i);
    i++;
    if (i < 15)
      continue;
  } while (false);

  getchar();
  return 0;
}
```

Output: 1
Explanation: The do wile loop checks condition after each iteration. So after continue statement, control transfers to the statement while(false). Since the condition is false 'i' is printed only once.

Now try below program.

```
#include <stdlib.h>
#include <stdio.h>
enum {false, true};
int main()
{
  int i = 1;
  do
  {
    printf("%d\n", i);
    i++;
    if (i < 15)
      break;
    } while (true);

    getchar();
    return 0;
}
```

Question 5

```
char *getString()
{
  char *str = "Nice test for strings";
  return str;
}
```

```
int main()
{
  printf("%s", getString());
  getchar();
  return 0;
}
```

Output: “Nice test for strings”

The above program works because string constants are stored in Data Section (not in Stack Section). So, when getString returns *str is not lost.

Output of C Programs | Set 2

Predict the output of below programs.

Question 1

```
char *getString()
{
   char str[] = "Will I be printed?";
   return str;
}
int main()
{
   printf("%s", getString());
   getchar();
}
```

Output: Some garbage value

The above program doesn’t work because array variables are stored in Stack Section. So, when getString returns values at str are deleted and str becomes dangling pointer.

Question 2

```
int main()
{
   static int i=5;
   if(--i){
      main();
      printf("%d ",i);
   }
}
```

Output: 0 0 0 0

Explanation: Since i is a static variable and is stored in Data Section, all calls to main share same i.

Question 3

```
int main()
{
   static int var = 5;
   printf("%d ",var--);
   if(var)
      main();
}
```

Output: 5 4 3 2 1

Explanation: Same as previous question. The only difference here is, sequence of calling main and printf is changed, therefore different output.

Question 4

```
int main()
{
   int x;
   printf("%d",scanf("%d",&x));
   /* Suppose that input value given
      for above scanf is 20 */
   return 1;
}
```

Output: 1
scanf returns the no. of inputs it has successfully read.

Question 5

```
# include <stdio.h>
int main()
{
  int i=0;
  for(i=0; i<20; i++)
  {
    switch(i)
    {
      case 0:
        i+=5;
      case 1:
        i+=2;
      case 5:
        i+=5;
      default:
        i+=4;
        break;
    }
    printf("%d  ", i);
  }

  getchar();
  return 0;
}
```

Output: 16 21
Explanation:
Initially i = 0. Since case 0 is true i becomes 5, and since there is no break statement till last statement of switch block, i becomes 16. Now in next iteration no case is true, so execution goes to default and i becomes 21.

In C, if one case is true switch block is executed until it finds break statement. If no break statement is present all cases are executed after the true case. If you want to know why switch is implemented like this, well this implementation is useful for situations like below.

```
switch (c)
{
   case 'a':
   case 'e':
   case 'i' :
   case 'o':
   case 'u':
     printf(" Vowel character");
     break;
   default :
     printf("Not a Vowel character");; break;
 }
```

GATE CS Corner   Company Wise Coding Practice

Output
C-Output

Output of C Programs | Set 3

Predict the output of the below program.

Question 1

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

Output: Address of function main.

Explanation: Name of the function is actually a pointer variable to the function and prints the address of the function. Symbol table is implemented like this.

```
struct
{
  char *name;
  int (*funcptr)();
}
symtab[] = {
  "func", func,
  "anotherfunc", anotherfunc,
};
```

Question 2

```
#include <stdio.h>
int main()
{
  printf("\new_c_question\by");
  printf("\rgeeksforgeeks");

  getchar();
  return 0;
}
```

Output: geeksforgeeksl

Explanation: First printf prints “ew_c_questioy”. Second printf has \r in it so it goes back to start of the line and starts printing characters.

Now try to print following without using any of the escape characters.

```
new c questions by
geeksforgeeks
```

Question 3

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

void fun(int *a)
{
   a = (int*)malloc(sizeof(int));
}

int main()
{
   int *p;
   fun(p);
   *p = 6;
   printf("%d\n",*p);

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

It does not work. Try replacing “int *p;” with “int *p = NULL;” and it will try to dereference a null pointer.

This is because fun() makes a copy of the pointer, so when malloc() is called, it is setting the copied pointer to the memory location, not p. p is pointing to random memory before and after the call to fun(), and when you dereference it, it will crash.

If you want to add memory to a pointer from a function, you need to pass the address of the pointer (ie. double pointer).

Thanks to John Doe for providing the correct solution.

Question 4

```
#include <stdio.h>
int main()
{
   int i;
   i = 1, 2, 3;
   printf("i = %d\n", i);

   getchar();
   return 0;
}
```

Output: 1
The above program prints 1. Associativity of comma operator is from left to right, but = operator has higher precedence than comma operator.
Therefore the statement i = 1, 2, 3 is treated as (i = 1), 2, 3 by the compiler.

Now it should be easy to tell output of below program.

```
#include <stdio.h>
int main()
{
   int i;
   i = (1, 2, 3);
   printf("i  = %d\n", i);

   getchar();
    return 0;
}
```

Question 5

```
#include <stdio.h>
int main()
{
   int first = 50, second = 60, third;
   third = first /* Will this comment work? */ + second;
   printf("%d /* And this? */ \n", third);

   getchar();
   return 0;
}
```

Output: 110 /* And this? */
Explanation: Compiler removes everything between “/*” and “*/” if they are not present inside double quotes (“”).

GATE CS Corner   Company Wise Coding Practice

Output
C-Output

Output of C Programs | Set 4

Predict the output of below programs

Question 1

```
#include<stdio.h>
int main()
{
   struct site
   {
      char name[] = "GeeksforGeeks";
      int no_of_pages = 200;
   };
   struct site *ptr;
   printf("%d",ptr->no_of_pages);
   printf("%s",ptr->name);
   getchar();
   return 0;
}
```

Output:
Compiler error

Explanation:
Note the difference between structure/union declaration and variable declaration. When you declare a structure, you actually declare a new data type suitable for your purpose. So you cannot initialize values as it is not a variable declaration but a data type declaration.

Reference:
http://www.lix.polytechnique.fr/~liberti/public/computing/prog/c/C/SYNTAX/struct.html

Question 2

```
int main()
{
   char a[2][3][3] = {'g','e','e','k','s','f','o',
                      'r','g','e','e','k','s'};
   printf("%s ", **a);
   getchar();
   return 0;
}
```

Output:
geeksforgeeks

Explanation:
We have created a 3D array that should have 2*3*3 (= 18) elements, but we are initializing only 13 of them. In C when we initialize less no of elements in an array all uninitialized elements become '\0' in case of char and 0 in case of integers.

Question 3

```
int main()
{
  char str[]= "geeks\nforgeeks";
  char *ptr1, *ptr2;

  ptr1 = &str[3];
  ptr2 = str + 5;
  printf("%c", ++*str - --*ptr1 + *ptr2 + 2);
  printf("%s", str);

  getchar();
  return 0;
}
```

Output:
heejs
forgeeks

Explanation:
Initially ptr1 points to 'k' and ptr2 points to '\n' in "geeks\nforgeeks". In print statement value at str is incremented by 1 and value at ptr1 is decremented by 1. So string becomes "heejs\nforgeeks" .

First print statement becomes
printf("%c", 'h' – 'j' + 'n' + 2)

‘h’ – ‘j’ + ‘\n’ + 2 = -2 + ‘\n’ + 2 = ‘\n’

First print statements newline character. and next print statement prints “heejs\nforgeeks”.

Question 4

```
#include <stdio.h>
int fun(int n)
{
    int i, j, sum = 0;
    for(i = 1;i<=n;i++)
        for(j=i;j<=i;j++)
            sum=sum+j;
    return(sum);
}

int main()
{
    printf("%d", fun(15));
    getchar();
    return 0;
}
```

Output: 120
Explanation: fun(n) calculates sum of first n integers or we can say it returns n(n+1)/2.

Question 5

```
#include <stdio.h>
int main()
{
    int c = 5, no = 1000;
    do {
        no /= c;
    } while(c--);

    printf ("%d\n", no);
    return 0;
}
```

Output: Exception – Divide by zero

Explanation: There is a bug in the above program. It goes inside the do-while loop for c = 0 also. Be careful when you are using do-while loop like this!!

GATE CS Corner   Company Wise Coding Practice

Output
C-Output

Output of C Programs | Set 5

Predict the output of below programs

Question 1

```
int main()
{
    while(1){
        if(printf("%d",printf("%d")))
            break;
        else
            continue;
    }
    return 0;
}
```

Output:
Can’t be predicted

Explanation:
The condition in while loop is 1 so at first shot it looks infinite loop. Then there are break and continue in the body of the while loop, so it may not be infinite loop. The break statement will be executed if the condition under if is met, otherwise continue will be executed. Since there’s no other statements after continue in the while loop, continue doesn’t serve any purpose. In fact it is extraneous. So let us see the if condition. If we look carefully, we will notice that the condition of the if will be met always, so break will be executed at the first iteration itself and we will come out of while loop. The reason why the condition of if will be met is printf function. Function printf always returns the no. of bytes it has output. For example, the return value of printf(“geeks”) will be 5 because printf will output 5 characters here. In our case, the inner printf will be executed first but this printf doesn’t have argument for format specifier i.e. %d. It means this printf will print any arbitrary value. But notice that even for any arbirary value, the no. of bytes output by inner printf would be non-zero. And those no. of bytes will work as argument to outer printf. The outer printf will print that many no. of bytes and return non-zero value always. So the condition for if is also true always. Therefore, the while loop be executed only once. As a side note, even without outer printf also, the condition for if is always true.

Question 2

```
int main()
{
   unsigned int i=10;
   while(i-- >= 0)
      printf("%u ",i);
   return 0;
}
```

Output:
9 8 7 6 5 4 3 2 1 0 4294967295 4294967294 …… (on a machine where int is 4 bytes long)

9 8 7 6 5 4 3 2 1 0 65535 65534 …. (on a machine where int is 2 bytes long)

Explanation:
Let us examine the condition of while loop. It is obvious that as far as the condition of while loop is met, printf will be executed. There are two operators in the condition of while loop: post-decrement operator and comparison operator. From operator precedence, we know that unary operator post-decrement has higher priority than comparison operator. But due to post-decrement property, the value of i will be decremented only after it had been used for comparison. So at the first iteration, the condition is true because 10>=0 and then i is decremented. Therefore 9 will be printed. Similarly the loop continues and the value of i keeps on decrementing. Let us see what what happen when condition of while loop becomes 0 >= 0. At this time, condition is met and i is decremented. Since i is unsigned integer, the roll-over happens and i takes the value of the highest +ve value an unsigned int can take. So i is never negative. Therefore, it becomes infinite while loop.

As a side note, if i was signed int, the while loop would have been terminated after printing the highest negative value.

Question 3

```
int main()
{
   int x,y=2,z,a;
   if ( x = y%2)
       z =2;
   a=2;
   printf("%d %d ",z,x);
   return 0;
}
```

Output:
< some garbage value of z > 0

Explanation:
This question has some stuff for operator precedence. If the condition of if is met, then z will be initialized to 2 otherwise z will contain garbage value. But the condition of if has two operators: assignment operator and modulus operator. The precedence of modulus is higher than assignment. So y%2 is zero and it’ll be assigned to x. So the value of x becomes zero which is also the effective condition for if. And therefore, condition of if is false.

Question 4

```
int main()
{
   int a[10];
   printf("%d",*a+1-*a+3);
   return 0;
}
```

Output: 4

Explanation:
From operator precedence, de-reference operator has higher priority than addition/subtraction operator. So de-reference will be applied first. Here, a is an array which is not initialized. If we use a, then it will point to the first element of the array. Therefore *a will be the first element of the array. Suppose first element of array is x, then the argument inside printf becomes as follows. It's effective value is 4.

x + 1 – x + 3 = 4

Question 5

```
#define prod(a,b) a*b
int main()
{
   int x=3,y=4;
   printf("%d",prod(x+2,y-1));
   return 0;
}
```

Output:
10

Explanation:
This program deals with macros, their side effects and operator precedence. Here prod is a macro which multiplies its two arguments a and b. Let us take a closer look.

prod(a, b) = a*b
prod(x+2, y-1) = x+2*y-1 = 3+2*4-1 = 3+8-1=10

If the programmer really wanted to multiply x+2 and y-1, he should have put parenthesis around a and b as follows.

prod(a,b) = (a)*(b)

This type of mistake in macro definition is called – macro side-effects.

Output
C-Output

Output of C Programs | Set 6

Predict the output of below programs

Question 1

```
int main()
{
   unsigned int i=65000;
   while ( i++ != 0 );
   printf("%d",i);
   return 0;
}
```

Output:
1

Explanation:
It should be noticed that there's a semi-colon in the body of while loop. So even though, nothing is done as part of while body, the control will come out of while only if while condition isn't met. In other words, as soon as i inside the condition becomes 0, the condition will

become false and while loop would be over. But also notice the post-increment operator in the condition of while. So first i will be compared with 0 and i will be incremented no matter whether condition is met or not. Since i is initialized to 65000, it will keep on incrementing till it reaches highest positive value. After that roll over happens, and the value of i becomes zero. The condition is not met, but i would be incremented i.e. to 1. Then printf will print 1.

Question 2

```
int main()
{
    int i=0;
    while ( +(+i--) != 0)
        i-=i++;
    printf("%d",i);
    return 0;
}
```

Output:
-1

Explanation:
Let us first take the condition of while loop. There are several operator there. Unary + operator doesn’t do anything. So the simplified condition becomes (i–) != 0. So i will be compared with 0 and then decremented no matter whether condition is true or false. Since i is initialized to 0, the condition of while will be false at the first iteration itself but i will be decremented to -1. The body of while loop will not be executed. And printf will print -1.

So it wasn’t that scary as it seemed to be!

Question 3

```
int main()
{
    float f=5,g=10;
    enum{i=10,j=20,k=50};
    printf("%d\n",++k);
    printf("%f\n",f<<2);
    printf("%lf\n",f%g);
    printf("%lf\n",fmod(f,g));
    return 0;
}
```

Output:
Program will not compile and give 3 errors

Explanation:
Here, i, j and k are inside the enum and therefore, they are like constants. In other words, if want to us 10 anywhere in the program , we can use i instead. In the first printf, the value of i is being modified which is not allowed because it’s enum constant. In the second printf, left-shift operator is being applied on a float which is also not allowed. Similarly, in the third printf, modulus operator is being applied on float f and g which is also not allowed.

Question 4

```
int main()
{
    int i=10;
    void pascal f(int,int,int);
    f(i++, i++, i++);
    printf(" %d",i);
    return 0;
}
void pascal f(integer :i,integer:j,integer :k)
{
  write(i,j,k);
}
```

Output:
Program will give compile-time error

Explanation:

Compiler specific question. Not all compilers support this.

Otherwise, pascal enforces left to right processing of arguments. So even though, the argument processing order can be changed by the use of pascal, we can't use Pascal language routines such as write inside C program.

Question 5

```
void pascal f(int i,int j,int k)
{
  printf("%d %d %d",i, j, k);
}

void cdecl f(int i,int j,int k)
{
  printf("%d %d %d",i, j, k);
}

main()
{
    int i=10;
    f(i++,i++,i++);
    printf(" %d\n",i);
    i=10;
    f(i++,i++,i++);
    printf(" %d",i);
}
```

Output:
Compiler specific question. Not all the compilers allow this.

Explanation:
This question deals with the argument passing mechanism. If we call a function, the order in which the arguments of the function are processed is not governed by C Standard. So one compiler can process the arguments from left to right while the other compiler can process them right to left. Usually, the programs are not affected with this because the arguments of the programs are different. For example if we call funtion fun as fun(i,j), then no matter in which order the arguments are processed, the value of i and j will be consistent.

But in this case, we are passing the arguments to function f using the same variable. So the order in which arguments are processed by the function will determine the value of those arguments. cdecl enforces right to left processing of arguments while pascal enforces left to right processing of arguments.

So the value of i, j and k inside the first function f will be 10, 11 and 12 respectively while the value of i, j and k inside the second function f will be 12, 11 and 10 respectively.

GATE CS Corner   Company Wise Coding Practice

Output
C-Output

Output of C Programs | Set 7

Predict the output of below programs

Question 1

```
int main()
{
    int i = 0;
    while (i <= 4)
    {
       printf("%d", i);
       if (i > 3)
        goto inside_foo;
       i++;
    }
    getchar();
    return 0;
}
```

```
void foo()
{
  inside_foo:
    printf("PP");
}
```

Output: Compiler error: Label “inside_foo” used but not defined.

Explanation: Scope of a label is within a function. We cannot goto a label from other function.

Question 2

```
#define a 10
int main()
{
  #define a 50
  printf("%d",a);

  getchar();
  return 0;
}
```

Output: 50

Preprocessor doesn’t give any error if we redefine a preprocessor directive. It may give warning though. Preprocessor takes the most recent value before use of and put it in place of a.

Now try following

```
#define a 10
int main()
{
  printf("%d ",a);
  #define a 50
  printf("%d ",a);

  getchar();
  return 0;
}
```

Question 3

```
int main()
{
    char str[] = "geeksforgeeks";
    char *s1 = str, *s2 = str;
    int i;

    for(i = 0; i < 7; i++)
    {
      printf(" %c ", *str);
      ++s1;
    }

    for(i = 0; i < 6; i++)
    {
      printf(" %c ", *s2);
      ++s2;
     }

     getchar();
     return 0;
}
```

Output

```
g g g g g g g g e e k s f
```

Explanation

Both s1 and s2 are initialized to str. In first loop str is being printed and s1 is being incremented, so first loop will print only g. In second loop s2 is incremented and s2 is printed so second loop will print “g e e k s f ”

Question 4

```
int main()
{
   char str[] = "geeksforgeeks";
   int i;
   for(i=0; str[i]; i++)
      printf("\n%c%c%c%c", str[i], *(str+i), *(i+str), i[str]);

   getchar();
   return 0;
}
```

Output:

```
gggg
eeee
eeee
kkkk
ssss
ffff
oooo
rrrr
gggg
eeee
eeee
kkkk
ssss
```

Explanaition:
Following are different ways of indexing both array and string.

arr[i]
*(arr + i)
*(i + arr)
i[arr]

So all of them print same character.

Question 5

```
int main()
{
   char *p;
   printf("%d %d ", sizeof(*p), sizeof(p));

   getchar();
   return 0;
}
```

Output: Compiler dependent. I got output as “1 4”

Explanation:
Output of the above program depends on compiler. sizeof(*p) gives size of character. If characters are stored as 1 byte then sizeof(*p) gives 1.
sizeof(p) gives the size of pointer variable. If pointer variables are stored as 4 bytes then it gives 4.

GATE CS Corner   Company Wise Coding Practice

Output
C-Output

Output of C programs | Set 8

Predict the output of below C programs.

Question 1:

```
#include<stdio.h>
int main()
{
   int x = 5, p = 10;
   printf("%*d", x, p);

   getchar();
   return 0;
}
```

Output:

```
  10
```

Explanation:
Please see standard printf function definition

```
     int printf ( const char * format, ... );
```

format: String that contains the text to be written to stdout. It can optionally contain embedded format tags that are substituted by the values specified in subsequent argument(s) and formatted as requested. The number of arguments following the format parameters should at least be as much as the number of format tags. The format tags follow this prototype:

```
     %[flags][width][.precision][length]specifier
```

You can see details of all above parts here http://www.cplusplus.com/reference/clibrary/cstdio/printf/.

The main thing to note is below the line about precision
*** (star)**: The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

So, in the above example 5 is precision and 10 is the actual value to be printed. (Thanks to Ajay Mishra for providing the solution)

Question 2

```
int main()
{
  char arr[]  = "geeksforgeeks";
  char *ptr  = arr;

  while(*ptr != '\0')
      ++*ptr++;
  printf("%s %s", arr, ptr);

  getchar();
  return 0;
}
```

Output: hffltgpshfflt

Explanation:
The crust of this question lies in expression ++*ptr++.

If one knows the precedence and associativity of the operators then there is nothing much left. Below is the precedence of operators.

```
  Postfixx ++          left-to-right
  Prefix  ++           right-to-left
  Dereference *         right-to-left
```

Therefore the expression ++*ptr++ has following effect
Value of *ptr is incremented
Value of ptr is incremented

Question 3

```
int main()
{
  signed char i=0;
  for(; i >= 0; i++);
  printf("%d\n", i);

  getchar();
  return 0;
}
```

Output: -128

Explanation:
Here, the first thing to be noticed is the semi-colon in the end of for loop. So basically, there's no body for the for loop. And printf will print the value of i when control comes out of the for loop. Also, notice that the first statement i.e. initializer in the for loop is not present. But i has been initialized at the declaration time. Since i is a signed character, it can take value from -128 to 127. So in the for loop, i will keep incrementing and the condition is met till roll over happens. At the roll over, i will become -128 after 127, in that case condition is not met and control comes out of the for loop.

Question 4

```
#include <stdio.h>
void fun(const char **p) { }
int main(int argc, char **argv)
{
   fun(argv);
   getchar();
   return 0;
}
```

Output: Compiler error.

Explanation:
Parameter passed to fun() and parameter expected in definition are of incompatible types. fun() expects const char** while passed parameter is of type char **.

Now try following program, it will work.

```
void fun(const char **p) { }
int main(int argc, char **argv)
{
   const char **temp;
   fun(temp);
   getchar();
   return 0;
}
```

Output of C Programs | Set 9

Predict the output of the below programs.

Question 1

```
int main()
{
 int c=5;
 printf("%d\n%d\n%d", c, c <<= 2, c >>= 2);
 getchar();
}
```

Output: Compiler dependent
Evaluation order of parameters is not defined by C standard and is dependent on compiler implementation. It is never safe to depend on the order of parameter evaluation. For example, a function call like above may very well behave differently from one compiler to another.

References:
http://gcc.gnu.org/onlinedocs/gcc/Non_002dbugs.html

Question 2

```
int main()
{
   char arr[] = {1, 2, 3};
   char *p = arr;
   if(&p == &arr)
    printf("Same");
   else
    printf("Not same");
   getchar();
}
```

Output: Not Same
&arr is an alias for &arr[0] and returns the address of the first element in array, but &p returns the address of pointer p.
Now try below program

```
int main()
{
   char arr[] = {1, 2, 3};
   char *p = arr;
   if(p == &arr)
    printf("Same");
   else
    printf("Not same");
   getchar();
}
```

Question 3

```
int main()
{
   char arr[] = {1, 2, 3};
   char *p = arr;
   printf(" %d ", sizeof(p));
   printf(" %d ", sizeof(arr));
   getchar();
}
```

Output 4 3
sizeof(arr) returns the amount of memory used by all elements in array
and sizeof(p) returns the amount of memory used by the pointer variable itself.

Question 4

```
int x = 0;
int f()
{
  return x;
}

int g()
{
  int x = 1;
  return f();
}

int main()
```

```
{
 printf("%d", g());
 printf("\n");
 getchar();
}
```

Output: 0

In C, variables are always statically (or lexically) scoped. Binding of x inside f() to global variable x is defined at compile time and not dependent on who is calling it. Hence, output for the above program will be 0.

On a side note, Perl supports both dynamic ans static scoping. Perl's keyword "my" defines a statically scoped local variable, while the keyword "local" defines dynamically scoped local variable. So in Perl, similar (see below) program will print 1.

```
$x = 0;
sub f
{
  return $x;
}
sub g
{
  local $x = 1; return f();
}
print g()."\n";
```

Reference:
http://en.wikipedia.org/wiki/Scope_%28programming%29

Please write comments if you find any of the above answers/explanations incorrect.

GATE CS Corner   Company Wise Coding Practice

Output
C-Output

Output of C Programs | Set 10

Predict the output of the below programs.

Difficulty Level: Rookie

Question 1

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

Output: 0

There is no problem with the program. It simply creates a user defined type *i* and creates a variable *a* of type *i*.

Question 2

```
#include<stdio.h>
int main()
{
 typedef int *i;
 int j = 10;
 i *a = &j;
 printf("%d", **a);
 getchar();
 return 0;
}
```

Output: Compiler Error -> Initialization with incompatible pointer type.
The line *typedef int *i* makes *i* as type int *. So, the declaration of *a* means *a* is pointer to a pointer. The Error message may be different on different compilers.

Question 3

```
#include<stdio.h>
int main()
{
  typedef static int *i;
  int j;
  i a = &j;
  printf("%d", *a);
  getchar();
  return 0;
}
```

Output: Compiler Error -> Multiple Storage classes for a.
In C, typedef is considered as a storage class. The Error message may be different on different compilers.

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above.

References:
http://www.itee.uq.edu.au/~comp2303/Leslie_C_ref/C/SYNTAX/typedef.html
http://publib.boulder.ibm.com/infocenter/macxhelp/v6v81/index.jsp?topic=/com.ibm.vacpp6m.doc/language/ref/clrc03sc03.htm
http://msdn.microsoft.com/en-us/library/4x7sfztk.aspx

GATE CS Corner   Company Wise Coding Practice

Output

Output of C Programs | Set 11

Asked by Shobhit

```
#include<stdio.h>
int fun(int n, int *fg)
{
   int t, f;
   if(n <= 1)
   {
     *fg = 1;
      return 1;
   }
   t = fun(n-1, fg);
   f = t + *fg;
   *fg = t;
   return f;
}
int main( )
{
  int x = 15;
  printf ( "%d\n", fun (5, &x));
  getchar();
  return 0;
}
```

In the above program, there will be recursive calls till n is **not** smaller than or equal to 1.

```
fun(5, &x)
       \
        \
      fun(4, fg)
```

```
         \
          \
         fun(3, fg)
              \
               \
              fun(2, fg)
                 \
                  \
                    fun(1, fg)
```

fun(1, fg) does not further call fun() because n is 1 now, and it goes inside the if part. It changes value at address fg to 1, and returns 1.

Inside fun(2, fg)

```
 t = fun(n-1, fg); --> t = 1
/* After fun(1, fg) is called, fun(2, fg) does following */
 f = t + *fg;     -->  f = 1 + 1 (changed by fun(1, fg)) = 2
 *fg = t;         --> *fg = 1
return f (or return 2)
```

Inside fun(3, fg)

```
 t = fun(2, fg); --> t = 2
/* After fun(2, fg) is called, fun(3, fg) does following */
 f = t + *fg;      -->  f = 2 + 1 = 3
 *fg = t;          --> *fg = 2
return f (or return 3)
```

Inside fun(4, fg)

```
 t = fun(3, fg);   --> t = 3
/* After fun(3, fg) is called, fun(4, fg) does following */
 f = t + *fg;      -->  f = 3 + 2 = 5
 *fg = t;          --> *fg = 3
 return f (or return 5)
```

Inside fun(5, fg)

```
 t = fun(4, fg);   -->  t = 5
/* After fun(4, fg) is called, fun(5, fg) does following */
 f = t + *fg;      -->  f = 5 + 3 = 8
 *fg = t;          --> *fg = 5
 return f (or return 8 )
```

Finally, value returned by fun(5, &x) is printed, so 8 is printed on the screen

Output
C-Output

Output of C Programs | Set 12

Predict the output of below programs.

Question 1

```
int fun(char *str1)
{
  char *str2 = str1;
  while(*++str1);
  return (str1-str2);
}

int main()
{
  char *str = "geeksforgeeks";
  printf("%d", fun(str));
  getchar();
  return 0;
}
```

Output: 13

Inside fun(), pointer str2 is initialized as str1 and str1 is moved till '\0' is reached (note **;** after while loop). So str1 will be incremented by 13 (assuming that char takes 1 byte).

Question 2

```
void fun(int *p)
{
  static int q = 10;
  p = &q;
}

int main()
{
  int r = 20;
  int *p = &r;
  fun(p);
  printf("%d", *p);
  getchar();
  return 0;
}
```

Output: 20

Inside fun(), q is a copy of the pointer p. So if we change q to point something else then p remains unaffected.

Question 3

```
void fun(int **p)
{
  static int q = 10;
  *p = &q;
}

int main()
{
  int r = 20;
  int *p = &r;
  fun(&p);
  printf("%d", *p);
  getchar();
  return 0;
}
```

Output 10

Note that we are passing address of p to fun(). p in fun() is actually a pointer to p in main() and we are changing value at p in fun(). So p of main is changed to point q of fun(). To understand it better, let us rename p in fun() to p_ref or ptr_to_p

```
void fun(int **ptr_to_p)
{
  static int q = 10;
  *ptr_to_p = &q;  /*Now p of main is pointing to q*/
}
```

Also, note that the program won't cause any problem because q is a static variable. Static variables exist in memory even after functions return. For an auto variable, we might have seen some weird output because auto variable may not exist in memory after functions return.

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above.

Output
C-Output

Output of C Programs | Set 13

Difficulty Level: Rookie

Question 1

Predict the output of below program.

```
int main()
{
  char arr[] = "geeksforgeeks";
  printf("%d", sizeof(arr));
  getchar();
  return 0;
}
```

Output: 14

The string “geeksforgeeks” has 13 characters, but the size is 14 because compiler includes a single ‘\0’ (string terminator) when char array size is not explicitly mentioned.

Question 2

In below program, what would you put in place of “?” to print “geeks”. Obviously, something other than “geeks”.

```
int main()
{
  char arr[] = "geeksforgeeks";
  printf("%s", ?);
  getchar();
  return 0;
}
```

Answer: (arr+8)

The printf statement prints everything starting from arr+8 until it finds ‘\0’

Question 3

Predict the output of below program.

```
int main()
{
  int x, y = 5, z = 5;
  x = y==z;
  printf("%d", x);

  getchar();
  return 0;
}
```

The crux of the question lies in the statement x = y==z. The operator == is executed before = because precedence of comparison operators (= and ==) is higher than assignment operator =.

The result of a comparison operator is either 0 or 1 based on the comparison result. Since y is equal to z, value of the expression y == z becomes 1 and the value is assigned to x via the assignment operator.

Question 4

Predict the output of below program.

```
int main()
{
  printf(" \"GEEKS %% FOR %% GEEKS\"");
  getchar();
  return 0;
}
```

Output: “GEEKS % FOR % GEEKS”

Backslash (\) works as escape character for double quote (“). For explanation of %%, please see this.

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above

GATE CS Corner   Company Wise Coding Practice

Output
C-Output

Result of sizeof operator

Asked by Kapil

Predict the output of below program.

```
#include <stdio.h>
#define TOTAL_ELEMENTS (sizeof(array) / sizeof(array[0]))
int array[] = {1, 2, 3, 4, 5, 6, 7};

int main()
{
 int i;

 for(i = -1; i <= (TOTAL_ELEMENTS-2); i++)
   printf("%d\n", array[i+1]);

 getchar();
 return 0;
}
```

Output: Nothing is printed as loop condition is not true for the first time itself.

The result of sizeof for an array operand is number of elements in the array multiplied by size of an element in bytes. So value of the expression TOTAL_ELEMENTS in the above program is 7.
The data type of the sizeof result is unsigned int or unsigned long depending upon the compiler implementation. Therefore, in the loop condition i
Please write comments if you find anything incorrect in the above post.

GATE CS Corner   Company Wise Coding Practice

Output
C-Output

Output of C Programs | Set 14

Predict the output of below C programs.

Question 1

```
#include<stdio.h>
int main()
{
   int a;
   char *x;
   x = (char *) &a;
   a = 512;
   x[0] = 1;
   x[1] = 2;
   printf("%d\n",a);

   getchar();
   return 0;
}
```

Answer: The output is dependent on endianness of a machine. Output is 513 in a little endian machine and 258 in a big endian machine.

Let integers are stored using 16 bits. In a little endian machine, when we do x[0] = 1 and x[1] = 2, the number a is changed to 00000001 00000010 which is representation of 513 in a little endian machine. The output would be same for 32 bit numbers also.

In a big endian machine, when we do x[0] = 1 and x[1] = 2, the number is changed to 00000001 00000010 which is representation of 258 in a big endian machine.

Question 2

```
int main()
{
  int f = 0, g = 1;
  int i;
  for(i = 0; i < 15; i++)
  {
    printf("%d \n", f);
    f = f + g;
    g = f - g;
  }
  getchar();
  return 0;
}
```

Answer: The function prints first 15 Fibonacci Numbers.

Question 3
Explain functionality of following function.

```
int func(int i)
{
  if(i%2) return (i++);
  else return func(func(i-1));
}
```

Answer: If n is odd then returns n, else returns (n-1). So if n is 12 then we get and if n is 11 then we get 11.

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above.

GATE CS Corner   Company Wise Coding Practice

Output
C-Output

Output of C Programs | Set 15

Predict the output of below C programs.

Question 1

```
#include<stdio.h>
int main(void)
{
  int a = 1;
  int b = 0;
  b = ++a + ++a;
  printf("%d %d",a,b);
  getchar();
  return 0;
}
```

Output: Undefined Behavior
See http://en.wikipedia.org/wiki/C_syntax#Undefined_behavior)

Question 2

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

```
int main()
{
  int a[] = {1, 2, 3, 4, 5, 6};
  int *ptr = (int*)(&a+1);
  printf("%d ", *(ptr-1) );
  getchar();
  return 0;
}
```

Output: 6
&a is address of the whole array *a[]*. If we add 1 to *&a*, we get "base address of a[] + sizeof(a)". And this value is typecasted to int *. So *ptr – 1* points to last element of a[]

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above.

GATE CS Corner   Company Wise Coding Practice

Output
C-Output

Output of C Programs | Set 16

Predict the output of below C programs.

Question 1

```
#include <stdio.h>

char* fun()
{
  return "awake";
}
int main()
{
  printf("%s",fun()+ printf("I see you"));
  getchar();
  return 0;
}
```

Output: Some string starting with "I see you"
Explanation: (Thanks to Venki for suggesting this solution)
The function fun() returns pointer to char. Apart from printing string "I see you", printf() function returns number of characters it printed(i.e. 9). The expression [fun()+ printf("I see you")] can be boiled down to ["awake" + 9] which is nothing but base address of string literal "awake" displaced by 9 characters. Hence, the expression ["awake" + 9] returns junk data when printed via %s specifier till it finds '\0'.

Question 2

```
#include <stdio.h>

int main()
{
  unsigned i ;
  for( i = 0 ; i < 4 ; ++i )
  {
    fprintf( stdout , "i = %d\n" , ("11213141") ) ;
  }

  getchar();
  return 0 ;
}
```

Output: Prints different output on different machines.
Explanation: (Thanks to Venki for suggesting this solution)
The format specifier is %d, converts the base address of string "11213141" as an integer. The base address of string depends on memory

allocation by the compiler. The for loop prints same address four times. Try to use C++ streams, you will see power of type system.

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above

Output
C-Output

Output of C++ Program | Set 1

Predict the output of below C++ programs.

Question 1

```
// Assume that integers take 4 bytes.
#include<iostream>

using namespace std;

class Test
{
  static int i;
  int j;
};

int Test::i;

int main()
{
    cout << sizeof(Test);
    return 0;
}
```

Output: 4 (size of integer)
static data members do not contribute in size of an object. So 'i' is not considered in size of Test. Also, all functions (static and non-static both) do not contribute in size.

Question 2

```
#include<iostream>

using namespace std;
class Base1 {
 public:
     Base1()
     { cout << " Base1's constructor called" << endl;  }
};

class Base2 {
 public:
     Base2()
     { cout << "Base2's constructor called" << endl;  }
};

class Derived: public Base1, public Base2 {
   public:
     Derived()
     {  cout << "Derived's constructor called" << endl;  }
};

int main()
{
   Derived d;
   return 0;
}
```

Ouput:
Base1’s constructor called
Base2’s constructor called
Derived’s constructor called

In case of Multiple Inheritance, constructors of base classes are always called in derivation order from left to right and Destructors are called in reverse order.

GATE CS Corner   Company Wise Coding Practice

Output
CPP-Output

Output of C Program | Set 17

Predict the output of following C programs.

Question 1

```
#include<stdio.h>

#define R 10
#define C 20

int main()
{
  int (*p)[R][C];
  printf("%d",  sizeof(*p));
  getchar();
  return 0;
}
```

Output: 10*20*sizeof(int) which is “800” for compilers with integer size as 4 bytes.
The pointer p is de-referenced, hence it yields type of the object. In the present case, it is an array of array of integers. So, it prints R*C*sizeof(int).
Thanks to Venki for suggesting this solution.

Question 2

```
#include<stdio.h>
#define f(g,g2) g##g2
int main()
{
  int var12 = 100;
  printf("%d", f(var,12));
  getchar();
  return 0;
}
```

Output: 100
The operator ## is called “Token-Pasting” or “Merge” Operator. It merges two tokens into one token. So, after preprocessing, the main function becomes as follows, and prints 100.

```
int main()
{
  int var12 = 100;
  printf("%d", var12);
  getchar();
  return 0;
}
```

Question 3

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

```
int main()
{
   unsigned int x = -1;
   int y = ~0;
   if(x == y)
      printf("same");
   else
      printf("not same");
   printf("\n x is %u, y is %u", x, y);
   getchar();
   return 0;
}
```

Output: “same x is MAXUINT, y is MAXUINT” Where MAXUINT is the maximum possible value for an unsigned integer.
-1 and ~0 essentially have same bit pattern, hence x and y must be same. In the comparison, y is promoted to unsigned and compared against x. The result is “same”. However, when interpreted as signed and unsigned their numerical values will differ. x is MAXUNIT and y is -1. Since we have %u for y also, the output will be MAXUNIT and MAXUNIT.
Thanks to Venki for explanation.

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above

GATE CS Corner   Company Wise Coding Practice

Output
C-Output

Output of C Program | Set 18

Predict the output of following C programs.

Question 1

```
#include<stdio.h>
int fun()
{
  static int num = 40;
  return num--;
}

int main()
{
  for(fun(); fun(); fun())
  {
    printf("%d ", fun());
  }
  getchar();
  return 0;
}
```

Output: 38 35 32 29 26 23 20 17 14 11 8 5 2

Since *num* is static in fun(), the old value of *num* is preserved for subsequent functions calls. Also, since the statement *return num–* is postfix, it returns the old value of *num*, and updates the value for next function call.

Question 2

```
#include<stdio.h>
int main()
{
  char *s[] = { "knowledge","is","power"};
  char **p;
  p = s;
  printf("%s ", ++*p);
  printf("%s ", *p++);
  printf("%s ", ++*p);

  getchar();
  return 0;
}
```

Output: nowledge nowledge s

Let us consider the expression *++*p* in first printf(). Since precedence of prefix ++ and * is same, associativity comes into picture. **p* is evaluated first because both prefix ++ and * are right to left associative. When we increment **p* by 1, it starts pointing to second character of *"knowledge"*. Therefore, the first printf statement prints *"nowledge"*.

Let us consider the expression **p++* in second printf() . Since precedence of postfix ++ is higher than *, *p++* is evaluated first. And since it's a psotfix ++, old value of *p* is used in this expression. Therefore, second printf statement prints *"nowledge"*.

In third printf statement, the new value of *p* (updated by second printf) is used, and third printf() prints *"s"*.

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above

GATE CS Corner   Company Wise Coding Practice

Output
C-Output

Output of C Program | Set 19

Predict the outputs of following program.

Difficulty Level: Rookie

Question 1

```
#include <stdio.h>
int main()
{
  int a = 10, b = 20, c = 30;
  if (c > b > a)
  {
    printf("TRUE");
  }
  else
  {
    printf("FALSE");
  }
  getchar();
  return 0;
}
```

Output: *FALSE*
Let us consider the condition inside the if statement. Since there are two greater than (>) operators in expression "c > b > a", associativity of > is considered. Associativity of > is left to right. So, expression c > b > a is evaluated as ((c > b) > a) which is false.

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above

GATE CS Corner   Company Wise Coding Practice

Output
C-Output

Output of C Program | Set 20

Predict the outputs of following C programs.

Question 1

```
int main()
```

```
{
  int x = 10;
  static int y = x;

  if(x == y)
     printf("Equal");
  else if(x > y)
     printf("Greater");
  else
     printf("Less");

  getchar();
  return 0;
}
```

Output: Compiler Error

In C, static variables can only be initialized using constant literals. See this GFact for details.

Question 2

```
#include <stdio.h>

int main()
{
  int i;

  for (i = 1; i != 10; i += 2)
  {
    printf(" GeeksforGeeks ");
  }

  getchar();
  return 0;
}
```

Output: Infinite times GeeksforGeeks

The loop termination condition never becomes true and the loop prints GeeksforGeeks infinite times. In general, if a *for* or *while* statement uses a loop counter, then it is safer to use a relational operator (such as this for details.

Question 3

```
#include<stdio.h>
struct st
{
    int x;
    struct st next;
};

int main()
{
    struct st temp;
    temp.x = 10;
    temp.next = temp;
    printf("%d", temp.next.x);

    getchar();
    return 0;
}
```

Output: Compiler Error

A C structure cannot contain a member of its own type because if this is allowed then it becomes impossible for compiler to know size of such struct. Although a pointer of same type can be a member because pointers of all types are of same size and compiler can calculate size of struct.

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above

Output of C Program | Set 21

Predict the output of following C programs.

Question 1

```
#include<stdio.h>
#define fun (x) (x)*10

int main()
{
   int t = fun(5);
   int i;
   for(i = 0; i < t; i++)
     printf("GeesforGeeks\n");

   return 0;
}
```

Output: Compiler Error

There is an extra space in macro declaration which causes fun to be raplaced by (x). If we remove the extra space then program works fine and prints "GeeksforGeeks" 50 times. Following is the working program.

```
#include<stdio.h>
#define fun(x) (x)*10

int main()
{
   int t = fun(5);
   int i;
   for(i = 0; i < t; i++)
     printf("GeesforGeeks\n");

   return 0;
}
```

Be careful when dealing with macros. Extra spaces may lead to problems.

Question 2

```
#include<stdio.h>
int main()
{
   int i = 20,j;
   i = (printf("Hello"), printf(" All Geeks "));
   printf("%d", i);

   return 0;
}
```

Output: Hello All Geeks 11

The printf() function returns the number of characters it has successfully printed. The comma operator evaluates it operands from left to right and returns the value returned by the rightmost expression (See this for more details). First *printf("Hello")* executes and prints *"Hello"*, the *printf(" All Geeks ")* executes and prints *" All Geeks "*. This printf statement returns 11 which is assigned to i.

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above

Output of C Program | Set 22

Predict the output of following C programs.

Question 1

```
#include<stdio.h>

int main()
{
    enum channel {star, sony, zee};
    enum symbol {hash, star};

    int i = 0;
    for(i = star; i <= zee; i++)
    {
        printf("%d ", i);
    }

    return 0;
}
```

Output:

```
compiler error: redeclaration of enumerator 'star'
```

In the above program, enumartion constant *'star'* appears two times in main() which causes the error. An enumaration constant must be unique within the scope in which it is defined. The following program works fine and prints 0 1 2 as the enumaration constants automatically get the values starting from 0.

```
#include<stdio.h>

int main()
{
    enum channel {star, sony, zee};

    int i = 0;
    for(i = star; i <= zee; i++)
    {
        printf("%d ", i);
    }

    return 0;
}
```

Output:

```
0 1 2
```

Question 2

```
#include<stdio.h>

int main()
{
    int i, j;
    int p = 0, q = 2;

    for(i = 0, j = 0; i < p, j < q; i++, j++)
    {
      printf("GeeksforGeeks\n");
    }

    return 0;
}
```

Output:

```
GeeksforGeeks
GeeksforGeeks
```

Following is the main expression to consider in the above program.

```
i < p, j < q
```

When two expressions are separated by comma operator, the first expression (i this for more details.

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above

GATE CS Corner   Company Wise Coding Practice

Output
C-Output

Output of C Program | Set 23

Predict the output of following C Program.

```
#include <stdio.h>
#define R 4
#define C 4

void modifyMatrix(int mat[][C])
{
  mat++;
  mat[1][1] = 100;
  mat++;
  mat[1][1] = 200;
}

void printMatrix(int mat[][C])
{
   int i, j;
   for (i = 0; i < R; i++)
   {
      for (j = 0; j < C; j++)
         printf("%3d ", mat[i][j]);
      printf("\n");
   }
}

int main()
{
   int mat[R][C] = { {1, 2, 3, 4},
      {5, 6, 7, 8},
      {9, 10, 11, 12},
      {13, 14, 15, 16}
   };
   printf("Original Matrix \n");
   printMatrix(mat);

   modifyMatrix(mat);

   printf("Matrix after modification \n");
   printMatrix(mat);

   return 0;
}
```

Output: The program compiles fine and produces following output:

```
Original Matrix
  1   2   3   4
  5   6   7   8
  9  10  11  12
 13  14  15  16
Matrix after modification
  1   2   3   4
  5   6   7   8
  9 100  11  12
 13 200  15  16
```

At first look, the line *"mat++;"* in *modifyMatrix()* seems invalid. But this is a valid C line as array parameters are always pointers (see this and this for details). In *modifyMatrix()*, *mat* is just a pointer that points to block of size *C*sizeof(int)*. So following function prototype is same as *"void modifyMatrix(int mat[][C])"*

```
void modifyMatrix(int (*mat)[C]);
```

When we do *mat++, mat* starts pointing to next row, and *mat[1][1]* starts referring to value 10. *mat[1][1]* (value 10) is changed to 100 by the statement *"mat[1][1] = 100;"*. *mat* is again incremented and *mat[1][1]* (now value 14) is changed to 200 by next couple of statements in *modifyMatrix()*.

The line *"mat[1][1] = 100;"* is valid as pointer arithmetic and array indexing are equivalent in C.

On a side note, we can't do *mat++* in *main()* as *mat* is 2 D array in *main()*, not a pointer.

Please write comments if you find above answer/explanation incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

Output
C-Output
Output of C Program

Output of C Program | Set 24

Predict the output of following C programs:

Difficulty Level: Rookie

Question 1

```
#include<stdio.h>
int main()
{
    int arr[] = {10, 20, 30, 40, 50, 60};
    int *ptr1 = arr;
    int *ptr2 = arr + 5;
    printf ("ptr2 - ptr1 = %d\n", ptr2 - ptr1);
    printf ("(char*)ptr2 - (char*) ptr1 = %d",  (char*)ptr2 - (char*)ptr1);
    getchar();
    return 0;
}
```

Output:

```
 ptr2 - ptr1 = 5
 (char*)ptr2 - (char*) ptr1 = 20
```

In C, array name gives address of the first element in the array. So when we do ptr1 = arr, ptr1 starts pointing to address of first element of arr. Since array elements are accessed using pointer arithmetic, arr + 5 is a valid expression and gives the address of 6th element. Predicting value ptr2 – ptr1 is easy, it gives 5 as there are 5 inetegers between these two addresses. When we do (char *)ptr2, ptr2 is typecasted to char pointer. In expression "(int*)ptr2 – (int*)ptr1", pointer arithmetic happens considering character poitners. Since size of a character is one byte, we get 5*sizeof(int) (which is 20) as difference of two pointers.
As an excercise, predict the output of following program.

```
#include<stdio.h>
int main()
{
   char arr[] = "geeksforgeeks";
   char *ptr1 = arr;
   char *ptr2 = ptr1 + 3;
   printf ("ptr2 - ptr1 = %d\n", ptr2 - ptr1);
   printf ("(int*)ptr2 - (int*) ptr1 = %d",  (int*)ptr2 - (int*)ptr1);
   getchar();
   return 0;
}
```

Question 2

```
#include<stdio.h>

int main()
{
  char arr[] = "geeks\0 for geeks";
  char *str = "geeks\0 for geeks";
  printf ("arr = %s, sizeof(arr) = %d \n", arr, sizeof(arr));
  printf ("str = %s, sizeof(str) = %d", str, sizeof(str));
  getchar();
  return 0;
}
```

Output:

```
  arr = geeks, sizeof(arr) = 17
  str = geeks, sizeof(str) = 4
```

Let us first talk about first output *"arr = geeks"*. When %s is used to print a string, printf starts from the first character at given address and keeps printing characters until it sees a string termination character, so we get *"arr = geeks"* as there is a \0 after geeks in arr[].
Now let us talk about output *"sizeof(arr) = 17"*. When a character array is initialized with a double quoted string and array size is not specified, compiler automatically allocates one extra space for string terminator '\0' (See this Gfact), that is why size of arr is 17.
Explanation for printing "str = geeks" is same as printing "arr = geeks". Talking about value of sizeof(str), str is just a pointer (not array), so we get size of a pointer as output.

Please write comments if you find above answer/explanation incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

Output
C-Output

Output of C program | Set 25

Predict the output of following C program.

```
int main(void)
{
   struct str
   {
      int i: 1;
      int j: 2;
      int k: 3;
      int l: 4;
   };

   struct str s;

   s.i = 1;
   s.j = 2;
   s.k = 5;
```

```
    s.l = 10;

    printf(" i: %d \n j: %d \n k: %d \n l: %d \n", s.i, s.j, s.k, s.l);

    getchar();
    return 0;
}
```

The above code is non-portable and output is compiler dependent. We get following output using GCC compiler for intel 32 bit machine.

```
 [narendra@ubuntu]$ ./structure
 i: -1
 j: -2
 k: -3
 l: -6
```

Let us take a closer look at declaration of structure.

```
struct str
{
    int i: 1;
    int j: 2;
    int k: 3;
    int l: 4;
};
```

In the structure declaration, for structure member 'i', we used width of bit field as 1, width of 'j' as 2, and so on. At first, it looks we can store values in range [0-1] for 'i', range [0-3] for 'j', and so on. But in the above declaration, type of bit fields is integer (**signed**). That's why out of available bits, 1 bit is used to store sign information. So for 'i', the values we can store are 0 or -1 (for a machine that uses two's complement to store signed integers). For variable 'k', number of bits is 3. Out of these 3 bits, 2 bits are used to store data and 1 bit is used to store sign.

Let use declare structure members as "unsigned int" and check output.

```
int main(void)
{
    struct str
    {
        unsigned int i: 1;
        unsigned int j: 2;
        unsigned int k: 3;
        unsigned int l: 4;
    };
    struct str s;

    s.i = 1;
    s.j = 2;
    s.k = 5;
    s.l = 10;

    printf(" i: %d \n j: %d \n k: %d \n l: %d \n", s.i, s.j, s.k, s.l);

    getchar();
    return 0;
}
```

output:

```
 [narendra@ubuntu]$ ./structure
 i: 1
 j: 2
 k: 5
 l: 10
```

This article is compiled by "Narendra Kangralkar" and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Output of C Program | Set 26

Predict the output of following C programs.

Question 1

```
#include <stdio.h>

int main()
{
  int arr[] = {};
  printf("%d", sizeof(arr));
  return 0;
}
```

Output: 0

C (or C++) allows arrays of size 0. When an array is declared with empty initialization list, size of the array becomes 0.

Question 2

```
#include<stdio.h>

int main()
{
  int i, j;
  int arr[4][4] = { {1, 2, 3, 4},
                    {5, 6, 7, 8},
                    {9, 10, 11, 12},
                    {13, 14, 15, 16} };
  for(i = 0; i < 4; i++)
    for(j = 0; j < 4; j++)
      printf("%d ", j[i[arr]] );

  printf("\n");

  for(i = 0; i < 4; i++)
    for(j = 0; j < 4; j++)
      printf("%d ", i[j[arr]] );

  return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1 5 9 13 2 6 10 14 3 7 11 15 4 8 12 16
```

Array elements are accessed using pointer arithmetic. So the meaning of arr[i][j] and j[i[arr]] is same. They both mean (arr + 4*i + j). Similarly, the meaning of arr[j][i] and i[j[arr]] is same.

Question 3

```
#include<stdio.h>
int main()
{
    int a[2][3] = {2,1,3,2,3,4};
    printf("Using pointer notations:\n");
    printf("%d %d %d\n", *(*(a+0)+0), *(*(a+0)+1), *(*(a+0)+2));
    printf("Using mixed notations:\n");
    printf("%d %d %d\n", *(a[1]+0), *(a[1]+1), *(a[1]+2));
    return 0;
}
```

Output:

```
Using pointer notations:
2 1 3
Using mixed notations:
2 3 4
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

Output
C-Output

Output of C Program | Set 27

Predict the output of following C programs.

Question 1

```
#include <stdio.h>

int main(void)
{
 int i;
 int power_of_ten[5] = {
       00001,
       00010,
       00100,
       01000,
       10000,
      };

 for (i = 0; i < 5; ++i)
  printf("%d ", power_of_ten[i]);
 printf("\n");

 return 0;
}
```

In the above example, we have created an array of 5 elements, whose elements are power of ten (i.e. 1, 10, 100, 1000 and 10,000) and we are printing these element by using a simple for loop. So we are expecting output of above program is " 1 10 100 1000 and 10000″, but above program doesn't show this output, instead it shows

```
"1 8 64 512 10000"
```

Let us discuss above program in more detail. Inside array we declared elements which starts with "0", and this is octal representation of decimal number (See this GFact for details).
That's why all these numbers are in octal representation. "010" is octal representation of decimal "8", "0100" is octal representation of decimal "64" and so on.
Last element "10000" which doesn't start with "0", that's why it is in decimal format and it is printed as it is.

Question 2

```
#include <stdio.h>

int main(void)
{
    http://geeksforgeeks.org

    printf("Hello, World !!!\n");

    return 0;
}
```

At first sight, it looks like that the above program will give compilation error, but it will work fine (but will give compilation warning). Observe the above program carefully, in this program "http:" will be treated as goto label and two forward slashes will act as single line comment. That's why program will work work correctly.

Question 3

```
#include <stdio.h>

#define ARRAY_SIZE(arr) (sizeof(arr) / sizeof(arr[0]))

int main(void)
{
 int i;
 int arr[] = {1, 2, 3, 4, 5};

 for (i = -1; i < ARRAY_SIZE(arr) - 1; ++i)
  printf("%d ", arr[i + 1]);

 printf("\n");

 return 0;
}
```

In above program, "ARRAY_SIZE" macro substitutes the expression to get the total number of elements present in array, and we are printing these element by using a for loop. But program doesn't print anything. Let us see what is wrong with code.
Value returned by the substituted expression is in "unsigned int" format, and inside for loop we are comparing unsigned 5 with signed -1. In this comparison -1 is promoted to unsigned integer. -1 in unsigned format is represented as all it's bit are set to 1 (i.e. 0xffffffff), which is too big number

After substituting value of macro, for loop look like this.

```
for (i = -1; 0xffffffff
In above loop indeed 0xffffffff is not less than 5, that's why for loop condition fails, and program exits from loop without printing anything.
```

This article is compiled by **Narendra Kangralkar**. Please write comments if you find anything incorrect, or you want to share more information about

GATE CS Corner

Company Wise Coding Practice

Output
C-Output

Output of C Program | Set 28

Predict the output of following C program.

Question 1

```
#include <stdio.h>

int main()
{
   char a = 30;
   char b = 40;
   char c = 10;
   char d = (a * b) / c;
   printf ("%d ", d);

   return 0;
}
```

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. For example, the following code snippet prints -80 on a 32 bit little endian machine.

```
   char d = 1200;
   printf ("%d ", d);
```

Arithmetic overflow doesn't happen in the original program and the output of the program is 120. In C, *char* and *short* are converted to *int* for arithmetic calculations. So in the expression '(a*b)/c', a, b and c are promoted to *int* and no overflow happens.

Question 2

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

Output:

```
a = -10, b = 9
```

The statement 'a = -b–;' compiles fine. Unary minus and unary decrement have save precedence and right to left associativity. Therefore '-b–' is treated as -(b–) which is valid. So -10 will be assigned to 'a', and 'b' will become 9.
Try the following program as an exercise.

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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

Output
C-Output

Question 1
Predict the output of following program?

```
#include "stdio.h"
int main()
{
    char arr[100];
    printf("%d", scanf("%s", arr));
    /* Suppose that input value given
       for above scanf is "GeeksQuiz" */
    return 1;
}
```

A9
B1
C10
D100
Input and Output
Discuss it
Question 1 Explanation:
In C, scanf returns the no. of inputs it has successfully read. See http://geeksforgeeks.org/archives/674
Question 2
Predict output of the following program

```
#include <stdio.h>
int main()
{
   printf("\new_c_question\by");
   printf("\rgeeksforgeeks");
   getchar();
   return 0;
}
```

Aew_c_question
  geeksforgeeks
Bnew_c_ques
  geeksforgeeks
C
  geeksforgeeks
DDepends on terminal configuration
Input and Output
Discuss it
Question 2 Explanation:
See http://stackoverflow.com/questions/17236242/usage-of-b-and-r-in-c
Question 3

```
#include <stdio.h>

int main()
{
  printf(" \"GEEKS %% FOR %% GEEKS\"");
  getchar();
  return 0;
}
```

A“GEEKS % FOR % GEEKS”
BGEEKS % FOR % GEEKS
C\"GEEKS %% FOR %% GEEKS\"
DGEEKS %% FOR %% GEEKS
Input and Output
Discuss it
Question 3 Explanation:
Backslash (\\\\) works as escape character for double quote (“). For explanation of %%, see http://www.geeksforgeeks.org/how-to-print-using-printf/
Question 4

```
#include <stdio.h>
// Assume base address of "GeeksQuiz" to be 1000
int main()
{
    printf(5 + "GeeksQuiz");
    return 0;
}
```

AGeeksQuiz
BQuiz
C1005
DCompile-time error
Input and Output
Discuss it
Question 4 Explanation:
printf is a library function defined under *stdio.h* header file. The compiler adds 5 to the base address of the string through the expression ***5 + "GeeksQuiz"*** . Then the string "Quiz" gets passed to the standard library function as an argument.

Question 5
Predict the output of the below program:

```
#include <stdio.h>

int main()
{
    printf("%c ", 5["GeeksQuiz"]);
    return 0;
}
```

ACompile-time error
BRuntime error
CQ
Ds
Input and Output
Discuss it
Question 5 Explanation:
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.
Question 6
Predict the output of below program:

```
#include <stdio.h>
int main()
{
    printf("%c ", "GeeksQuiz"[5]);
    return 0;
}
```

ACompile-time error
BRuntime error
CQ
Ds
Input and Output
Discuss it
Question 6 Explanation:
The crux of the program lies in the expression: **"GeeksQuiz"[5]**. This expression is broken down by the compiler as: *(“GeeksQuiz” + 5). 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.
Question 7
Which of the following is true
Agets() can read a string with newline chacters but a normal scanf() with %s can not.
Bgets() can read a string with spaces but a normal scanf() with %s can not.
Cgets() can always replace scanf() without any additional code.
DNone of the above
Input and Output
Discuss it
Question 7 Explanation:
gets() can read a string with spaces but a normal scanf() with %s can not. Consider following program as an example. If we enter "Geeks Quiz" as input in below program, the program prints "Geeks" 1 But in the following program, if we enter "Geeks Quiz", it prints "Geeks Quiz" 1
Question 8
Which of the following is true
Agets() doesn't do any array bound testing and should not be used.

Bfgets() should be used in place of gets() only for files, otherwise gets() is fine
Cgets() cannot read strings with spaces
DNone of the above
Input and Output
Discuss it
Question 8 Explanation:
See gets() is risky to use!
Question 9
What does the following C statement mean?

```
scanf("%4s", str);
```

ARead exactly 4 characters from console.
BRead maximum 4 characters from console.
CRead a string str in multiples of 4
DNothing
Input and Output
Discuss it
Question 9 Explanation:
Try following program, enter GeeksQuiz, the output would be "Geek"

```
#include <stdio.h>

int main()
{
   char str[50] = {0};
   scanf("%4s", str);
   printf(str);
   getchar();
   return 0;
}
```

Question 10

```
#include<stdio.h>

int main()
{
   char *s = "Geeks Quiz";
   int n = 7;
   printf("%.*s", n, s);
   return 0;
}
```

AGeeks Quiz
BNothing is printed
CGeeks Q
DGeeks Qu
Input and Output
Discuss it
Question 10 Explanation:
.* 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.
Question 11
Predict the output of following program?

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

AGeeksQuiz9
BGeeksQuiz10
CGeeksQuizGeeksQuiz
DGeeksQuiz1
Input and Output
Discuss it

Question 11 Explanation:
The printf function returns the number of characters successfully printed on the screen. The string "GeeksQuiz" has 9 characters, so the first printf prints GeeksQuiz and returns 9.
Question 12
Output of following program?

```
#include<stdio.h>
int main()
{
   printf("%d", printf("%d", 1234));
   return 0;
}
```

A12344
B12341
C11234
D41234
Input and Output
Discuss it
Question 12 Explanation:
printf() returns the number of characters successfully printed on the screen.
Question 13
What is the return type of getchar()?
Aint
Bchar
Cunsigned char
Dfloat
Input and Output
Discuss it
Question 13 Explanation:
The return type of getchar() is int to accommodate EOF which indicates failure:
Question 14
Normally user programs are prevented from handling I/O directly by I/O instructions in them. For CPUs having explicit I/O instructions, such I/O protection is ensured by having the I/O instructions privileged. In a CPU with memory mapped I/O, there is no explicit I/O instruction. Which one of the following is true for a CPU with memory mapped I/O?
AI/O protection is ensured by operating system routine (s)
BI/O protection is ensured by a hardware trap
CI/O protection is ensured during system configuration
DI/O protection is not possible
Input and Output **GATE-CS-2005**
Discuss it
Question 14 Explanation:
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.
Question 15
Which of the following functions from “stdio.h” can be used in place of **printf()**?
Afputs() with FILE stream as stdout.
Bfprintf() with FILE stream as stdout.
Cfwrite() with FILE stream as stdout.
DAll of the above three - a, b and c.
EIn “stdio.h”, there’s no other equivalent function of printf().
Input and Output **C Quiz - 103**
Discuss it
Question 15 Explanation:
Though fputs() and fwrite() can accept FILE stream stdout and can output the given string but the input string wouldn’t result in formatted (i.e. containing format specifiers) output. But fprintf() can be used for formatted output. That’s why *fprintf(stdout,"=%d=",a);* and *printf("=%d=",a);* both are equivalent. The correct answer is B.
There are 15 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Question 1

Predict the output of following program. Assume that the numbers are stored in 2's complement form.

```
#include<stdio.h>
int  main()
{
   unsigned int x = -1;
   int y = ~0;
   if (x == y)
      printf("same");
   else
      printf("not same");
   return 0;
}
```

Asame
Bnot same
Data Types
Discuss it
Question 1 Explanation:
-1 and ~0 essentially have same bit pattern, hence x and y must be same. In the comparison, y is promoted to unsigned and compared against x (See this for promotion rules). The result is “same”. However, when interpreted as signed and unsigned their numerical values will differ. x is MAXUNIT and y is -1. Since we have %u for y also, the output will be MAXUNIT and MAXUNIT.
Question 2
Which of the following is not a valid declaration in C?

```
1. short int x;
```

```
2. signed short x;
```

```
3. short x;
```

```
4. unsigned short x;
```

A3 and 4
B2
C1
DAll are valid
Data Types
Discuss it
Question 2 Explanation:
All are valid. First 3 mean the same thing. 4th means unsigned.
Question 3
Predict the output

```
#include <stdio.h>

int main()
{
   float c = 5.0;
   printf ("Temperature in Fahrenheit is %.2f", (9/5)*c + 32);
   return 0;
}
```

ATemperature in Fahrenheit is 41.00
BTemperature in Fahrenheit is 37.00
CTemperature in Fahrenheit is 0.00
DCompiler Error
Data Types
Discuss it
Question 3 Explanation:
Since 9 and 5 are integers, integer arithmetic happens in subexpression (9/5) and we get 1 as its value. To fix the above program, we can use 9.0 instead of 9 or 5.0 instead of 5 so that floating point arithmetic happens. 1
Question 4
Predict the output of following C program

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

```
{
    char a = '\012';

    printf("%d", a);

    return 0;
}
```

ACompiler Error
B12
C10
DEmpty
Data Types
Discuss it
Question 4 Explanation:
The value '\012' means the character with value 12 in octal, which is decimal 10
Question 5
In C, sizes of an integer and a pointer must be same.
ATrue
BFalse
Data Types
Discuss it
Question 5 Explanation:
Sizes of integer and pointer are compiler dependent. The both sizes need not be same.
Question 6
Output?

```
int main()
{
    void *vptr, v;
    v = 0;
    vptr = &v;
    printf("%v", *vptr);
    getchar();
    return 0;
}
```

A0
BCompiler Error
CGarbage Value
Data Types
Discuss it
Question 6 Explanation:
void is not a valid type for declaring variables. void * is valid though.
Question 7
Assume that the size of char is 1 byte and negatives are stored in 2's complement form

```
#include<stdio.h>
int main()
{
    char c = 125;
    c = c+10;
    printf("%d", c);
    return 0;
}
```

A135
B+INF
C-121
D-8
Data Types
Discuss it
Question 8

```
#include <stdio.h>
int main()
{
    if (sizeof(int) > -1)
        printf("Yes");
    else
        printf("No");
    return 0;
}
```

AYes
BNo
CCompiler Error
DRuntime Error
Data Types
Discuss it
Question 8 Explanation:
In C, when an intger value is compared with an unsigned it, the int is promoted to unsigned. Negative numbers are stored in 2's complement form and unsigned value of the 2's complement form is much higher than the sizeof int.
Question 9
Suppose n and p are unsigned int variables in a C program. We wish to set p to nC3. If n is large, which of the following statements is most likely to set p correctly?
Ap = n * (n-1) * (n-2) / 6;
Bp = n * (n-1) / 2 * (n-2) / 3;
Cp = n * (n-1) / 3 * (n-2) / 2;
Dp = n * (n-1) * (n-2) / 6.0;
Data Types **GATE-CS-2014-(Set-2)**
Discuss it
Question 9 Explanation:
As n is large, the product n*(n-1)*(n-2) will go out of the range(overflow) and it will return a value different from what is expected. So we consider a shorter product n*(n-1). n*(n-1) is always an even number. So the subexpression "n * (n-1) / 2 " in option B would always produce an integer, which means no precision loss in this subexpression. And when we consider `n*(n-1)/2*(n-2)`, it will always give a number which is a multiple of 3. So dividing it with 3 won't have any loss.
Question 10
Output of following program?

```
#include<stdio.h>
int main()
{
    float x = 0.1;
    if ( x == 0.1 )
        printf("IF");
    else if (x == 0.1f)
        printf("ELSE IF");
    else
        printf("ELSE");
}
```

AELSE IF
BIF
CELSE
Data Types
Discuss it
Question 11
Suppose a C program has floating constant 1.414, what's the best way to convert this as "float" data type?
A(float)1.414
Bfloat(1.414)
C1.414f or 1.414F
D1.414 itself of "float" data type i.e. nothing else required.
Data Types **C Quiz - 102**
Discuss it
Question 11 Explanation:
By default floating constant is of double data type. By suffixing it with f or F, it can be converted to float data type. For more details, this post can be referred here.
Question 12
In a C program, following variables are defined:

```
float     x = 2.17;
double   y = 2.17;
long double z = 2.17;
```

Which of the following is correct way for printing these variables via printf.
Aprintf("%f %lf %Lf",x,y,z);
Bprintf(“%f %f %f”,x,y,z);
Cprintf("%f %ff %fff",x,y,z);
Dprintf("%f %lf %llf",x,y,z);

Data Types **C Quiz - 106**
Discuss it
Question 12 Explanation:
In C language, float, double and long double are called real data types. For “float”, “double” and “long double”, the right format specifiers are %f, %lf and %Lf from the above options. It should be noted that C standard has specified other format specifiers as well for real types which are %g, %e etc.
Question 13
“typedef” in C basically works as an alias. Which of the following is correct for “typedef”?
Atypedef can be used to alias compound data types such as struct and union.
Btypedef can be used to alias both compound data types and pointer to these compound types.
Ctypedef can be used to alias a function pointer.
Dtypedef can be used to alias an array.
EAll of the above.
Data Types **C Quiz - 106**
Discuss it
Question 13 Explanation:
In C, typedef can be used to alias or make synonyms of other types. Since function pointer is also a type, typedef can be used here. Since, array is also a type of symmetric data types, typedef can be used to alias array as well.
Question 14
In a C program snippet, followings are used for definition of Integer variables?

```
signed s;
unsigned u;
long l;
long long ll;
```

Pick the best statement for these.
AAll of the above variable definitions are incorrect because basic data type int is missing.
BAll of the above variable definitions are correct because int is implicitly assumed in all of these.
COnly “long l;” and “long long ll;” are valid definitions of variables.
DOnly “unsigned u;” is valid definition of variable.
Data Types **C Quiz - 107**
Discuss it
Question 14 Explanation:
Please note that *signed*, *unsigned* and *long* all three are Type specifiers. And *int* is implicitly assumed in all of three. As per C standard, “int, signed, or signed int” are equivalent. Similarly, “unsigned, or unsigned int” are equivalent. Besides, “long, signed long, long int, or signed long int” are all equivalent. And “long long, signed long long, long long int, or signed long long int“ are equivalent.
There are 14 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Question 1
Consider the following two C lines

```
int var1;
extern int var2;
```

Which of the following statements is correct
ABoth statements only declare variables, don't define them.
BFirst statement declares and defines var1, but second statement only declares var2
CBoth statements declare define variables var1 and var2
Variable Declaration and Scope
Discuss it
Question 1 Explanation:
See Understanding “extern” keyword in C
Question 2
Predict the output

```
#include <stdio.h>
int var = 20;
int main()
{
    int var = var;
    printf("%d ", var);
    return 0;
}
```

AGarbage Value
B20
CCompiler Error
Variable Declaration and Scope
Discuss it
Question 2 Explanation:
First var is declared, then value is assigned to it. As soon as var is declared as a local variable, it hides the global variable var.
Question 3

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

ACompiler Error: var is not defined
B20
C0
Variable Declaration and Scope
Discuss it
Question 3 Explanation:
var is only declared and not defined (no memory allocated for it) Refer: Understanding “extern” keyword in C
Question 4

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

A10
BCompiler Error: var is not defined
C0
Variable Declaration and Scope
Discuss it
Question 4 Explanation:
If a variable is only declared and an initializer is also provided with that declaration, then the memory for that variable will be allocated i.e. that variable will be considered as defined. Refer: Understanding “extern” keyword in C
Question 5
Output?

```
int main()
{
  {
      int var = 10;
  }
  {
      printf("%d", var);
  }
  return 0;
}
```

A10
BCompiler Errror
CGarbage Value
Variable Declaration and Scope
Discuss it
Question 5 Explanation:
x is not accessible. The curly brackets define a block of scope. Anything declared between curly brackets goes out of scope after the closing bracket.
Question 6

Output?

```
#include <stdio.h>
int main()
{
  int x = 1, y = 2, z = 3;
  printf(" x = %d, y = %d, z = %d \n", x, y, z);
  {
       int x = 10;
       float y = 20;
       printf(" x = %d, y = %f, z = %d \n", x, y, z);
       {
             int z = 100;
             printf(" x = %d, y = %f, z = %d \n", x, y, z);
       }
  }
  return 0;
}
```

A
```
x = 1, y = 2, z = 3
x = 10, y = 20.000000, z = 3
x = 1, y = 2, z = 100
```

BCompiler Error

C
```
x = 1, y = 2, z = 3
x = 10, y = 20.000000, z = 3
x = 10, y = 20.000000, z = 100
```

D
```
x = 1, y = 2, z = 3
x = 1, y = 2, z = 3
x = 1, y = 2, z = 3
```

Variable Declaration and Scope
Discuss it
Question 6 Explanation:
See Scope rules in C
Question 7

```
int main()
{
  int x = 032;
  printf("%d", x);
  return 0;
}
```

A32
B0
C26
D50
Variable Declaration and Scope
Discuss it
Question 7 Explanation:
When a constant value starts with 0, it is considered as octal number. Therefore the value of x is 3*8 + 2 = 26
Question 8
Consider the following C program, which variable has the longest scope?

```
int a;
int main()
{
   int b;
   // ..
   // ..
}
int c;
```

Aa
Bb
Cc
DAll have same scope

Variable Declaration and Scope
Discuss it
Question 8 Explanation:
a is accessible everywhere. b is limited to main() c is accessible only after its declaration.
Question 9
Consider the following variable declarations and definitions in C

```
i) int var_9 = 1;
ii) int 9_var = 2;
iii) int _ = 3;
```

Choose the correct statement w.r.t. above variables.
ABoth i) and iii) are valid.
BOnly i) is valid.
CBoth i) and ii) are valid.
DAll are valid.
Variable Declaration and Scope **C Quiz - 101**
Discuss it
Question 9 Explanation:
In C language, a variable name can consists of letters, digits and underscore i.e. _ . But a variable name has to start with either letter or underscore. It can't start with a digit. So valid variables are var_9 and _ from the above question. Even two back to back underscore i.e. __ is also a valid variable name. Even _9 is a valid variable. But 9var and 9_ are invalid variables in C. This will be caught at the time of compilation itself. That's why the correct answer is A).
Question 10
Find out the correct statement for the following program.

```
#include "stdio.h"

int * gPtr;

int main()
{
 int * lPtr = NULL;

 if(gPtr == lPtr)
 {
   printf("Equal!");
 }
 else
 {
  printf("Not Equal");
 }

 return 0;
}
```

AIt’ll always print Equal.
BIt’ll always print Not Equal.
CSince gPtr isn’t initialized in the program, it’ll print sometimes Equal and at other times Not Equal.
Variable Declaration and Scope **C Quiz - 109**
Discuss it
Question 10 Explanation:
It should be noted that global variables such gPtr (which is a global pointer to int) are initialized to ZERO. That’s why gPtr (which is a global pointer and initialized implicitly) and lPtr (which a is local pointer and initialized explicitly) would have same value i.e. correct answer is a.
Question 11
Consider the program below in a hypothetical language which allows global variable and a choice of call by reference or call by value methods of parameter passing.

```
 int i ;
program main ()
{
    int j = 60;
    i = 50;
    call f (i, j);
    print i, j;
}
procedure f (x, y)
{
    i = 100;
    x = 10;
    y = y + i ;
}
```

Which one of the following options represents the correct output of the program for the two parameter passing mechanisms?
ACall by value : i = 70, j = 10; Call by reference : i = 60, j = 70
BCall by value : i = 50, j = 60; Call by reference : i = 50, j = 70
CCall by value : i = 10, j = 70; Call by reference : i = 100, j = 60
DCall by value : i = 100, j = 60; Call by reference : i = 10, j = 70
Variable Declaration and Scope C Quiz - 113 Gate IT 2007
Discuss it
Question 11 Explanation:
Call by value: A copy of parameters will be passed and whatever updations are performed will be valid only for that copy, leaving original values intact.
Call by reference: A link to original variables will be passed, by allowing the function to manipulate the original variables.
There are 11 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Question 1
Which of the following is not a storage class specifier in C?
Aauto
Bregister
Cstatic
Dextern
Evolatile
Ftypedef
Storage Classes and Type Qualifiers
Discuss it
Question 1 Explanation:
volatile is not a storage class specifier. volatile and const are type qualifiers.
Question 2
Output of following program?

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

A4 3 2 1
B1 2 3 4
C0 0 0 0
DCompiler Error
Storage Classes and Type Qualifiers
Discuss it
Question 2 Explanation:
A static variable is shared among all calls of a function. All calls to main() in the given program share the same i. i becomes 0 before the printf() statement in all calls to main().
Question 3

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

A4 3 2 1
B1 2 3 4
C4 4 4 4
D0 0 0 0
Storage Classes and Type Qualifiers
Discuss it
Question 3 Explanation:
Since i is static variable, it is shared among all calls to main(). So is reduced by 1 by every function call.
Question 4

```
#include <stdio.h>
int main()
{
    int x = 5;
    int * const ptr = &x;
    ++(*ptr);
    printf("%d", x);

    return 0;
}
```

ACompiler Error
BRuntime Error
C6
D5
Storage Classes and Type Qualifiers
Discuss it
Question 4 Explanation:
See following declarations to know the difference between constant pointer and a pointer to a constant. **int * const ptr** —> ptr is constant pointer. You can change the value at the location pointed by pointer p, but you can not change p to point to other location. **int const * ptr** —> ptr is a pointer to a constant. You can change ptr to point other variable. But you cannot change the value pointed by ptr. Therefore above program works well because we have a constant pointer and we are not changing ptr to point to any other location. We are only icrementing value pointed by ptr.
Question 5

```
#include <stdio.h>
int main()
{
    int x = 5;
    int const * ptr = &x;
    ++(*ptr);
    printf("%d", x);

    return 0;
}
```

ACompiler Error
BRuntime Error
C6
D5
Storage Classes and Type Qualifiers
Discuss it
Question 5 Explanation:
See following declarations to know the difference between constant pointer and a pointer to a constant. **int * const ptr** —> ptr is constant pointer. You can change the value at the location pointed by pointer p, but you can not change p to point to other location. **int const * ptr** —> ptr is a pointer to a constant. You can change ptr to point other variable. But you cannot change the value pointed by ptr. In the above program, ptr is a pointer to a constant. So the value pointed cannot be changed.
Question 6

```
#include<stdio.h>
int main()
{
  typedef static int *i;
  int j;
  i a = &j;
  printf("%d", *a);
  return 0;
}
```

ARuntime Error
B0

CGarbage Value
DCompiler Error
Storage Classes and Type Qualifiers
Discuss it
Question 6 Explanation:
Compiler Error -> Multiple Storage classes for a. In C, typedef is considered as a storage class. The Error message may be different on different compilers.
Question 7
Output?

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

ACompiler Error
BRuntime Error
C0
D1
Storage Classes and Type Qualifiers
Discuss it
Question 7 Explanation:
There is no problem with the program. It simply creates a user defined type i and creates a variable a of type i.
Question 8

```
#include<stdio.h>
int main()
{
  typedef int *i;
  int j = 10;
  i *a = &j;
  printf("%d", **a);
  return 0;
}
```

ACompiler Error
BGarbage Value
C10
D0
Storage Classes and Type Qualifiers
Discuss it
Question 8 Explanation:
Compiler Error -> Initialization with incompatible pointer type. The line typedef int *i makes i as type int *. So, the declaration of a means a is pointer to a pointer. The Error message may be different on different compilers.
Question 9
Output?

```
#include <stdio.h>
int fun()
{
  static int num = 16;
  return num--;
}

int main()
{
  for(fun(); fun(); fun())
    printf("%d ", fun());
  return 0;
}
```

AInfinite loop
B13 10 7 4 1
C14 11 8 5 2
D15 12 8 5 2
Storage Classes and Type Qualifiers
Discuss it

Question 9 Explanation:
Since *num* is static in *fun()*, the old value of *num* is preserved for subsequent functions calls. Also, since the statement return *num*– is postfix, it returns the old value of *num*, and updates the value for next function call.

fun() called first time: num = 16 // for loop initialization done;

In test condition, compiler checks for non zero value

fun() called again : num = 15

printf("%d \n", fun());:**num=14 ->printed**

Increment/decrement condition check

fun(); called again : num = 13

----

fun() called second time: num: 13

In test condition,compiler checks for non zero value

fun() called again : num = 12

printf("%d \n", fun());:**num=11 ->printed**

fun(); called again : num = 10

----

fun() called second time : num = 10

In test condition,compiler checks for non zero value

fun() called again : num = 9

printf("%d \n", fun());:**num=8 ->printed**

fun(); called again   : num = 7

----

fun() called second time: num = 7

In test condition,compiler checks for non zero value

fun() called again : num = 6

printf("%d \n", fun());:**num=5 ->printed**

fun(); called again   : num = 4

----

fun() called second time: num: 4

In test condition,compiler checks for non zero value

fun() called again : num = 3

printf("%d \n", fun());:**num=2 ->printed**

fun(); called again   : num = 1

----

fun() called second time: num: 1

In test condition,compiler checks for non zero value

fun() called again : num = 0 => **STOP**

Question 10

```
#include <stdio.h>
int main()
{
  int x = 10;
  static int y = x;

  if(x == y)
     printf("Equal");
  else if(x > y)
     printf("Greater");
  else
     printf("Less");
  return 0;
}
```

ACompiler Error
BEqual
CGreater
DLess
Storage Classes and Type Qualifiers
Discuss it
Question 10 Explanation:
In C, static variables can only be initialized using constant literals. This is allowed in C++ though. See this GFact for details.
Question 11
Consider the following C function

```
int f(int n)
{
   static int i = 1;
   if (n >= 5)
      return n;
   n = n+i;
   i++;
   return f(n);
}
```

The value returned by f(1) is (GATE CS 2004)
A5
B6
C7
D8
Storage Classes and Type Qualifiers
Discuss it
Question 11 Explanation:
Since i is static, first line of f() is executed only once.

```
Execution of f(1)
    i = 1
    n = 2
    i = 2
 Call f(2)
    i = 2
    n = 4
    i = 3
 Call f(4)
   i = 3
   n = 7
   i = 4
 Call f(7)
  since n >= 5 return n(7)
```

Question 12
In C, static storage class cannot be used with:
AGlobal variable
BFunction parameter
CFunction name
DLocal variable

Storage Classes and Type Qualifiers
Discuss it

Question 12 Explanation:
Declaring a global variable as static limits its scope to the same file in which it is defined. A static function is only accessible to the same file in which it is defined. A local variable declared as static preserves the value of the variable between the function calls.
Question 13
Output? (GATE CS 2012)

```
#include <stdio.h>
int a, b, c = 0;
void prtFun (void);
int main ()
{
   static int a = 1; /* line 1 */
   prtFun();
   a += 1;
   prtFun();
   printf ( "\n %d %d " , a, b) ;
}

void prtFun (void)
{
   static int a = 2; /* line 2 */
   int b = 1;
   a += ++b;
   printf (" \n %d %d " , a, b);
}
```

A3 1
  4 1
  4 2
B4 2
  6 1
  6 1
C4 2
  6 2
  2 0
D3 1
  5 2
  5 2
Storage Classes and Type Qualifiers
Discuss it
Question 13 Explanation:
'a' and 'b' are global variable. prtFun() also has 'a' and 'b' as local variables. The local variables hide the globals (See Scope rules in C). When prtFun() is called first time, the local 'b' becomes 2 and local 'a' becomes 4. When prtFun() is called second time, same instance of local static 'a' is used and a new instance of 'b' is created because 'a' is static and 'b' is non-static. So 'b' becomes 2 again and 'a' becomes 6. main() also has its own local static variable named 'a' that hides the global 'a' in main. The printf() statement in main() accesses the local 'a' and prints its value. The same printf() statement accesses the global 'b' as there is no local variable named 'b' in main. Also, the defaut value of static and global int variables is 0. That is why the printf statement in main() prints 0 as value of b.
Question 14
What output will be generated by the given code d\segment if: Line 1 is replaced by "auto int a = 1;" Line 2 is replaced by "register int a = 2;" (GATE CS 2012)
A3 1
  4 1
  4 2
B4 2
  6 1
  6 1
C4 2
  6 2
  2 0
D4 2
  4 2
  2 0
Storage Classes and Type Qualifiers
Discuss it
Question 14 Explanation:
If we replace line 1 by "auto int a = 1;" and line 2 by "register int a = 2;", then 'a' becomes non-static in prtFun(). The output of first prtFun() remains same. But, the output of second prtFun() call is changed as a new instance of 'a' is created in second call. So "4 2″ is printed again. Finally, the printf() in main will print "2 0″. Making 'a' a register variable won't change anything in output. Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above.
Question 15

Output?

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

APrints 10 on all compilers
BMay generate compiler Error
CPrints 0 on all compilers
DMay generate runtime Error
Storage Classes and Type Qualifiers
Discuss it
Question 15 Explanation:
See point 1 of Register Keyword
Question 16

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

A0 10
BCompiler Error
C0 0
D10 10
Storage Classes and Type Qualifiers
Discuss it
Question 16 Explanation:
See extern keyword
Question 17
Output?

```
#include <stdio.h>

int main(void)
{
    int i = 10;
    const int *ptr = &i;
    *ptr = 100;
    printf("i = %d\n", i);
    return 0;
}
```

Ai = 100
Bi = 10
CCompiler Error
DRuntime Error
Storage Classes and Type Qualifiers
Discuss it
Question 17 Explanation:
Note that ptr is a pointer to a constant. So value pointed cannot be changed using the pointer ptr. See Const Qualifier in C for more details.
Question 18
Output of following program

```
#include <stdio.h>
int fun(int n)
{
    static int s = 0;
    s = s + n;
    return (s);
}
```

```
int main()
{
    int i = 10, x;
    while (i > 0)
    {
        x = fun(i);
        i--;
    }
    printf ("%d ", x);
    return 0;
}
```

A0
B100
C110
D55
Storage Classes and Type Qualifiers
Discuss it
Question 18 Explanation:
Since s is static, different values of i are added to it one by one. So final value of s is s = i + (i-1) + (i-2) + ... 3 + 2 + 1. The value of s is i*(i+1)/2. For i = 10, s is 55.
Question 19

```
#include <stdio.h>
char *fun()
{
    static char arr[1024];
    return arr;
}

int main()
{
    char *str = "geeksforgeeks";
    strcpy(fun(), str);
    str = fun();
    strcpy(str, "geeksquiz");
    printf("%s", fun());
    return 0;
}
```

Ageeksforgeeks
Bgeeksquiz
Cgeeksforgeeks geeksquiz
DCompiler Error
Storage Classes and Type Qualifiers
Discuss it
Question 19 Explanation:
Note that arr[] is static in fun() so no problems of returning address, arr[] will stay there even after the fun() returns and all calls to fun() share the same arr[].

```
    strcpy(fun(), str);  // Copies "geeksforgeeks" to arr[]
    str = fun();    // Assigns address of arr to str
    strcpy(str, "geeksquiz");  // copies geeksquiz to str which is address of arr[]
    printf("%s", fun());   // prints "geeksquiz"
```

Question 20
Consider the following C function, what is the output?

```
#include <stdio.h>
int f(int n)
{
    static int r = 0;
    if (n <= 0) return 1;
    if (n > 3)
    {
        r = n;
        return f(n-2)+2;
    }
    return f(n-1)+r;
}
```

```
int main()
{
   printf("%d", f(5));
}
```

A5
B7
C9
D18
Storage Classes and Type Qualifiers   GATE-CS-2007
Discuss it
Question 20 Explanation:

```
f(5) = f(3)+2
The line "r = n" changes value of r to 5.  Since r
is static, its value is shared be all subsequence
calls.  Also, all subsequent calls don't change r
because the statement "r = n" is in a if condition
with n > 3.

f(3) = f(2)+5
f(2) = f(1)+5
f(1) = f(0)+5
f(0) = 1

So f(5) = 1+5+5+5+2 = 18
```

Question 21
In the context of C data types, which of the followings is correct?
A“unsigned long long int” is a valid data type.
B“long long double” is a valid data type.
C“unsigned long double” is a valid data type.
DA), B) and C) all are valid data types.
EA), B) and C) all are invalid data types.
Storage Classes and Type Qualifiers   C Quiz - 102
Discuss it
Question 21 Explanation:
In C, "float" is single precision floating type. "double" is double precision floating type. "long double"is often more precise than double precision floating type. So the maximum floating type is "long double". There's nothing called "long long double". If someone wants to use bigger range than "long double", we need to define our own data type i.e. user defined data type. Besides, Type Specifiers "signed" and "unsigned" aren't applicable for floating types (float, double, long double). Basically, floating types are always signed only.

But integer types i.e. "int", "long int" and "long long int" are valid combinations. As per C standard, "long long int" would be at least 64 bits i.e. 8 bytes. By default integer types would be signed. If we need to make these integer types as unsigned, one can use Type Specifier "unsigned". That's why A) is correct answer.

Question 22
For the following “typedef” in C, pick the best statement

```
typedef int INT, *INTPTR, ONEDARR[10], TWODARR[10][10];
```

AIt will cause compile error because typedef is being used to define multiple aliases of incompatible types in the same statement.
B“INT x” would define x of type int. Remaining part of the statement would be ignored.
C“INT x” would define x of type int and “INTPTR y” would define pointer y of type int *. Remaining part of the statement would be ignored.
D“INT x” would define x of type int. “INTPTR y” would define pointer y of type int *. ONEDARR is an array of 10 int. TWODARR is a 2D array of 10 by 10 int.
E“INT x” would define x of type int. “INTPTR *y” would define pointer y of type int **. “ONEDARR z” would define z as array of 10 int. “TWODARR t” would define t as array of 10 by 10 int.
Storage Classes and Type Qualifiers   C Quiz - 106
Discuss it
Question 22 Explanation:
Here, INT is alias of int. INTPTR is alias of int *. That’s why INTPTR * would be alias of int **. Similarly, ONEDARR is defining the alias not array itself. ONEDARR would be alias to int [10]. That’s why “ONEDARR z” would define array z of int [10]. Similarly, TWODARR would be alias to int [10][10]. Hence “TWODARR t” would define array t of int [10][10]. We can see that typedef can be used to create alias or synonym of other types.
Question 23
Which of the followings is correct for a function definition along with storage-class specifier in C language?
Aint fun(auto int arg)
Bint fun(static int arg)
Cint fun(register int arg)

Dint fun(extern int arg)
EAll of the above are correct.
Storage Classes and Type Qualifiers   C Quiz - 107
Discuss it
Question 23 Explanation:
As per C standard, “*The only storage-class specifier that shall occur in a parameter declaration is register.*” That’s why correct answer is C.
Question 24
Pick the correct statement for const and volatile.
Aconst is the opposite of volatile and vice versa.
Bconst and volatile can’t be used for struct and union.
Cconst and volatile can’t be used for enum.
Dconst and volatile can’t be used for typedef.
Econst and volatile are independent i.e. it’s possible that a variable is defined as both const and volatile.
Storage Classes and Type Qualifiers   C Quiz - 107
Discuss it
Question 24 Explanation:
In C, const and volatile are type qualifiers and these two are independent. Basically, const means that the value isn’t modifiable by the program. And volatile means that the value is subject to sudden change (possibly from outside the program). In fact, C standard mentions an example of valid declaration which is both const and volatile. The example is “extern const volatile int real_time_clock;” where real_time_clock may be modifiable by hardware, but cannot be assigned to, incremented, or decremented. So we should already treat const and volatile separately. Besides, these type qualifier applies for struct, union, enum and typedef as well.
Question 25
Pick the best statement for the following program snippet:

```
#include "stdio.h"
void foo(void)
{
 static int staticVar;
 staticVar++;
 printf("foo: %d\n",staticVar);
}

void bar(void)
{
 static int staticVar;
 staticVar++;
 printf("bar: %d\n",staticVar);
}

int main()
{
 foo(), bar(), foo();
 return 0;
}
```

ACompile error because same static variable name is used in both foo and bar. Since these static variables retain their values even after function is over, same name can’t be used in both the functions.
BCompile error because semicolon isn’t used while calling foo() and bar() in side main function.
CNo compile error and only one copy of staticVar would be used across both the functions and that’s why final value of that single staticVar would be 3.
DNo compile error and separate copies of staticVar would be used in both the functions. That’s why staticVar in foo() would be 2 while staticVar in bar() would be 1.
Storage Classes and Type Qualifiers   C Quiz - 111
Discuss it
Question 25 Explanation:
Here, even though life of static variables span across function calls but their scope is respective to their function body only. That’s why staticVar of each function has separate copies whose life span across function calls. And d is correct.
There are 25 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Question 1

```
#include "stdio.h"
int main()
```

```
{
  int x, y = 5, z = 5;
  x = y == z;
  printf("%d", x);

  getchar();
  return 0;
}
```

A0
B1
C5
DCompiler Error
Operators
Discuss it
Question 1 Explanation:
The crux of the question lies in the statement x = y==z. The operator == is executed before = because precedence of comparison operators (= and ==) is higher than assignment operator =. The result of a comparison operator is either 0 or 1 based on the comparison result. Since y is equal to z, value of the expression y == z becomes 1 and the value is assigned to x via the assignment operator.
Question 2

```
#include <stdio.h>

int main()
{
    int i = 1, 2, 3;

    printf("%d", i);

    return 0;
}
```

A1
B3
CGarbage value
DCompile time error
Operators
Discuss it
Question 2 Explanation:
Comma acts as a separator here. The compiler creates an integer variable and initializes it with 1. The compiler fails to create integer variable 2 because 2 is not a valid identifier.
Question 3

```
#include <stdio.h>

int main()
{
    int i = (1, 2, 3);

    printf("%d", i);

    return 0;
}
```

A1
B3
CGarbage value
DCompile time error
Operators
Discuss it
Question 3 Explanation:
The bracket operator has higher precedence than assignment operator. The expression within bracket operator is evaluated from left to right but it is always the result of the last expression which gets assigned.
Question 4

```
#include <stdio.h>

int main()
{
    int i;

    i = 1, 2, 3;
    printf("%d", i);

    return 0;
}
```

A1
B3
CGarbage value
DCompile time error
Operators
Discuss it
Question 4 Explanation:
Comma acts as an operator. The assignment operator has higher precedence than comma operator. So, the expression is considered as (i = 1), 2, 3 and 1 gets assigned to variable i.
Question 5

```
#include <stdio.h>
int main()
{
    int i = 3;
    printf("%d", (++i)++);
    return 0;
}
```

What is the output of the above program?
A3
B4
C5
DCompile-time error
Operators
Discuss it
Question 5 Explanation:
In C, prefix and postfix operators need l-value to perform operation and return r-value. The expression **(++i)++** when executed increments the value of variable i(i is a l-value) and returns r-value. The compiler generates the error(l-value required) when it tries to post-incremeny the value of a r-value.
Question 6
What is the output of below program?

```
#include <stdio.h>
int foo(int* a, int* b)
{
    int sum = *a + *b;
    *b = *a;
    return *a = sum - *b;
}
int main()
{
    int i = 0, j = 1, k = 2, l;
    l = i++ || foo(&j, &k);
    printf("%d %d %d %d", i, j, k, l);
    return 0;
}
```

A1 2 1 1
B1 1 2 1
C1 2 2 1
D1 2 2 2
Operators
Discuss it
Question 6 Explanation:
The control in the logical OR goes to the second expression only if the first expression results in FALSE. The function foo() is called because **i++** returns 0(post-increment) after incrementing the value of i to 1. The foo() function actually swaps the values of two variables and returns the value of second parameter. So, values of variables j and k gets exchanged and OR expression evaluates to be TRUE.
Question 7

```
#include <stdio.h>
int main()
{
    int i = 5, j = 10, k = 15;
    printf("%d ", sizeof(k /= i + j));
    printf("%d", k);
    return 0;
}
```

Assume size of an integer as 4 bytes. What is the output of above program?
A4 1
B4 15
C2 1
DCompile-time error
Operators
Discuss it
Question 7 Explanation:
The main theme of the program lies here: **sizeof(k /= i + j)**. An expression doesn't get evaluated inside **sizeof** operator. **sizeof** operator returns sizeof(int) because the result of expression will be an integer. As the expression is not evaluated, value of **k** will not be changed.
Question 8

```
#include <stdio.h>
int main()
{
    //Assume sizeof character is 1 byte and sizeof integer is 4 bytes
    printf("%d", sizeof(printf("GeeksQuiz")));
    return 0;
}
```

AGeeksQuiz4
B4GeeksQuiz
CGeeksQuiz9
D4
ECompile-time error
Operators
Discuss it
Question 8 Explanation:
An expression doesn't get evaluated inside **sizeof** operator. **GeeksQuiz** will not be printed. **printf** returns the number of characters to be printed i.e. 9 which is an integer value. **sizeof** operator returns sizeof(int).
Question 9
Output of following program?

```
#include <stdio.h>
int f1() { printf ("Geeks"); return 1;}
int f2() { printf ("Quiz"); return 1;}

int main()
{
  int p = f1() + f2();
  return 0;
}
```

AGeeksQuiz
BQuizGeeks
CCompiler Dependent
DCompiler Error
Operators
Discuss it
Question 9 Explanation:
The operator '+' doesn't have a standard defined order of evaluation for its operands. Either f1() or f2() may be executed first. So a compiler may choose to output either "GeeksQuiz" or "QuizGeeks".
Question 10
What is the output of following program?

```
#include <stdio.h>

int main()
{
   int a = 1;
   int b = 1;
```

```
    int c = a || --b;
    int d = a-- && --b;
    printf("a = %d, b = %d, c = %d, d = %d", a, b, c, d);
    return 0;
  }
```

Aa = 0, b = 1, c = 1, d = 0
Ba = 0, b = 0, c = 1, d = 0
Ca = 1, b = 1, c = 1, d = 1
Da = 0, b = 0, c = 0, d = 0
Operators
Discuss it
Question 10 Explanation:
Let us understand the execution line by line. Initial values of a and b are 1.

```
    // Since a is 1, the expression --b is not executed because
    // of the short-circuit property of logical or operator
    // So c becomes 1, a and b remain 1
    int c = a || --b;

    // The post decrement operator -- returns the old value in current expression
    // and then updates the value. So the value of expression --a is 1.  Since the
    // first operand of logical and is 1, shortcircuiting doesn't happen here.  So
    // the expression --b is executed and --b returns 0 because it is pre-increment.
    // The values of a and b become 0, and the value of d also becomes 0.
    int d = a-- && --b;
```

Question 11

```
#include <stdio.h>
int main()
{
  int a = 10, b = 20, c = 30;
  if (c > b > a)
    printf("TRUE");
  else
    printf("FALSE");
  return 0;
}
```

ATRUE
BFALSE
CCompiler Error
DOutput is compiler dependent
Operators
Discuss it
Question 11 Explanation:
Let us consider the condition inside the if statement. Since there are two greater than (>) operators in expression “c > b > a”, associativity of > is considered. Associativity of > is left to right. So, expression c > b > a is evaluated as ((c > b) > a) which is false.
Question 12

```
#include<stdio.h>
int main()
{
  char *s[] = { "knowledge","is","power"};
  char **p;
  p = s;
  printf("%s ", ++*p);
  printf("%s ", *p++);
  printf("%s ", ++*p);

  return 0;
}
```

Ais power
Bnowledge nowledge s
Cis ower
Dnowledge knowledge is

Operators

Discuss it

Question 12 Explanation:

Let us consider the expression ++*p in first printf(). Since precedence of prefix ++ and * is same, associativity comes into picture. *p is evaluated first because both prefix ++ and * are right to left associative. When we increment *p by 1, it starts pointing to second character of “knowledge”. Therefore, the first printf statement prints “nowledge”. Let us consider the expression *p++ in second printf() . Since precedence of postfix ++ is higher than *, p++ is evaluated first. And since it’s a psotfix ++, old value of p is used in this expression. Therefore, second printf statement prints “nowledge”. In third printf statement, the new value of p (updated by second printf) is used, and third printf() prints “s”.

Question 13

```
#include<stdio.h>
int main(void)
{
  int a = 1;
  int b = 0;
  b = a++ + a++;
  printf("%d %d",a,b);
  return 0;
}
```

A3 6

BCompiler Dependent

C3 4

D3 3

Operators

Discuss it

Question 13 Explanation:

See http://www.geeksforgeeks.org/sequence-points-in-c-set-1/ for explanation.

Question 14

Which of the following is not a logical operator?

A&&

B!

C||

D|

Operators

Discuss it

Question 14 Explanation:

&&- Logical AND **!**- Logical NOT ||- Logical OR |- Bitwise OR(used in bitwise manipulations)

Question 15

Predict the output of following program. Assume that the characters are represented using ASCII Values.

```
#include <stdio.h>
#define VAL 32

int main()
{
    char arr[] = "geeksquiz";
    *(arr + 0) &= ~VAL;
    *(arr + 5) &= ~VAL;
    printf("%s", arr);

    return 0;
}
```

AGeeksQuiz

BgeeksQuiz

CGeeksquiz

Dgeeksquiz

EGarbage eeks Garbage uiz

Operators

Discuss it

Question 15 Explanation:

The crux of the question lies in the statement: **(arr + 5) &= ~VAL;* This statement subtracts 32 from the ascii value of a lower case character and thus converts it to upper case. This is another way to convert an alphabet to upper case by resetting its bit positioned at value 32 i.e. 5th bit from the LSB(assuming LSB bit at position 0).

Question 16

Predict the output of the below program:

```
#include <stdio.h>
int main()
{
    printf("%d", 1 << 2 + 3 << 4);
    return 0;
}
```

A112
B52
C512
D0
Operators
Discuss it
Question 16 Explanation:
The main logic behind the program is the precedence and associativity of the operators. The addition(+) operator has higher precedence than shift(1 which in turn reduces to (1
Question 17
Which of the following can have different meaning in different contexts?
A&
B*
CBoth of the above
DThere are no such operators in C
Operators
Discuss it
Question 17 Explanation:
'&' can be used to get address of a variable and can also be used to do bitwise and operation. Similarly '*' can be used to get value at an address and can also be used to multiplication.
Question 18
In C, two integers can be swapped using minimum
A0 extra variable
B1 extra variable
C2 extra variable
Operators
Discuss it
Question 18 Explanation:
We can swap two variables without any extra variable using bitwise XOR operator '^'. Let X and Y be two variables to be swapped. Following steps swap X and Y.

```
 X = X ^ Y;
 Y = X ^ Y;
 X = X ^ Y;
```

See http://en.wikipedia.org/wiki/XOR_swap_algorithm
Question 19
What does the following statement do?

```
x  = x | 1 << n;
```

ASets x as 2^n
BSets (n+1)th bit of x
CToggles (n+1)th bit of x
DUnsets (n+1)th bit of x
Operators
Discuss it
Question 19 Explanation:
Let n be 3, the value of expression 1
Question 20
Predict the output of following C program

```
#include <stdio.h>
int main()
{
    int i = 0;
    do
    {
        printf("GeeqsQuiz ");
        i = i++;
    }
    while (i < 5);
    return 0;
}
```

AGeeqsQuiz GeeqsQuiz GeeqsQuiz GeeqsQuiz GeeqsQuiz

BInfinite time GeeksQuiz
CUndefined Behavior
Operators
Discuss it
Question 20 Explanation:
The below statement causes undefined behavior.

```
 i = i++;
```

See http://www.geeksforgeeks.org/sequence-points-in-c-set-1/ for details.
Question 21
Assume that the size of an integer is 4 bytes, predict the output of following program.

```
#include <stdio.h>
int main()
{
    int i = 12;
    int j = sizeof(i++);
    printf("%d  %d", i, j);
    return 0;
}
```

A12 4
B13 4
CCompiler Error
D0 4
Operators
Discuss it
Question 21 Explanation:
The expressions written inside sizeof are not evaluated, so i++ is not performed.
Question 22

```
#include<stdio.h>
int main()
{
  int a = 2,b = 5;
  a = a^b;
  b = b^a;
  printf("%d %d",a,b);
  return 0;
}
```

A5 2
B2 5
C7 7
D7 2
Operators
Discuss it
Question 22 Explanation:
^ is bitwise xor operator. a = 2 (10) b = 5 (101) a = a^b (10 ^ 101) = 7(111) b = a^b (111 ^ 101) = 2(10)
Question 23
Predict the output of following program?

```
# include <stdio.h>
int main()
{
    int x = 10;
    int y = 20;
    x += y += 10;
    printf (" %d %d", x, y);
    return 0;
}
```

A40 20
B40 30
C30 30
D30 40
Operators
Discuss it
Question 23 Explanation:
The main statement in question is "x += y += 10". Since there are two += operators in the statement, associativity comes into the picture.

Associativity of compound assignment operators is right to left, so the expression is evaluated as x += (y += 10).
Question 24

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

A13 12
B13 13
C10 10
DCompiler Dependent
Operators
Discuss it
Question 24 Explanation:
The comma operator defines a sequence point, so the option (d) is not correct. All expressions are executed from left to right and the value of rightmost expression is returned by the comma operator.
Question 25

```
#include <stdio.h>
int main()
{
  int y = 0;
  int x = (~y == 1);
  printf("%d", x);
  return 0;
}
```

A0
B1
CA bog negative Number
DCompiler Error
Operators
Discuss it
Question 25 Explanation:
The important thing to note here is ~ is a bitwise not operator. So the value of ~0 would be all 1s in binary representation which means decimal value of ~0 is not 1. Therefore the result of comparison operator becomes 0.
Question 26

```
#include <stdio.h>
int main()
{
  int a = 0;
  int b;
  a = (a == (a == 1));
  printf("%d", a);
  return 0;
}
```

A0
B1
CBig negative number
D-1
Operators
Discuss it
Question 26 Explanation:
We need to figure out value of "(a == (a == 1))"
(a == 1) returns false as a is initialized as 0. So in outer bracket, false is compared with a. Since a is 0, result of of outer bracket becomes true.
The important thing to note is, in C, when a boolean value is compared or assigned to an integer value, false is considered as 0 and true is considered as 1.
Question 27

```
#include <stdio.h>
#include <stdlib.h>
int top=0;
int fun1()
```

```
{
   char a[]= {'a','b','c','(','d'};
   return a[top++];
}
int main()
{
   char b[10];
   char ch2;
   int i = 0;
   while (ch2 = fun1() != '(')
   {
      b[i++] = ch2;
   }
   printf("%s",b);
   return 0;
}
```

Aabc(
Babc
C3 special characters with ASCII value 1
DEmpty String
Operators
Discuss it
Question 27 Explanation:
Precedence of '!=' is higher than '='. So the expression "ch2 = fun1() != '('" is treated as "ch2 = (fun1() != '(')". So result of "fun1() != '(' " is assigned to ch2. The result is 1 fir first three characters. Smile character has ASCII value 1. Since the condition is true for first three characters, you get three smilies. If we, put a bracket in while statement, we get "abc". 1 This modified program prints "abc"
Question 28
In the context of modulo operation (i.e. remainder on division) for floating point (say 2.1 and 1.1), pick the best statement.
AFor floating point, modulo operation isn't defined and that's why modulo can't be found.
B(2.1 % 1.1) is the result of modulo operation.
Cfmod(2.1,1.1) is the result of module operation.
D((int)2.1) % ((int)1.1) is the result of modulo operation.
Operators **C Quiz - 102**
Discuss it
Question 28 Explanation:
% works on integer types only not for floating types. Typecasting to integer type might approximate the intended result but it won't produce the correct result. Basically the *fmod(x,y)* function returns the value x – ny, for some integer n such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y. *fmod()* is declared in "math.h" and its prototype is "*double fmod(double x, double y)*". For *float* and *long double* also, modulo has been implemented in math.h library through *fmodf()* and *fmodl()*.
Question 29
For a given integer, which of the following operators can be used to “set” and “reset” a particular bit respectively?
A| and &
B&& and ||
C& and |
D|| and &&
Operators **C Quiz - 103**
Discuss it
Question 29 Explanation:
Bitwise operator | can be used to “set” a particular bit while bitwise operator & can be used to “reset” a particular bit. Please note that && and || are logical operators which evaluate their operands to logical TRUE or FALSE. It should be noted that bitwise operator & can be used for checking a particular bit also i.e. whether a bit is set or not. So correct answer it A.
Question 30
Suppose a, b, c and d are int variables. For ternary operator in C (? :), pick the best statement.
Aa>b ? : ; is valid statement i.e. 2nd and 3rd operands can be empty and they are implicitly replaced with non-zero value at run-time.
Ba>b ? c=10 : d=10; is valid statement. Based on the value of a and b, either c or d gets assigned the value of 10.
Ca>b ? (c=10,d=20) : (c=20,d=10); is valid statement. Based on the value of a and b, either c=10,d=20 gets executed or c=20,d=10 gets executed.
DAll of the above are valid statements for ternary operator.
Operators **C Quiz - 107**
Discuss it
Question 30 Explanation:
For ternary operator, both 2nd and 3rd operands are necessary. So A) isn’t correct. As per operator precedence, ternary operator has higher precedence over assignment operator. So B) isn’t correct.
Question 31
The below program would give compile error because comma has been used after foo(). Instead, semi-colon should be used i.e. the way it has been used after bar(). That's why if we use semi-colon after foo(), the program would compile and run successfully while printing "GeeksQuiz"

```
#include "stdio.h"
```

```
void foo(void)
{
 printf("Geeks");
}
void bar(void)
{
 printf("Quiz");
}

int main()
{
 foo(), bar();
 return 0;
}
```

ATRUE
BFALSE
Operators C Quiz - 110
Discuss it
Question 31 Explanation:
Here, comma is acting as an operator instead of separator. For a comma operator in C, first left operand is evaluated and then right operand is evaluated. That's why foo() would be called followed by bar(). There's no issue with the given program. It'll compile and print "GeeksQuiz" without any modification itself.
There are 31 questions to complete.

GATE CS Corner

Question 1

```
#include <stdio.h>

int main()
{
    int i = 1024;
    for (; i; i >>= 1)
        printf("GeeksQuiz");
    return 0;
}
```

How many times will GeeksQuiz be printed in the above program?

A10
B11
CInfinite
DThe program will show compile-time error

Loops & Control Structure
Discuss it

Question 1 Explanation:
In for loop, mentioning expression is optional. **>>=** is a composite operator. It shifts the binary representation of the value by 1 to the right and assigns the resulting value to the same variable. The for loop is executed until value of variable **i** doesn't drop to 0.

Question 2

```
#include <stdio.h>
#define PRINT(i, limit) do \
                        { \
                            if (i++ < limit) \
                            { \
                                printf("GeeksQuiz\n"); \
                                continue; \
                            } \
                        }while(0)

int main()
{
    int i = 0;
    PRINT(i, 3);
    return 0;
}
```

How many times **GeeksQuiz** is printed in the above program ?

A1
B3
C4
DCompile-time error

Loops & Control Structure
Discuss it

Question 2 Explanation:
If a macro needs to be expanded in multiple lines, it is the best practice to write those lines within **do{ }while(0)** to avoid macro side effects. After **GeeksQuiz** is printed once, the control reaches the while statement to check for the condition. Since, the condition is false, the loop gets terminated.

Question 3
What is the output of the below program?

```
#include <stdio.h>
int main()
{
    int i = 0;
    switch (i)
    {
        case '0': printf("Geeks");
                break;
        case '1': printf("Quiz");
                break;
        default: printf("GeeksQuiz");
    }
    return 0;
}
```

AGeeks
BQuiz

CGeeksQuiz
DCompile-time error
Loops & Control Structure
Discuss it
Question 3 Explanation:
At first look, the output of the program seems to be **Geeks**. But, the cases are labeled with characters which gets converted to their ascii values 48(for 0) and 49(for 1). None of the cases is labeled with value 0. So, the control goes to the default block and **GeeksQuiz** is printed.
Question 4

```
#include <stdio.h>
int main()
{
   int i = 3;
   switch (i)
   {
      case 0+1: printf("Geeks");
             break;
      case 1+2: printf("Quiz");
             break;
      default: printf("GeeksQuiz");
   }
   return 0;
}
```

What is the output of the above program?
AGeeks
BQuiz
CGeeksQuiz
DCompile-time error
Loops & Control Structure
Discuss it
Question 4 Explanation:
Expression gets evaluated in cases. The control goes to the second case block after evaluating **1+2 = 3** and **Quiz** is printed.
Question 5
Predict the output of the below program:

```
#include <stdio.h>
#define EVEN 0
#define ODD 1
int main()
{
   int i = 3;
   switch (i & 1)
   {
      case EVEN: printf("Even");
             break;
      case ODD: printf("Odd");
             break;
      default: printf("Default");
   }
   return 0;
}
```

AEven
BOdd
CDefault
DCompile-time error
Loops & Control Structure
Discuss it
Question 5 Explanation:
The expression **i & 1** returns 1 if the rightmost bit is set and returns 0 if the rightmost bit is not set. As all odd integers have their rightmost bit set, the control goes to the block labeled ODD.
Question 6

```
#include <stdio.h>
int main()
{
   int i;
   if (printf("0"))
      i = 3;
```

```
    else
        i = 5;
    printf("%d", i);
    return 0;
}
```

Predict the output of above program?
A3
B5
C03
D05
Loops & Control Structure
Discuss it
Question 6 Explanation:
The control first goes to the if statement where **0** is printed. The **printf("0")** returns the number of characters being printed i.e. 1. The block under if statement gets executed and i is initialized with 3.
Question 7

```
#include <stdio.h>
int i;
int main()
{
    if (i);
    else
        printf("Ëlse");
    return 0;
}
```

What is correct about the above program?
Aif block is executed.
Belse block is executed.
CIt is unpredictable as i is not initialized.
DError: misplaced else
Loops & Control Structure
Discuss it
Question 7 Explanation:
Since i is defined globally, it is initialized with default value 0. The Else block is executed as the expression within if evaluates to FALSE. Please note that the empty block is equivalent to a semi-colon(;). So the statements **if (i);** and **if (i) {}** are equivalent.
Question 8

```
#include<stdio.h>
int main()
{
   int n;
   for (n = 9; n!=0; n--)
     printf("n = %d", n--);
   return 0;
}
```

What is the output?
A9 7 5 3 1
B9 8 7 6 5 4 3 2 1
CInfinite Loop
D9 7 5 3
Loops & Control Structure
Discuss it
Question 8 Explanation:
The program goes in an infinite loop because n is never zero when loop condition (n != 0) is checked. n changes like 7 5 3 1 -1 -3 -5 -7 -9 ...
Question 9
Output?

```
#include <stdio.h>
int main()
{
    int c = 5, no = 10;
    do {
        no /= c;
    } while(c--);

    printf ("%d\n", no);
    return 0;
}
```

A1
BRuntime Error
C0
DCompiler Error
Loops & Control Structure
Discuss it
Question 9 Explanation:
There is a bug in the above program. It goes inside the do-while loop for c = 0 also. So it fails during runtime.
Question 10

```
# include <stdio.h>
int main()
{
  int i = 0;
  for (i=0; i<20; i++)
  {
    switch(i)
    {
      case 0:
        i += 5;
      case 1:
        i += 2;
      case 5:
        i += 5;
      default:
        i += 4;
        break;
    }
    printf("%d  ", i);
  }
  return 0;
}
```

A5 10 15 20
B7 12 17 22
C16 21
DCompiler Error
Loops & Control Structure
Discuss it
Question 10 Explanation:
Initially i = 0. Since case 0 is true i becomes 5, and since there is no break statement till last statement of switch block, i becomes 16. Now in next iteration no case is true, so execution goes to default and i becomes 21. In C, if one case is true switch block is executed until it finds break statement. If no break statement is present all cases are executed after the true case. If you want to know why switch is implemented like this, well this implementation is useful for situations like below.

```
 switch (c)
 {
   case 'a':
   case 'e':
   case 'i' :
   case 'o':
   case 'u':
     printf(" Vowel character");
     break;
   default :
     printf("Not a Vowel character");; break;
  }
```

Question 11
Output of following C program?

```
#include<stdio.h>
int main()
{
   int i = 0;
   for (printf("1st\n"); i < 2 && printf("2nd\n"); ++i && printf("3rd\n"))
   {
       printf("*\n");
   }
   return 0;
}
```

A1st
  2nd
  *
  3rd
  2nd
  *
B1st
  2nd
  *
  3rd
  2nd
  *
  3rd
C1st
  2nd
  3rd
  *
  2nd
  3rd
D1st
  2nd
  3rd
  *
  1st
  2nd
  3rd

Loops & Control Structure
Discuss it

Question 11 Explanation:
It is just one by one execution of statements in for loop. a) The initial statement is executed only once. b) The second condition is printed before '*' is printed. The second statement also has short circuiting logical && operator which prints the second part only if 'i' is smaller than 2 b) The third statement is printed after '*' is printed. This also has short circuiting logical && operator which prints the second part only if '++i' is not zero.
Question 12

```
#include <stdio.h>
int main()
{
  int i;
  for (i = 1; i != 10; i += 2)
    printf(" GeeksQuiz ");
  return 0;
}
```

AGeeksQuiz GeeksQuiz GeeksQuiz GeeksQuiz GeeksQuiz
BGeeksQuiz GeeksQuiz GeeksQuiz .... infinite times
CGeeksQuiz GeeksQuiz GeeksQuiz GeeksQuiz
DGeeksQuiz GeeksQuiz GeeksQuiz GeeksQuiz GeeksQuiz GeeksQuiz

Loops & Control Structure
Discuss it

Question 12 Explanation:
The loop termination condition never becomes true and the loop prints *GeeksQuiz* infinite times. In general, if a for or while statement uses a loop counter, then it is safer to use a relational operator (such as
Question 13
What will be the output of the following C program segment? (GATE CS 2012)

```
char inchar = 'A';
switch (inchar)
{
case 'A' :
   printf ("choice A \n") ;
case 'B' :
   printf ("choice B ") ;
case 'C' :
case 'D' :
case 'E' :
default:
   printf ("No Choice") ;
}
```

ANo choice
BChoice A
CChoice A
  Choice B No choice
DProgram gives no output as it is erroneous
Loops & Control Structure
Discuss it
Question 13 Explanation:
There is no break statement in case ‘A’. If a case is executed and it doesn’t contain break, then all the subsequent cases are executed until a break statement is found. That is why everything inside the switch is printed. Try following program as an exercise.

```
int main()
{
   char inchar = 'A';
   switch (inchar)
   {
   case 'A' :
      printf ("choice A \n") ;
   case 'B' :
   {
      printf ("choice B") ;
      break;
   }
   case 'C' :
   case 'D' :
   case 'E' :
   default:
      printf ("No Choice") ;
   }
}
```

Question 14
Predict the output of the below program:

```
#include <stdio.h>
int main()
{
   int i = 3;
   switch(i)
   {
      printf("Outside ");
      case 1: printf("Geeks");
         break;
      case 2: printf("Quiz");
         break;
      defau1t: printf("GeeksQuiz");
   }
   return 0;
}
```

AOutside GeeksQuiz
BGeeksQuiz
CNothing gets printed
Loops & Control Structure
Discuss it
Question 14 Explanation:
In a switch block, the control directly flows within the case labels(or dafault label). So, statements which do not fall within these labels, *Outside* is not printed. Please take a closer look at the default label. Its *defau1t*, not *default* which s interpreted by the compiler as a label used for goto statements. Hence, nothing is printed in the above program.
Question 15
In the following program, *X* represents the Data Type of the variable *check*.

```
#include <stdio.h>
int main()
{
   X check;
   switch (check)
   {
      // Some case labels
   }
```

```
    return 0;
}
```

Which of the following cannot represent *X*?
Aint
Bchar
Cenum
Dfloat
Loops & Control Structure
Discuss it
Question 15 Explanation:
A switch expression can be int, char and enum. A float variable/expression cannot be used inside switch.
Question 16
What is the output of the following program?

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

        case 'b' || 2: printf("Quiz ");
                    break;
        default: printf("GeeksQuiz");
    }
    return 0;
}
```

AGeeks
BGeeks Quiz
CGeeks Quiz GeeksQuiz
DCompile-time error
Loops & Control Structure
Discuss it
Question 16 Explanation:
An expression gets evaluated in a case label. Both the cases used are evaluated to 1(true). So *compile-time error: duplicate case value* is flashed as duplicated cases are not allowed.
Question 17
Predict the output of the following program:

```
#include <stdio.h>
int main()
{
    int check = 20, arr[] = {10, 20, 30};
    switch (check)
    {
        case arr[0]: printf("Geeks ");
        case arr[1]: printf("Quiz ");
        case arr[2]: printf("GeeksQuiz");
    }
    return 0;
}
```

AQuiz
BQuiz GeeksQuiz
CGeeksQuiz
DCompile-time error
Loops & Control Structure
Discuss it
Question 17 Explanation:
The case labels must be constant inside switch block. Thats why the *compile-time error: case label does not reduce to an integer constant* is flashed.
Question 18
How many times GeeksQuiz is printed

```
#include<stdio.h>
int main()
{
    int i = -5;
    while (i <= 5)
    {
        if (i >= 0)
            break;
        else
        {
            i++;
            continue;
        }
        printf("GeeksQuiz");
    }
    return 0;
}
```

A10 times
B5 times
CInfinite times
D0 times
Loops & Control Structure
Discuss it
Question 18 Explanation:
The loop keeps incrementing i while it is smaller than 0. When i becomes 0, the loop breaks. So GeeksQuiz is not printed at all.
Question 19

```
#include <stdio.h>
int main()
{
    int i = 3;
    while (i--)
    {
        int i = 100;
        i--;
        printf("%d ", i);
    }
    return 0;
}
```

AInfinite Loop
B99 99 99
C99 98 97
D2 2 2
Loops & Control Structure
Discuss it
Question 19 Explanation:
Note that the i–- in the statement while(i–-) changes the i of main() And i== just after declaration statement int i=100; changes local i of while loop.
Question 20

```
#include <stdio.h>
int main()
{
    int x = 3;
    if (x == 2); x = 0;
    if (x == 3) x++;
    else x += 2;

    printf("x = %d", x);

    return 0;
}
```

Ax = 4
Bx = 2
CCompiler Error
Dx = 0
Loops & Control Structure
Discuss it
Question 20 Explanation:
Value of x would be 2. Note the semicolon after first if statement. x becomes 0 after the first if statement. So control goes to else part of second if else statement.
Question 21

```
#include<stdio.h>
int main()
{
   int a = 5;
   switch(a)
   {
   default:
      a = 4;
   case 6:
      a--;
   case 5:
      a = a+1;
   case 1:
      a = a-1;
   }
   printf("%d \n", a);
   return 0;
}
```

A3
B4
C5
DNone of these
Loops & Control Structure
Discuss it
Question 21 Explanation:
There is no break statement, so first a = a + 1 is executed, then a = a-1 is executed.
Question 22
Which combination of the integer variables x, y and z makes the variable a get the value 4 in the following expression?

```
a = ( x > y ) ? (( x > z ) ? x : z) : (( y > z ) ? y : z )
```

Ax = 3, y = 4, z = 2
Bx = 6, y = 5, z = 3
Cx = 6, y = 3, z = 5
Dx = 5, y = 4, z = 5
Loops & Control Structure **GATE CS 2008**
Discuss it
Question 22 Explanation:
The given expression assigns maximum among three elements (x, y and z) to a.
Question 23
Consider the C program below.

```
#include <stdio.h>
int *A, stkTop;
int stkFunc (int opcode, int val)
{
   static int size=0, stkTop=0;
   switch (opcode)
   {
   case -1:
      size = val;
      break;
   case 0:
      if (stkTop < size ) A[stkTop++]=val;
      break;
   default:
      if (stkTop) return A[--stkTop];
   }
   return -1;
}
int main()
{
   int B[20];
   A=B;
   stkTop = -1;
   stkFunc (-1, 10);
   stkFunc (0, 5);
   stkFunc (0, 10);
   printf ("%d\n", stkFunc(1, 0)+ stkFunc(1, 0));
}
```

The value printed by the above program is ___________
A9
B10
C15
D17
Loops & Control Structure GATE-CS-2015 (Set 2)
Discuss it
Question 23 Explanation:
The code in main, basically initializes a stack of size 10, then pushes 5, then pushes 10. Finally the printf statement prints sum of two pop operations which is 10 + 5 = 15.

```
    stkFunc (-1, 10);   // Initialize size as 10
    stkFunc (0, 5);    // push 5
    stkFunc (0, 10);   // push 10

    // print sum of two pop
    printf ("%d\n", stkFunc(1, 0) + stkFunc(1, 0));
```

Question 24
With respect to following “for” loops in C, pick the best statement Assume that there is a prior declaration of 'i' in all cases

```
for (i < 10; i = 0 ; i++) // (i)
for (i < 10; i++ ; i = 0) // (ii)
for (i = 0; i < 10 ; i++) // (iii)
for (i = 0; i++ ; i < 10) // (iv)
for (i++; i = 0 ; i < 10) // (v)
for (i++; i < 0 ; i = 10) // (vi)
```

AAll the above “for” loops would compile successfully.
BAll the above “for” loops would compile successfully. Except (iii), the behaviour of all the other “for” loops depend on compiler implementation.
COnly (iii) would compile successfully.
DOnly (iii) and (iv) would compile successfully.
EOnly (iii) and (iv) would compile successfully but behaviour of (iv) would depend on compiler implementation.
Loops & Control Structure C Quiz - 104
Discuss it
Question 24 Explanation:
Basically, all of the “for” loops are valid i.e. . In the above examples, it doesn’t matter what expression has been put in which part of a “for” loop. The execution order of these expressions remain same irrespective of where they have been put i.e. "1st expression" followed by "2nd expression" followed by "body of the loop" followed by "3rd expression". But the exact behavior of each of the above "for" loop depends on the body of loop as well. In fact the following is also valid and work without any issue in C. *for(printf("1st") ; printf("2nd") ; printf("3rd")) { break; }*
Question 25
With respect to following “for” loops in C, pick the best statement. Assume that there is a prior declaration of 'i' in all cases

```
for (i = 0; i < 10 ; i++) // (i)
for ( ; i < 10 ; i++) // (ii)
for (i = 0;  ; i++) // (iii)
for (i = 0; i < 10 ; ) // (iv)
for ( ; ; ) // (v)
```

AOnly (i) and (v) would compile successfully. Also (v) can be used as infinite loop.
BOnly (i) would compile successfully.
CAll would compile successfully but behavior of (ii), (iii) and (iv) would depend on compiler.
DAll would compile successfully.
Loops & Control Structure C Quiz - 104
Discuss it
Question 25 Explanation:
In C, any of the 3 expressions of “for” loop can be empty. The exact behavior of the loop depends on the body of the loop as well. Basically, all of the 3 expressions of loop can be put inside the loop body. So as per C language standard, all of the above are valid for loops.
Question 26
What’s going to happen when we compile and run the following C program?

```
#include "stdio.h"

int main()
{
  int i = 1, j;
  for ( ; ; )
  {
    if (i)
        j = --i;
    if (j < 10)
       printf("GeeksQuiz", j++);
    else
       break;
  }
  return 0;
}
```

ACompile Error.
BNo compile error but it will run into infinite loop printing GeeksQuiz.
CNo compile error and it’ll print GeeksQuiz 10 times.
DNo compile error but it’ll print GeeksQuiz 9 times.
Loops & Control Structure   **C Quiz - 104**
Discuss it
Question 26 Explanation:
Basically, even though the for loop doesn't have any of three expressions in parenthesis, the initialization, control and increment has been done in the body of the loop. So j would be initialized to 0 via first if. This if itself would be executed only once due to i--. Next if and else blocks are being used to check the value of j and existing the loop if j becomes 10. Please note that j is getting incremented in printf even though there's no format specifier in format string. That's why GeeksQuiz would be printed for j=0 to j=9 i.e. a total of 10 times.
Question 27
What’s going to happen when we compile and run the following C program?

```
#include "stdio.h"
int main()
{
 int j = 0;
 for ( ; j < 10 ; )
 {
   if (j < 10)
     printf("Geeks", j++);
   else
     continue;
   printf(“Quiz”);
 }
 return 0;
}
```

ACompile Error due to use of continue in for loop.
BNo compile error but it will run into infinite loop printing Geeks.
CNo compile error and it’ll print GeeksQuiz 10 times followed by Quiz once.
DNo compile error and it’ll print GeeksQuiz 10 times.
Loops & Control Structure   **C Quiz - 104**
Discuss it
Question 27 Explanation:
Here, initialization of j has been done outside *for* loop. *if* condition serves as control statement and prints GeeksQuiz 10 times due to two printfs. Please note that *continue* comes in picture when j becomes 10. At that time, second printf gets skipped and second expression in for is checked and it fails. Due to this, for loop ends.
Question 28
Which of the following statement is correct for **switch** controlling expression?
AOnly int can be used in “switch” control expression.
BBoth int and char can be used in “switch” control expression.
CAll types i.e. int, char and float can be used in “switch” control expression.
D“switch” control expression can be empty as well.
Loops & Control Structure   **C Quiz - 104**
Discuss it
Question 28 Explanation:
As per C standard, “*The controlling expression of a switch statement shall have integer type.*” Since char is prompted to integer in switch control expression, it’s allowed but float isn’t promoted. That’s why B is correct statement.
Question 29
Choose the best statement with respect to following three program snippets.

```
/*Program Snippet 1 with for loop*/
for (i = 0; i < 10; i++)
{
   /*statement1*/
   continue;
   /*statement2*/
}
```

```
/*Program Snippet 2 with while loop*/
i = 0;
while (i < 10)
{
  /*statement1*/
  continue;
  /*statement2*/
  i++;
}

/*Program Snippet 3 with do-while loop*/
i = 0;
do
{
  /*statement1*/
  continue;
  /*statement2*/
  i++;
}while (i < 10);
```

AAll the loops are equivalent i.e. any of the three can be chosen and they all will perform exactly same.
Bcontinue can't be used with all the three loops in C.
CAfter hitting the continue; statement in all the loops, the next expression to be executed would be controlling expression (i.e. i
DNone of the above is correct.
Loops & Control Structure   C Quiz - 105
Discuss it
Question 29 Explanation:
First and foremost, continue can be used in any of the 3 loops in C. In case of “for” loop, when continue is hit, the next expression to be executed would be i++ followed by controlling expression (i.e. i < 10). In case of “while” loop, when continue is hit, the next expression to be executed would be controlling expression (i.e. i < 10). In case of “do-while” loop, when continue is hit, the next expression to be executed would be controlling expression (i.e. i < 10). That’s why “while” and “do-while” loops would behave exactly same but not the “for” loop. Just to re-iterate, i++ would be executed in “for” loop when continue is hit.

Question 30
In the context of "**break**" and "**continue**" statements in C, pick the best statement.
A“break” can be used in “for”, “while” and “do-while” loop body.
B“continue” can be used in “for”, “while” and “do-while” loop body.
C“break” and “continue” can be used in “for”, “while”, “do-while” loop body and “switch” body.
D“break” and “continue” can be used in “for”, “while” and “do-while” loop body. But only “break” can be used in “switch” body.
E“break” and “continue” can be used in “for”, “while” and “do-while” loop body. Besides, “continue” and “break” can be used in “switch” and “if-else” body.
Loops & Control Structure   C Quiz - 105
Discuss it
Question 30 Explanation:
As per C standard, “continue” can be used in loop body only. “break” can be used in loop body and switch body only. That’s why correct answer is D.
Question 31
What would happen when we compile and run this program?

```
#include "stdio.h"
int main()
{
 int i;
 goto LOOP;
 for (i = 0 ; i < 10 ; i++)
 {
   printf("GeeksQuiz\n");
   LOOP:
    break;
 }
 return 0;
}
```

ANo compile error and it will print GeeksQuiz 10 times because goto label LOOP wouldn’t come in effect.
BNo compile error and it’ll print GeeksQuiz only once because goto label LOOP would come in picture only after entering for loop.
CCompile Error because any goto label isn’t allowed in for loop in C.
DNo compile error but behaviour of the program would depend on C compiler due to nondeterministic behaviour of goto statement.
ENo compile error and it will not print anything.
Loops & Control Structure   C Quiz - 105
Discuss it

Question 31 Explanation:
goto statement can be used inside a function and its label can point to anywhere in the same function. Here, for loop expressions i.e. i = 0 and i
Question 32
A typical “switch” body looks as follows:

```
switch (controlling_expression)
{
  case label1:
    /*label1 statements*/
    break;
  case label2:
    /*label1 statements*/
    break;
  default:
    /*Default statements*/
}
```

Which of the following statement is not correct statement?
A“switch” body may not have any “case” label at all and it would still compile.
B“switch” body may not have the “default” label and it would still compile.
C“switch” body may contain more than one “case” labels where the label value of these “case” is same and it would still compile. If “switch” controlling expression results in this “case” label value, the “case” which is placed first would be executed.
D“switch” body may not have any “break” statement and it would still compile.
E“switch” body can have the “default” label at first i.e. before all the other “case” labels. It would still compile.
Loops & Control Structure   C Quiz - 105
Discuss it
Question 32 Explanation:
In "switch" body, two "case" can't result in same value. Though having only "case" or only "default" is okay. In fact, "switch" body can be empty also.
Question 33
Which of the following is correct with respect to “Jump Statements” in C?
Agoto
Bcontinue
Cbreak
Dreturn
EAll of the above.
Loops & Control Structure   C Quiz - 105
Discuss it
Question 33 Explanation:
As per C standard, “A jump statement causes an unconditional jump to another place”. So if we see carefully, all “goto”, “continue”, “break” and “return” makes the program control jump unconditionally from one place to another. That’s why correct answer is E.
Question 34

```
#include <stdio.h>

int main()
{
    unsigned int i = 65000;
    while (i++ != 0);
    printf("%d", i);
    return 0;
}
```

AInfinite Loop
B0
C1
DRun Time Error
Loops & Control Structure   C Quiz - 113
Discuss it
Question 34 Explanation:
The result will be 1 but after a really long time because while loop will keep on going until i becomes 4294967295 (Assuming unsigned int is stored using 4 bytes) and as i highest limit of unsigned int is 4294967295 in next ++ operation it will become zero and we'll come out of loop and 1 will be printed. Since the time taken is long, on-line compiler may terminate the program with time limit exceeded error. If instead of unsigned int, you use unsigned short int then result (1) may come faster.
There are 34 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Question 1

```
#include<stdio.h>
int main()
{
   struct site
   {
      char name[] = "GeeksQuiz";
      int no_of_pages = 200;
   };
   struct site *ptr;
   printf("%d ", ptr->no_of_pages);
   printf("%s", ptr->name);
   getchar();
   return 0;
}
```

A200 GeeksQuiz
B200
CRuntime Error
DCompiler Error
Structure & Union
Discuss it
Question 1 Explanation:
When we declare a structure or union, we actually declare a new data type suitable for our purpose. So we cannot initialize values as it is not a variable declaration but a data type declaration.
Question 2
Assume that size of an integer is 32 bit. What is the output of following program?

```
#include<stdio.h>
struct st
{
    int x;
    static int y;
};

int main()
{
    printf("%d", sizeof(struct st));
    return 0;
}
```

A4
B8
CCompiler Error
DRuntime Error
Structure & Union
Discuss it
Question 2 Explanation:
In C, struct and union types cannot have static members. In C++, struct types are allowed to have static members, but union cannot have static members in C++ also.
Question 3

```
struct node
{
   int i;
   float j;
};
struct node *s[10];
```

The above C declaration define 's' to be (GATE CS 2000)
AAn array, each element of which is a pointer to a structure of type node
BA structure of 2 fields, each field being a pointer to an array of 10 elements
CA structure of 3 fields: an integer, a float, and an array of 10 elements
DAn array, each element of which is a structure of type node.
Structure & Union
Discuss it
Question 3 Explanation:
Refer Structures in C.
Question 4
Consider the following C declaration

```
struct {
    short s[5];
    union {
         float y;
         long z;
    }u;
} t;
```

Assume that objects of the type short, float and long occupy 2 bytes, 4 bytes and 8 bytes, respectively. The memory requirement for variable t, ignoring alignment considerations, is (GATE CS 2000)
A22 bytes
B14 bytes
C18 bytes
D10 bytes
Structure & Union
Discuss it
Question 4 Explanation:
Short array s[5] will take 10 bytes as size of short is 2 bytes. When we declare a union, memory allocated for the union is equal to memory needed for the largest member of it, and all members share this same memory space. Since u is a union, memory allocated to u will be max of float y(4 bytes) and long z(8 bytes). So, total size will be 18 bytes (10 + 8).
Question 5

```
#include<stdio.h>
struct st
{
    int x;
    struct st next;
};

int main()
{
    struct st temp;
    temp.x = 10;
    temp.next = temp;
    printf("%d", temp.next.x);
    return 0;
}
```

ACompiler Error
B10
CRuntime Error
DGarbage Value
Structure & Union
Discuss it
Question 5 Explanation:
A structure cannot contain a member of its own type because if this is allowed then it becomes impossible for compiler to know size of such struct. Although a pointer of same type can be a member because pointers of all types are of same size and compiler can calculate size of struct
Question 6
Which of the following operators can be applied on structure variables?
AEquality comparison (==)
BAssignment (=)
CBoth of the above
DNone of the above
Structure & Union
Discuss it
Question 6 Explanation:
A structure variable can be assigned to other using =, but cannot be compared with other using ==
Question 7

```
union test
{
    int x;
    char arr[8];
    int y;
};

int main()
{
    printf("%d", sizeof(union test));
    return 0;
}
```

Predict the output of above program. Assume that the size of an integer is 4 bytes and size of character is 1 byte. Also assume that there is no alignment needed.
A12
B16
C8
DCompiler Error
Structure & Union
Discuss it
Question 7 Explanation:
When we declare a union, memory allocated for a union variable of the type is equal to memory needed for the largest member of it, and all members share this same memory space. In above example, "char arr[8]" is the largest member. Therefore size of union test is 8 bytes.
Question 8

```
union test
{
    int x;
    char arr[4];
    int y;
};

int main()
{
    union test t;
    t.x = 0;
    t.arr[1] = 'G';
    printf("%s", t.arr);
    return 0;
}
```

Predict the output of above program. Assume that the size of an integer is 4 bytes and size of character is 1 byte. Also assume that there is no alignment needed.

ANothing is printed

BG

CGarbage character followed by 'G'

DGarbage character followed by 'G', followed by more garbage characters

ECompiler Error

Structure & Union

Discuss it

Question 8 Explanation:

Since x and arr[4] share the same memory, when we set x = 0, all characters of arr are set as 0. O is ASCII value of '\0'. When we do "t.arr[1] = 'G'", arr[] becomes "\0G\0\0". When we print a string using "%s", the printf function starts from the first character and keeps printing till it finds a \0. Since the first character itself is \0, nothing is printed.

Question 9

```
# include <iostream>
# include <string.h>
using namespace std;

struct Test
{
  char str[20];
};

int main()
{
  struct Test st1, st2;
  strcpy(st1.str, "GeeksQuiz");
  st2 = st1;
  st1.str[0] = 'S';
  cout << st2.str;
  return 0;
}
```

ASegmentation Fault

BSeeksQuiz

CGeeksQuiz

DCompiler Error

Structure & Union

Discuss it

Question 9 Explanation:

Array members are deeply copied when a struct variable is assigned to another one. See Are array members deeply copied? for more details.

Question 10

Predict the output of following C program

```
#include<stdio.h>
struct Point
{
  int x, y, z;
};

int main()
{
  struct Point p1 = {.y = 0, .z = 1, .x = 2};
  printf("%d %d %d", p1.x, p1.y, p1.z);
  return 0;
}
```

ACompiler Error

B2 0 1

C0 1 2

D2 1 0

Structure & Union

Discuss it

Question 10 Explanation:

Refer designated Initialization discussed here.

Question 11
The following C declarations

```
struct node
{
   int i;
   float j;
};
struct node *s[10] ;
```

define s to be
AAn array, each element of which is a pointer to a structure of type node
BA structure of 2 fields, each field being a pointer to an array of 10 elements
CA structure of 3 fields: an integer, a float, and an array of 10 elements
DAn array, each element of which is a structure of type node.
Structure & Union   GATE-CS-2000
Discuss it
Question 11 Explanation:

```
// The following code declares a structure
struct node
{
   int i;
   float j;
};

// The following code declares and defines an array s[] each
// element of which is a pointer to a structure of type node
struct node *s[10] ;
```

Question 12
Anyone of the followings can be used to declare a node for a singly linked list. If we use the first declaration, “struct node * nodePtr;” would be used to declare pointer to a node. If we use the second declaration, “NODEPTR nodePtr;” can be used to declare pointer to a node.

```
/* First declaration */
struct node {
int data;
struct node * nextPtr;
};

/* Second declaration */
typedef struct node{
int data;
NODEPTR nextPtr;
} * NODEPTR;
```

ATRUE
BFALSE
Structure & Union   C Quiz - 108
Discuss it
Question 12 Explanation:
The *typedef* usage is incorrect. Basically, we can’t use yet to be typedef-ed data type inside while applying *typedef* itself. Here, NODEPTR is yet to be defined (i.e. typedef-ed) and we are using NODEPTR inside the struct itself.
Question 13
Anyone of the following can be used to declare a node for a singly linked list and “NODEPTR nodePtr;” can be used to declare pointer to a node using any of the following

```
/* First declaration */
typedef struct node
{
 int data;
 struct node *nextPtr;
}* NODEPTR;

/* Second declaration */
struct node
{
 int data;
 struct node * nextPtr;
};
typedef struct node * NODEPTR;
```

ATRUE
BFALSE
Structure & Union   C Quiz - 108
Discuss it
Question 13 Explanation:
Yes. Both are equivalent. Either of the above declarations can be used for “NODEPTR nodePtr;”. In fact, first one is the compact form of second one.
Question 14
In the following program snippet, both s1 and s2 would be variables of structure type defined as below and there won't be any compilation issue.

```
typedef struct Student
{
 int rollno;
 int total;
} Student;

Student s1;
struct Student s2;
```

ATRUE
BFALSE
Structure & Union   C Quiz - 109
Discuss it
Question 14 Explanation:
At first, it may seem that having same ‘struct tag name’ and ‘typedef name’ would cause issue here. But it’s perfectly fine for both of them having same name. s1 is defined using typedef name Student while s2 is defined using struct tag name Student.
Question 15
Pick the best statement for the below program:

```
#include "stdio.h"

int main()
{
 struct {int a[2];} arr[] = {{1},{2}};

 printf("%d %d %d %d",arr[0].a[0],arr[0].a[1],arr[1].a[0],arr[1].a[1]);

 return 0;
}
```

ACompile error because arr has been defined using struct type incorrectly. First struct type should be defined using tag and then arr should be defined using that tag.
BCompile error because apart from definition of arr, another issue is in the initialization of array of struct i.e. arr[].
CCompile error because of initialization of array of struct i.e. arr[].

DNo compile error and it’ll print 1 2 0 0
ENo compile error and it’ll print 1 0 2 0
Structure & Union   C Quiz - 112
Discuss it
Question 15 Explanation:
Here, struct type definition and definition of arr using that struct type has been done in the same line. This is okay as per C standard. Even initialization is also correct. The point to note is that array size of arr[] would be 2 i.e. 2 elements of this array of this struct type. This is decided due to the way it was initialized above. Here, arr[0].a[0] would be 1 and arr[1].a[0] would be 2. The remaining elements of the array would be ZERO. correct answer is E.
Question 16
Pick the best statement for the below program snippet:

```
struct {int a[2];} arr[] = {1,2};
```

ANo compile error and it’ll create array arr of 2 elements. Each of the element of arr contain a struct field of int array of 2 elements. arr[0].a[0] would be 1 and arr[1].a[0] would be 2.
BNo compile error and it’ll create array arr of 2 elements. Each of the element of arr contain a struct field of int array of 2 elements. arr[0].a[0] would be 1 and arr[0].a[1] would be 2. The second element arr[1] would be ZERO i.e. arr[1].a[0] and arr[1].a[1] would be 0.
CNo compile error and it’ll create array arr of 1 element. Each of the element of arr contain a struct field of int array of 2 elements. arr[0].a[0] would be 1 and arr[0].a[1] would be 2.
Structure & Union   C Quiz - 112
Discuss it
Question 16 Explanation:
Since size of array arr isn’t given explicitly, it would be decided based on the initialization here. Without any curly braces, arr is initialized sequentially i.e. arr[0].a[0] would be 1 and arr[0].a[1] would be 2. There’s no further initialization so size of arr would be 1. Correct answer is

C.
Question 17
Pick the best statement for the below program:

```
#include "stdio.h"

int main()
{
 struct {int a[2], b;} arr[] = {[0].a = {1}, [1].a = {2}, [0].b = 1, [1].b = 2};

 printf("%d %d %d and",arr[0].a[0],arr[0].a[1],arr[0].b);
 printf("%d %d %d\n",arr[1].a[0],arr[1].a[1],arr[1].b);

 return 0;
}
```

ACompile error because struct type (containing two fields i.e. an array of int and an int) has been specified along with the definition of array arr[] of this struct type.
BCompile error because of incorrect syntax for initialization of array arr[].
CNo compile error and two elements of arr[] would be defined and initialized. Output would be “1 0 1 and 2 0 2”.
DNo compile error and two elements of arr[] would be defined and initialized. Output would be “1 X 1 and 2 X 2” where X is some garbage random number.
Structure & Union **C Quiz - 112**
Discuss it
Question 17 Explanation:
In C, designators can be used to provide explicit initialization. For an array, elements which aren’t initialized explicitly in program are set as ZERO. That’s why correct answer is C.
Question 18
Pick the best statement for the below program:

```
#include "stdio.h"

int main()
{
 struct {int i; char c;} myVar = {.c ='A',.i = 100};
 printf("%d %c",myVar.i, myVar.c);
 return 0;
}
```

ACompile error because struct type (containing two fields of dissimilar type i.e. an int and a char) has been mentioned along with definition of myVar of that struct type.
BCompile error because of incorrect syntax of initialization of myVar. Basically, member of operator (i.e. dot .) has been used without myVar.
CCompile error for not only B but for incorrect order of fields in myVar i.e. field c has been initialized first and then field i has been initialized.
DNo compile error and it’ll print 100 A.
Structure & Union **C Quiz - 112**
Discuss it
Question 18 Explanation:
As per C language, initialization of a variable of complete data type can be done at the time of definition itself. Also, struct fields/members can be initialized out of order using field name and using dot operator without myVar is ok as per C. Correct answer is D.
Question 19
Pick the best statement for the below program:

```
#include "stdio.h"

int main()
{
 union {int i1; int i2;} myVar = {.i2 =100};
 printf("%d %d",myVar.i1, myVar.i2);
 return 0;
}
```

ACompile error due to incorrect syntax of initialization.
BNo compile error and it’ll print “0 100”.
CNo compile error and it’ll print “100 100”.
Structure & Union **C Quiz - 112**
Discuss it
Question 19 Explanation:
Since fields/members of union share same memory, both i1 and i2 refer to same location. Also, since both i1 and i2 are of same type, initializing one would initialize the other as well implicitly. So answer is C.

There are 19 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Question 1

```
#include <stdio.h>
#define PRINT(i, limit) do \
                        { \
                           if (i++ < limit) \
                           { \
                              printf("GeeksQuiz\n"); \
                              continue; \
                           } \
                        }while(1)

int main()
{
   PRINT(0, 3);
   return 0;
}
```

How many times **GeeksQuiz** is printed in the above program?
A1
B3
C4
DCompile-time error
Macro & Preprocessor
Discuss it
Question 1 Explanation:
The **PRINT** macro gets expanded at the pre-processor time i.e. before the compilation time. After the macro expansion, the if expression becomes: **if (0++ . Since 0 is a constant figure and represents only r-value, applying increment operator gives compile-time error: lvalue required. lvalue means a memory location with some address.**
Question 2

```
#include <stdio.h>
#if X == 3
   #define Y 3
#else
   #define Y 5
#endif

int main()
{
   printf("%d", Y);
   return 0;
}
```

What is the output of the above program?
A3
B5
C3 or 5 depending on value of X
DCompile time error
Macro & Preprocessor
Discuss it
Question 2 Explanation:
In the first look, the output seems to be compile-time error because macro X has not been defined. In C, if a macro is not defined, the pre-processor assigns 0 to it by default. Hence, the control goes to the conditional else part and 5 is printed. See the next question for better understanding.
Question 3
What is the output of following program?

```
#include <stdio.h>
#define macro(n, a, i, m) m##a##i##n
#define MAIN macro(n, a, i, m)
```

```
int MAIN()
{
   printf("GeeksQuiz");
   return 0;
}
```

ACompiler Error
BGeeksQuiz
CMAIN
Dmain
Macro & Preprocessor
Discuss it
Question 3 Explanation:
The program has a preprocessor that replaces "MAIN" with "macro(n, a, i, m)". The line "macro(n, a, i, m)" is again replaced by main. The key thing to note is token pasting operator ## which concatenates parameters to macro.
Question 4

```
#include <stdio.h>
#define X 3
#if !X
   printf("Geeks");
#else
   printf("Quiz");

#endif
int main()
{
      return 0;
}
```

AGeeks
BQuiz
CCompiler Error
DRuntime Error
Macro & Preprocessor
Discuss it
Question 4 Explanation:
A program is converted to executable using following steps 1) Preprocessing 2) C code to object code conversion 3) Linking The first step processes macros. So the code is converted to following after the preprocessing step.

```
printf("Quiz");
int main()
{
      return 0;
}
```

The above code produces error because printf() is called outside main. The following program works fine and prints "Quiz"

```
#include
#define X 3

int main()
{
#if !X
   printf("Geeks");
#else
   printf("Quiz");

#endif
   return 0;
}
```

Question 5

```
#include <stdio.h>
#define ISEQUAL(X, Y) X == Y
int main()
{
   #if ISEQUAL(X, 0)
      printf("Geeks");
```

```
    #else
       printf("Quiz");
    #endif
    return 0;
}
```

Output of the above program?
AGeeks
BQuiz
CAny of Geeks or Quiz
DCompile time error
Macro & Preprocessor
Discuss it
Question 5 Explanation:
The conditional macro *#if ISEQUAL(X, 0)* is expanded to *#if X == 0*. After the pre-processing is over, all the undefined macros are initialized with default value 0. Since macro X has not been defined, it is initialized with 0. So, **Geeks** is printed.
Question 6

```
#include <stdio.h>
#define square(x) x*x
int main()
{
  int x;
  x = 36/square(6);
  printf("%d", x);
  return 0;
}
```

A1
B36
C0
DCompiler Error
Macro & Preprocessor
Discuss it
Question 6 Explanation:
Preprocessor replaces square(6) by 6*6 and the expression becomes x = 36/6*6 and value of x is calculated as 36. Note that the macro will also fail for expressions "x = square(6-2)" If we want correct behavior from macro square(x), we should declare the macro as

```
#define square(x) ((x)*(x))
```

Question 7
Output?

```
# include <stdio.h>
# define scanf  "%s Geeks Quiz "
int main()
{
   printf(scanf, scanf);
   return 0;
}
```

ACompiler Error
B%s Geeks Quiz
CGeeks Quiz
D%s Geeks Quiz Geeks Quiz
Macro & Preprocessor
Discuss it
Question 7 Explanation:
After pre-processing phase of compilation, printf statement will become. printf("%s Geeks Quiz ", "%s Geeks Quiz "); Now you can easily guess why output is "%s Geeks Quiz Geeks Quiz".
Question 8

```
#include <stdio.h>
#define a 10
int main()
{
  printf("%d ",a);

  #define a 50

  printf("%d ",a);
  return 0;
}
```

ACompiler Error
B10 50
C50 50
D10 10
Macro & Preprocessor
Discuss it
Question 8 Explanation:
Preprocessor doesn't give any error if we redefine a preprocessor directive. It may give warning though. Preprocessor takes the most recent value before use of and put it in place of a.
Question 9
Output?

```
#include<stdio.h>
#define f(g,g2) g##g2
int main()
{
  int var12 = 100;
  printf("%d", f(var,12));
  return 0;
}
```

A100
BCompiler Error
C0
D1
Macro & Preprocessor
Discuss it
Question 9 Explanation:
The operator ## is called “Token-Pasting” or “Merge” Operator. It merges two tokens into one token. So, after preprocessing, the main function becomes as follows, and prints 100.

```
int main()
{
  int var12 = 100;
  printf("%d", var12);
  return 0;
}
```

Question 10
Which file is generated after pre-processing of a C program?
A.p
B.i
C.o
D.m
Macro & Preprocessor
Discuss it
Question 10 Explanation:
After the pre-processing of a C program, a **.i** file is generated which is passed to the compiler for compilation.
Question 11
What is the use of "#pragma once"?
AUsed in a header file to avoid its inclusion more than once.
BUsed to avoid multiple declarations of same variable.
CUsed in a c file to include a header file at least once.
DUsed to avoid assertions
Macro & Preprocessor
Discuss it
Question 12
Predict the output of following program?

```
#include <stdio.h>
#define MAX 1000
int main()
{
  int MAX = 100;
  printf("%d ", MAX);
  return 0;
}
```

A1000
B100
CCompiler Error
DGarbage Value
Macro & Preprocessor
Discuss it
Question 12 Explanation:
After preprocessing stage of compilation, the function main() changes to following

```
int main()
{
  int 1000 = 100;  // COMPILER ERROR: expected unqualified-id before numeric constant
  printf("%d ", 1000);
  return 0;
}
```

Question 13
Output of following C program?

```
#include<stdio.h>
#define max abc
#define abc 100

int main()
{
   printf("maximum is %d", max);
   return 0;
}
```

Amaximum is 100
Babcimum is 100
C100imum is 100
Dabcimum is abc
Macro & Preprocessor
Discuss it
Question 14

```
#include <stdio.h>
#define get(s) #s

int main()
{
   char str[] = get(GeeksQuiz);
   printf("%s", str);
   return 0;
}
```

ACompiler Error
B#GeeksQuiz
CGeeksQuiz
DGGeeksQuiz
Macro & Preprocessor
Discuss it
Question 14 Explanation:
The preprocessing operator '#' is used to convert a string argument into a string constant.
Question 15
Suppose someone writes increment macro (i.e. which increments the value by one) in following ways:

```
#define INC1(a) ((a)+1)

#define INC2 (a) ((a)+1)

#define INC3( a ) (( a ) + 1)

#define INC4 ( a ) (( a ) + 1)
```

Pick the correct statement for the above macros.
AOnly INC1 is correct.
BAll (i.e. INC1, INC2, INC3 and INC4) are correct.
COnly INC1 and INC3 are correct.
DOnly INC1 and INC2 are correct.

Macro & Preprocessor   C Quiz - 110
Discuss it
Question 15 Explanation:
In C, for macros with arguments, there can't be any space between *macro name* and *open parenthesis*. That's why only INC1 and INC3 are correct. Basically, "#define INC2 (a) ((a)+1)" results in "INC2" expansion to "(a) ((a)+1)" which is not the desired expansion.
Question 16
The following program won't compile because there're space between macro name and open parenthesis.

```
#include "stdio.h"

#define MYINC  ( a ) ( ( a ) + 1 )

int main()
{

 printf("GeeksQuiz!");

 return 0;
}
```

ATRUE
BFALSE
Macro & Preprocessor   C Quiz - 110
Discuss it
Question 16 Explanation:
Please note that #define is a preprocessor directive i.e. it's processed before actual compilation takes place. In the above program snippet MYINC isn't used in the program anywhere. So even though MYINC doesn't perform the intended behaviour i.e. it won't increment a, but MYINC is a valid macro. If we had used MYINC anywhere in the program, it would have been replaced with "(a) ((a) + 1)". So above program will compile and run without any issue.
Question 17
Typically, library header files in C (e.g. stdio.h) contain not only declaration of functions and macro definitions but they contain definition of user defined data types (e.g. struct, union etc), typedefs and definition of global variables as well. So if we include the same header file more than once in a C program, it would result in compile issue because re-definition of many of the constructs of the header file would happen. So it means the following program will give compile error.

```
#include “stdio.h”
#include “stdio.h”
#include “stdio.h”

int main()
{
 printf(“Whether this statement would be printed?”)
 return 0;
}
```

ATRUE
BFALSE
Macro & Preprocessor   C Quiz - 110
Discuss it
Question 17 Explanation:
It's okay to include library header files multiple times in a program. But actually the content of the header file is included only once. The way it's achieved is due to usage of "#ifndef", "#define" and "#endif". That's why it's recommended to use these preprocesor macros even in user defined header files. For an example and usage of this, please check out the "Discuss it" of this question.
Question 18
What is the output for the following code snippet?

```
#include<stdio.h>
#define A -B
#define B -C
#define C 5

int main()
{
  printf("The value of A is %d\n", A);
  return 0;
}
```

This question is contributed by **Aastha Anand**.
AThe value of A is 4
BThe value of A is 5

CCompilation Error
DRuntime Error
Macro & Preprocessor
Discuss it
There are 18 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Question 1
Predict the output of below program:

```
 #include <stdio.h>

int main()
{
 int arr[5];

 // Assume that base address of arr is 2000 and size of integer
        // is 32 bit
 arr++;
 printf("%u", arr);

 return 0;
}
```

A2002
B2004
C2020
Dlvalue required
Arrays
Discuss it
Question 1 Explanation:
Array name in C is implemented by a constant pointer. It is not possible to apply increment and decrement on constant types.
Question 2
Predict the output of below program:

```
#include <stdio.h>

int main()
{
   int arr[5];
   // Assume base address of arr is 2000 and size of integer is 32 bit
   printf("%u %u", arr + 1, &arr + 1);

   return 0;
}
```

A2004 2020
B2004 2004
C2004 Garbage value
DThe program fails to compile because Address-of operator cannot be used with array name
Arrays
Discuss it
Question 2 Explanation:
Name of array in C gives the address(except in sizeof operator) of the first element. Adding 1 to this address gives the **address plus the sizeof type** the array has. Applying the **Address-of** operator before the array name gives the address of the whole array. Adding 1 to this address gives the **address plus the sizeof whole array**.
Question 3
What is output?

```
# include <stdio.h>

void print(int arr[])
{
   int n = sizeof(arr)/sizeof(arr[0]);
   int i;
   for (i = 0; i < n; i++)
      printf("%d ", arr[i]);
}

int main()
{
   int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};
   print(arr);
   return 0;
}
```

A1, 2, 3, 4, 5, 6, 7, 8
BCompiler Error
C1
DRun Time Error
Arrays
Discuss it
Question 3 Explanation:
See http://www.geeksforgeeks.org/using-sizof-operator-with-array-paratmeters/ for explanation.
Question 4
Output of following program?

```
#include<stdio.h>

int main()
{
  int a[] = {1, 2, 3, 4, 5, 6};
  int *ptr = (int*)(&a+1);
  printf("%d ", *(ptr-1) );
  return 0;
}
```

A1
B2
C6
DRuntime Error
Arrays
Discuss it
Question 4 Explanation:
&a is address of the whole array a[]. If we add 1 to &a, we get “base address of a[] + sizeof(a)”. And this value is typecasted to int *. So ptr points the memory just after 6 is stored. ptr is typecasted to "int *" and value of *(ptr-1) is printed. Since ptr points memory after 6, ptr – 1 points to 6.
Question 5
Consider the following C-function in which a[n] and b[m] are two sorted integer arrays and c[n + m] be another integer array.

```
void xyz(int a[], int b [], int c[])
{
  int i, j, k;
  i = j = k = O;
  while ((i<n) && (j<m))
     if (a[i] < b[j]) c[k++] = a[i++];
     else c[k++] = b[j++];
}
```

Which of the following condition(s) hold(s) after the termination of the while loop? (GATE CS 2006) (i) j
Aonly (i)
Bonly (ii)
Ceither (i) or (ii) but not both
Dneither (i) nor (ii)
Arrays
Discuss it
Question 5 Explanation:
The function xyz() is similar to merge() of mergeSort(). The condition (i) is true if the last inserted element in c[] is from a[] and condition (ii) is true if the last inserted element is from b[].

Question 6
Assume the following C variable declaration

```
int *A [10], B[10][10];
```

Of the following expressions I A[2] II A[2][3] III B[1] IV B[2][3] which will not give compile-time errors if used as left hand sides of assignment statements in a C program (GATE CS 2003)?
AI, II, and IV only
BII, III, and IV only
CII and IV only
DIV only
Arrays
Discuss it
Question 6 Explanation:
See following for explanation.

```
int main()
{
  int *A[10], B[10][10];
  int C[] = {12, 11, 13, 14};

  /* No problem with below statement as A[2] is a pointer
     and we are assigning a value to pointer */
  A[2] = C;

  /* No problem with below statement also as array style indexing
      can be done with pointers*/
  A[2][3] = 15;

  /* Simple assignment to an element of a 2D array*/
  B[2][3]  = 15;

  printf("%d %d", A[2][0], A[2][3]);
  return 0;
}
```

Question 7
Consider the following declaration of a ‘two-dimensional array in C:

```
char a[100][100];
```

Assuming that the main memory is byte-addressable and that the array is stored starting from memory address 0, the address of a[40][50] is (GATE CS 2002)
A4040
B4050
C5040
D5050
Arrays
Discuss it
Question 7 Explanation:

```
Address of a[40][50] = Base address + 40*100*element_size + 50*element_size
                      = 0 + 4000*1 + 50*1
                     = 4050
```

Question 8
Which of the following is true about arrays in C.
AFor every type T, there can be an array of T.
BFor every type T except void and function type, there can be an array of T.
CWhen an array is passed to a function, C compiler creates a copy of array.
D2D arrays are stored in column major form
Arrays
Discuss it
Question 8 Explanation:
In C, we cannot have an array of void type and function types. For example, below program throws compiler error

```
int main()
{
   void arr[100];
}
```

But we can have array of void pointers and function pointers. The below program works fine.

```
int main()
{
    void *arr[100];
}
```

See examples of function pointers for details of array function pointers.
Question 9
Predict the output of the below program:

```
#include <stdio.h>
#define SIZE(arr) sizeof(arr) / sizeof(*arr);
void fun(int* arr, int n)
{
    int i;
    *arr += *(arr + n - 1) += 10;
}

void printArr(int* arr, int n)
{
    int i;
    for(i = 0; i < n; ++i)
        printf("%d ", arr[i]);
}

int main()
{
    int arr[] = {10, 20, 30};
    int size = SIZE(arr);
    fun(arr, size);
    printArr(arr, size);
    return 0;
}
```

A20 30 40
B20 20 40
C50 20 40
DCompile-time error
Arrays
Discuss it
Question 9 Explanation:
The crux of the question lies in the expression: **arr += *(arr + n - 1) += 10;* The composite operator (here +=) has right to left associativity. First 10 is added to the last element of the array. The result is then added to the first element of the array.
Question 10
Predict output of following program

```
int main()
{
    int i;
    int arr[5] = {1};
    for (i = 0; i < 5; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

A1 followed by four garbage values
B1 0 0 0 0
C1 1 1 1 1
D0 0 0 0 0
Arrays
Discuss it
Question 10 Explanation:
In C/C++, if we initialize an array with fewer members, all remaining members are automatically initialized as 0. For example, the following statement initializes an array of size 1000 with values as 0.

```
 int arr[1000] = {0};
```

Question 11
Does C perform array out of bound checking? What is the output of the following program?

```
int main()
```

```
{
    int i;
    int arr[5] = {0};
    for (i = 0; i <= 5; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

ACompiler Error: Array index out of bound.
BThe always prints 0 five times followed by garbage value
CThe program always crashes.
DThe program may print 0 five times followed by garbage value, or may crash if address (arr+5) is invalid.
Arrays
Discuss it
Question 12

```
#include <stdio.h>

int main()
{
    int a[][] = {{1,2},{3,4}};
    int i, j;
    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
            printf("%d ", a[i][j]);
    return 0;
}
```

A1 2 3 4
BCompiler Error in line " int a[][] = {{1,2},{3,4}};"
C4 garbage values
D4 3 2 1
Arrays
Discuss it
Question 12 Explanation:
There is compilation error in the declaration " int a[][] = {{1,2},{3,4}};". Except the first dimension, every other dimension must be specified. int arr[] = {5, 6, 7, 8} //valid int arr[][5] = {}; //valid int arr[][] = {}; //invalid int arr[][10][5] = {}; //valid int arr[][][5] = {}; //invalid
Question 13

```
#include<stdio.h>
int main()
{
    int a[10][20][30] = {0};
    a[5][2][1] = 2;
    return 0;
}
```

Which of the following will print the value 2 for the above code?
Aprintf("%d",*(((a+5)+2)+1));
Bprintf("%d",***((a+5)+2)+1);
Cprintf("%d",*(*(*(a+5)+2)+1));
DNone of these
Arrays
Discuss it
Question 14

```
#include <stdio.h>
int main()
{
    char p;
    char buf[10] = {1, 2, 3, 4, 5, 6, 9, 8};
    p = (buf + 1)[5];
    printf("%d\n", p);
    return 0;
}
```

A5
B6
C9
DNone of the above
Arrays

Discuss it
Question 15
For a C program accessing X[i][j][k], the following intermediate code is generated by a compiler. Assume that the size of an integer is 32 bits and the size of a character is 8 bits.

```
t0 = i ∗ 1024
t1 = j ∗ 32
t2 = k ∗ 4
t3 = t1 + t0
t4 = t3 + t2
t5 = X[t4]
```

Which one of the following statements about the source code for the C program is CORRECT?
AX is declared as “int X[32][32][8]”.
BX is declared as “int X[4][1024][32]”.
CX is declared as “char X[4][32][8]”.
DX is declared as “char X[32][16][2]”.
Arrays GATE-CS-2014-(Set-2)
Discuss it
Question 15 Explanation:
The final expression can be simplified in form ofi, j and k by following the intermediate code steps in reverse order

```
t5 = X[t4]
  = X[t3 + t2]
  = X[t1 + t0 + t2]
  = X[i*1024 + j*32 + k*4]
  = X + i*1024 + j*32 + k*4
```

Since k is multiplied by 4, the array must be an int array. We are left with 2 choices (A and B) among the 4 given choices. X[i][j][k]'th element in one dimensional array is equivalent to X[i*M*L + j*L + k]'th element in one dimensional array (Note that multi-dimensional arrays are stored in row major order in C). So we get following equations

```
j*L*4 = j*32, we get L = 8 (4 is the sizeof(int))
i*1024 = i*M*L*4, we get M = 1024/32 = 32
```

Therefore option A is the only correct option as M and L are 32 and 8 respectively only in option A.
Question 16
What’s the meaning of following declaration in C language?

```
int (*p)[5];
```

AIt will result in compile error because there shouldn't be any parenthesis i.e. “int *p[5]” is valid.
Bp is a pointer to 5 integers.
Cp is a pointer to integer array.
Dp is an array of 5 pointers to integers.
Ep is a pointer to an array of 5 integers
Arrays C Quiz - 106
Discuss it
Question 16 Explanation:
Here p is basically a pointer to integer array of 5 integers. In case of “int *p[5]”, p is array of 5 pointers to integers.
Question 17
For the following declaration of a function in C, pick the best statement

```
int [] fun(void (*fptr)(int *));
```

AIt will result in compile error.
BNo compile error. fun is a function which takes a function pointer fptr as argument and return an array of int.
CNo compile error. fun is a function which takes a function pointer fptr as argument and returns an array of int. Also, fptr is a function pointer which takes int pointer as argument and returns void.
DNo compile error. fun is a function which takes a function pointer fptr as argument and returns an array of int. The array of int depends on the body of fun i.e. what size array is returned. Also, fptr is a function pointer which takes int pointer as argument and returns void.
Arrays C Quiz - 107
Discuss it
Question 17 Explanation:
As per C standard, a function can’t have an explicit array as return type. That’s why the above would result in compile error. There’re indirect ways if we need an array as an output of a function call. For example, a pointer can be returned by function by *return* statement while providing the size of array via other means. Alternatively, function argument can be used for this.
Question 18
In a C file (say sourcefile1.c), an array is defined as follows. Here, we don’t need to mention array arr size explicitly in [] because the size would be determined by the number of elements used in the initialization.

```
int arr[] = {1,2,3,4,5};
```

In another C file (say sourcefile2.c), the same array is declared for usage as follows:

```
extern int arr[];
```

In sourcefile2.c, we can use sizeof() on arr to find out the actual size of arr.
ATRUE
BFALSE
Arrays **C Quiz - 108**
Discuss it
Question 18 Explanation:
First thing first, sizeof() operator works at compile time. So usage of sizeof on arr in sourcefile2.c won't work because arr in sourcefile2.c is an incomplete type. Please note that arr in sourcefile1.c is a complete type because size of array got determined at compile time due to initialization.
Question 19
Find out the correct statement for the following program.

```
#include "stdio.h"

int * arrPtr[5];

int main()
{
 if(*(arrPtr+2) == *(arrPtr+4))
 {
   printf("Equal!");
 }
 else
 {
  printf("Not Equal");
 }
 return 0;
}
```

ACompile Error
BIt'll always print Equal.
CIt'll always print Not Equal.
DSince elements of arrPtr aren't initialized in the program, it'll print either Equal or Not Equal.
Arrays **C Quiz - 109**
Discuss it
Question 19 Explanation:
Here arrPtr is a global array of pointers to int. It should be noted that global variables such arrPtr are initialized to ZERO. That's why all are elements of arrPtr are initialized implicitly to ZERO i.e. correct answer is b.
Question 20
In C, 1D array of int can be defined as follows and both are correct.

```
int array1D[4] = {1,2,3,4};
int array1D[] = {1,2,3,4};
```

But given the following definitions (along-with initialization) of 2D arrays

```
int array2D[2][4] = {1,2,3,4,5,6,7,8}; /* (i) */
int array2D[][4] = {1,2,3,4,5,6,7,8}; /* (ii) */
int array2D[2][] = {1,2,3,4,5,6,7,8}; /* (iii) */
int array2D[][] = {1,2,3,4,5,6,7,8}; /* (iv) */
```

Pick the correct statements.
AOnly (i) is correct.
BOnly (i) and (ii) are correct.
COnly (i), (ii) and (iii) are correct.
DAll (i), (ii), (iii) and (iv) are correct.

Arrays **C Quiz - 110**
Discuss it
Question 20 Explanation:
First of all, C language doesn't provide any true support for 2D array or multidimensional arrays. A 2D array is simulated via 1D array of arrays. So a 2D array of int is actually a 1D array of array of int. Another important point is that array size can be derived from its initialization but that's applicable for first dimension only. It means that 2D array need to have an explicit size of 2nd dimension. Similarly, for a 3D array, 2nd and 3rd dimensions need to have explicit size. That's why only (i) and (ii) are correct. But array2D[2][] and array2D[][] are of incomplete type because their complete size can't derived even from the initialization.

Question 21
Pick the best statement for the below:

```
int arr[50] = {0,1,2,[47]=47,48,49};
```

AThis isn't allowed in C and it'll give compile error
BThis is allowed in C as per standard. Basically, it'll initialize arr[0], arr[1], arr[2], arr[47], arr[48] and arr[49] to 0,1,2,47,48 and 49 respectively. The remaining elements of the array would be initialized to 0.
Arrays **C Quiz - 111**
Discuss it
Question 21 Explanation:
In C, initialization of array can be done for selected elements as well. By default, the initializer start from 0th element. Specific elements in array can be specified by []. It should be noted that the remaining elements (i.e. the ones not mentioned in array initialization) would be initialized to 0. For example, "int arr[10] = {100, [5]=100,[9]=100}" is also legal in C. This initializes arr[0], arr[5] and arr[9] to 100. All the remaining elements would be 0.
Question 22
Pick the best statement for the below program:

```
#include "stdio.h"

void fun(int n)
{
   int idx;
   int arr1[n] = {0};
   int arr2[n];

   for (idx=0; idx<n; idx++)
       arr2[idx] = 0;
}

int main()
{
   fun(4);
   return 0;
}
```

ADefinition of both arr1 and arr2 is incorrect because variable is used to specify the size of array. That's why compile error.
BApart from definition of arr1 arr2, initialization of arr1 is also incorrect. arr1 can't be initialized due to its size being specified as variable. That's why compile error.
CInitialization of arr1 is incorrect. arr1 can't be initialized due to its size being specified as variable. That's why compile error.
DNo compile error. The program would define and initializes both arrays to ZERO.
Arrays **C Quiz - 111**
Discuss it
Question 22 Explanation:
There's no issue with definition of *arr1* and *arr2*. In definition of these arrays, the mention of array size using variable is ok as per C standard but these types of arrays can't be initialized at the time of definition. That's why initialization of *arr1* is incorrect. But initialization of *arr2* is done correctly. Right answer is C.
Question 23
Pick the best statement for the below program:

```
#include "stdio.h"

int size = 4;
int arr[size];

int main()
{
 if(arr[0])
  printf("Initialized to ZERO");
 else
  printf("Not initialized to ZERO");

 return 0;
}
```

ANo compile error and it'll print "Initialized to ZERO".
BNo compile error and it'll print "Not initialized to ZERO".
CCompile error because size of arr has been defined using variable outside any function.
DNo compile error and it'll print either "Initialized to ZERO" or "Not initialized to ZERO" depending on what value is present at arr[0] at a particular run of the program.

Arrays **C Quiz - 111**
Discuss it
Question 23 Explanation:
An array whose size is specified as variable can’t be defined out any function. It can be defined only inside a function. So putting *arr[size]* outside main() would result in compile error. Answer is C.
Question 24
Let a be an array containing n integers in increasing order. The following algorithm determines whether there are two distinct numbers in the array whose difference is a specified number S > 0.

```
i = 0;
j = 1;
while (j < n )
{
   if (E) j++;
   else if (a[j] - a[i] == S) break;
   else i++;
}
if (j < n)
   printf("yes")
else
  printf ("no");
```

Choose the correct expression for E.

Aa[j] - a[i] > S
Ba[j] - a[i]
Ca[i] - a[j]
Da[i] - a[j] > S
Arrays **Gate IT 2005**
Discuss it
Question 24 Explanation:
Please see the link below for full explanation http://www.geeksforgeeks.org/find-a-pair-with-the-given-difference/
Question 25
Let a and b be two sorted arrays containing n integers each, in non-decreasing order. Let c be a sorted array containing 2n integers obtained by merging the two arrays a and b. Assuming the arrays are indexed starting from 0, consider the following four statements

1. a[i] ≥ b [i] => c[2i] ≥ a [i]
2. a[i] ≥ b [i] => c[2i] ≥ b [i]
3. a[i] ≥ b [i] => c[2i] ≤ a [i]
4. a[i] ≥ b [i] => c[2i] ≤ b [i]

Which of the following is TRUE?
Aonly I and II
Bonly I and IV
Conly II and III
Donly III and IV
Arrays **Gate IT 2005**
Discuss it
Question 26
The following function computes the maximum value contained in an integer array p[] of size n (n >= 1)

```
int max(int *p, int n)
{
   int a=0, b=n-1;
   while (___________)
   {
      if (p[a] <= p[b])
      {
         a = a+1;
      }
      else
      {
         b = b-1;
      }
   }
   return p[a];
}
```

The missing loop condition is
Aa != n

Bb != 0
Cb > (a + 1)
Db != a
Arrays GATE-CS-2016 (Set 1)
Discuss it
Question 26 Explanation:

```
#include<iostream>
int max(int *p, int n)
{
   int a=0, b=n-1;
   while (a!=b)
   {
      if (p[a] <= p[b])
      {
         a = a+1;
      }
      else
      {
         b = b-1;
      }
   }
   return p[a];
}

int main()
{
  int arr[] = {10, 5, 1, 40, 30};
  int n = sizeof(arr)/sizeof(arr[0]);
  std::cout << max(arr, 5);
}
```

Question 27
Consider the C program given below :

```
 #include <stdio.h>
int main ()   {
   int sum = 0, maxsum = 0,  i,  n = 6;
   int a [] = {2, -2, -1, 3, 4, 2};
   for (i = 0; i < n; i++)   {
         if (i == 0 || a [i]  < 0  || a [i] < a [i - 1])  {
                 if (sum > maxsum) maxsum = sum;
                 sum = (a [i] > 0) ? a [i] : 0;
         }
         else sum += a [i];
   }
   if (sum > maxsum) maxsum = sum ;
   printf ("%d\n", maxsum);

}
```

What is the value printed out when this program is executed?
A9
B8
C7
D6
Arrays C Quiz - 113 Gate IT 2007
Discuss it
Question 27 Explanation:
If you look for loop carefully, you will notice that it assigns sum variable to some value in if condition and increments it in the else condition. On further thought, it would be clear that this loop stores sum of increasing subsequence of positive integers in sum variable and max of sum in maxsum. Hence, maxsum - maximum sum of increasing subsequence of positive integers will get printed out when this program is executed, which is 3 + 4 = 7. This solution is contributed by **Vineet Purswani** //output will be 3+4 =7 {for || if 1st argument is true 2nd argument will not be calculated, and if 1st argument is false, 2nd argument will be calculated} **Another Solution When i=1** -> i==0 is false, but a[i] maxsum) is true, since sum=2 and maxsum=0.So maxsum=2. sum = (a [i] > 0) ? a [i] : 0; , sum=0 since a[i]**When i=2** -> i==0 is false, a[i] maxsum) is false, since sum=0 and maxsum=2.Since sum = (a [i] > 0) ? a [i] : 0; , sum=0 since a[i] **When i=3** -> i==0 is false , a[i]When i=4 -> i==0 is false , a[i]When i=5 -> i==0 is false , a[i] maxsum is true, since sum=7 and maxsum=2,so maxsum=7.Since sum = (a [i] > 0) ? a [i] : 0, so sum=2 since a[5]>0. This solution is contributed by **nirmal Bharadwaj**
Question 28

What is the output printed by the following C code?

```
# include <stdio.h>
int main ()
{
   char a [6] = "world";
   int i, j;
   for (i = 0, j = 5; i < j; a [i++] = a [j--]);
   printf ("%s\n", a);
}
 /* Add code here. Remove these lines if not writing code */
```

Adlrow
BNull String
Cdlrld
Dworow
Arrays   C Quiz - 113   Gate IT 2008
Discuss it
Question 28 Explanation:
As at the base address or starting of the string "Null" is placed, so while reading array if Null comes it assumes that this is the end of array, so it terminates here only.
Question 29
Consider the C program given below. What does it print?

```

#include <stdio.h>
int main ()
{
      int i, j;
      int a [8] = {1, 2, 3, 4, 5, 6, 7, 8};
      for(i = 0; i < 3; i++) {
          a[i] = a[i] + 1;
          i++;
      }
      i--;
      for (j = 7; j > 4; j--) {
          int i = j/2;
          a[i] = a[i] - 1;
      }
      printf ("%d, %d", i, a[i]);
}
 /* Add code here. Remove these lines if not writing code */
```

A2, 3
B2, 4
C3, 2
D3, 3
Arrays   C Quiz - 113   Gate IT 2008
Discuss it
Question 29 Explanation:
Be careful about the scope of i,
there are two variables named: i, with different scope. There are 2 main points to consider while solving this question. Scope of variable i and integer division. First for loop will run for i = 0, 2 and 4 as i is incremented twice inside loop and resultant array will be a = 2, 2, 4, 4, 5, 6, 7, 8 (Loop will terminate at i = 4) After that i value is 3 as there is a decrement operation after for loop. Next for loop is running for j = 7, 6 and 5 and corresponding i values which is a local variable inside for loop will be 3 (7/2), 3 (6/2) and 2 (5/2). Array after this for loop will be a = 2, 2, 3, 2, 5, 6, 7, 8 After the for loop, current i value is 3 and element at a[3] = 2. This solution is contributed by **Pranjul Ahuja.**
Question 30
C program is given below:

```
# include <stdio.h>
int main ()
{
      int i, j;
      char a [2] [3] = {{'a', 'b', 'c'}, {'d', 'e', 'f'}};
      char b [3] [2];
      char *p = *b;
      for (i = 0; i < 2; i++) {
          for (j = 0; j < 3; j++) {
          *(p + 2*j + i) = a [i] [j];
          }
      }
}
```

```
/* Add code here. Remove these lines if not writing code */
```

What should be the contents of the array b at the end of the program?
Aa b
  c d
  e f
Ba d
  b e
  c f
Ca c
  e b
  d f
Da e
  d c
  b f
Arrays **C Quiz - 113** **Gate IT 2008**
Discuss it
Question 30 Explanation:
*p= a[0][0]
*(p+2) = a[0][1] *(p+4) = a[0][2] *(p+1) = a[1][0] *(p+3) = a[1][1] *(p+5) = a[1][2]
Question 31
Consider the array A[]= {6,4,8,1,3} apply the insertion sort to sort the array . Consider the cost associated with each sort is 25 rupees , what is the total cost of the insertion sort when element 1 reaches the first position of the array ?
A50
B25
C75
D100
Sorting **Arrays** **InsertionSort**
Discuss it
Question 31 Explanation:
When the element 1 reaches the first position of the array two comparisons are only required hence 25 * 2= 50 rupees. *step 1: 4 6 8 1 3 . *step 2: 1 4 6 8 3.
Question 32
You are given an array A[] having n random bits and a function OR(i,j) which will take two indexes from an array as parameters and will return the result of (A[i] OR A[j]) i.e bitwise OR. What is the minimum no of OR calls required so as to determine all the bits inside the array i.e. to determine every index of A[] whether it has 0 or 1 ?
AN-1
BN*(N-1)/2
CN
DNot possible to determine the bit array
Arrays **GATE 2017 Mock**
Discuss it
Question 32 Explanation:
Answer will be N as we can send the same index into that function OR(i,i) and get to know the bit.
If OR(i,i) = 0 that means a[i] = 0 else a[i] = 1.
There are 32 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Load Comments

Question 1
Consider the following code. The function *myStrcat* concatenates two strings. It appends all characters of b to end of a. So the expected output is "Geeks Quiz". The program compiles fine but produces segmentation fault when run.

```
#include <stdio.h>

void myStrcat(char *a, char *b)
{
    int m = strlen(a);
    int n = strlen(b);
    int i;
    for (i = 0; i <= n; i++)
       a[m+i]  = b[i];
}

int main()
{
    char *str1 = "Geeks ";
    char *str2 = "Quiz";
    myStrcat(str1, str2);
    printf("%s ", str1);
    return 0;
}
```

Which of the following changes can correct the program so that it prints "Geeks Quiz"?
Achar *str1 = "Geeks "; can be changed to char str1[100] = "Geeks ";
Bchar *str1 = "Geeks "; can be changed to char str1[100] = "Geeks "; and a line a[m+n-1] = '\0' is added at the end of myStrcat
CA line a[m+n-1] = '\0' is added at the end of myStrcat
DA line 'a = (char *)malloc(sizeof(char)*(strlen(a) + strlen(b) + 1)) is added at the beginning of myStrcat()
String
Discuss it
Question 1 Explanation:
See following for explanation. http://www.geeksforgeeks.org/storage-for-strings-in-c/
Question 2
What is the output of following program?

```
# include <stdio.h>

int main()
{
   char str1[] = "GeeksQuiz";
   char str2[] = {'G', 'e', 'e', 'k', 's', 'Q', 'u', 'i', 'z'};
   int n1 = sizeof(str1)/sizeof(str1[0]);
   int n2 = sizeof(str2)/sizeof(str2[0]);
   printf("n1 = %d, n2 = %d", n1, n2);
   return 0;
}
```

An1 = 10, n2 = 9
Bn1 = 10, n2 = 10
Cn1 = 9, n2 = 9
Dn1 = 9, n2 = 10
String
Discuss it
Question 2 Explanation:
The size of str1 is 10 and size of str2 9. When an array is initialized with string in double quotes, compiler adds a '\0' at the end.
Question 3
What is the output of following program?

```
#include<stdio.h>
void swap(char *str1, char *str2)
{
  char *temp = str1;
  str1 = str2;
  str2 = temp;
}

int main()
{
  char *str1 = "Geeks";
  char *str2 = "Quiz";
  swap(str1, str2);
  printf("str1 is %s, str2 is %s", str1, str2);
  return 0;
}
```

Astr1 is Quiz, str2 is Geeks
Bstr1 is Geeks, str2 is Quiz
Cstr1 is Geeks, str2 is Geeks
Dstr1 is Quiz, str2 is Quiz
String

Discuss it
Question 3 Explanation:
The above swap() function doesn't swap strings. The function just changes local pointer variables and the changes are not reflected outside the function. See following for more details. http://www.geeksforgeeks.org/swap-strings-in-c/
Question 4
Predict the output?

```
#include <stdio.h>
int fun(char *str1)
{
  char *str2 = str1;
  while(*++str1);
  return (str1-str2);
}

int main()
{
  char *str = "GeeksQuiz";
  printf("%d", fun(str));
  return 0;
}
```

A10
B9
C8
DRandom Number
String
Discuss it
Question 4 Explanation:
The function fun() basically counts number of characters in input string. Inside fun(), pointer str2 is initialized as str1. The statement while(*++str1); increments str1 till '\0' is reached. str1 is incremented by 9. Finally the difference between str2 and str1 is returned which is 9.
Question 5
What does the following fragment of C-program print?

```
 char c[] = "GATE2011";
 char *p =c;
 printf("%s", p + p[3] - p[1]) ;
```

AGATE2011
BE2011
C2011
D011
String
Discuss it
Question 5 Explanation:
See comments for explanation.

```

char c[] = "GATE2011";

 // p now has the base address string "GATE2011"
char *p = c;

// p[3] is 'E' and p[1] is 'A'.
// p[3] - p[1] = ASCII value of 'E' - ASCII value of 'A' = 4
// So the expression  p + p[3] - p[1] becomes p + 4 which is
// base address of string "2011"
printf("%s", p + p[3] - p[1]);  // prints 2011
```

Question 6

```
#include<stdio.h>
int main()
{
    char str[] = "GeeksQuiz";
    printf("%s %s %s\n", &str[5], &5[str], str+5);
    printf("%c %c %c\n", *(str+6), str[6], 6[str]);
    return 0;
}
```

ARuntime Error

BCompiler Error
Cuiz uiz uiz u u u
DQuiz Quiz Quiz u u u
String
Discuss it
Question 6 Explanation:
The program has no error. All of the following expressions mean same thing &str[5] &5[str] str+5 Since compiler converts the array operation in pointers before accessing the array elements, all above result in same address. Similarly, all of the following expressions mean same thing. *(str+6) str[6] 6[str]
Question 7
In below program, what would you put in place of “?” to print “Quiz”?

```
#include <stdio.h>
int main()
{
  char arr[] = "GeeksQuiz";
  printf("%s", ?);
  return 0;
}
```

Aarr
B(arr+5)
C(arr+4)
DNot possible
String
Discuss it
Question 7 Explanation:
Since %s is used, the printf statement will print everything starting from arr+5 until it finds ‘\0’
Question 8
Output?

```
int main()
{
    char a[2][3][3] = {'g','e','e','k','s','q','u','i','z'};
    printf("%s ", **a);
    return 0;
}
```

ACompiler Error
Bgeeksquiz followed by garbage characters
Cgeeksquiz
DRuntime Error
String
Discuss it
Question 8 Explanation:
We have created a 3D array that should have 2*3*3 (= 18) elements, but we are initializing only 9 of them. In C, when we initialize less no of elements in an array all uninitialized elements become ‘\0′ in case of char and 0 in case of integers.
Question 9
Consider the following C program segment:

```
 char p[20];
 char *s = "string";
 int length = strlen(s);
 int i;
 for (i = 0; i < length; i++)
     p[i] = s[length — i];
 printf("%s", p);
```

The output of the program is? (GATE CS 2004)
Agnirts
Bgnirt
Cstring
Dno output is printed
String
Discuss it
Question 9 Explanation:
Let us consider below line inside the for loop p[i] = s[length — i]; For i = 0, p[i] will be s[6 — 0] and s[6] is ‘\0′ So p[0] becomes ‘\0’. It doesn't matter what comes in p[1], p[2]….. as P[0] will not change for i >0. Nothing is printed if we print a string with first character ‘\0′
Question 10

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

```
void my_toUpper(char* str, int index)
{
    *(str + index) &= ~32;
}

int main()
{
    char* arr = "geeksquiz";
    my_toUpper(arr, 0);
    my_toUpper(arr, 5);
    printf("%s", arr);
    return 0;
}
```

AGeeksQuiz
Bgeeksquiz
CCompiler dependent
String
Discuss it
Question 10 Explanation:
The memory for the string **arr** is allocated in the read/write only area of data section. The choice is compiler dependent. In the newer version of compilers, the memory is allocated in the read only section of the data area. So any modification in the string is not possible. In older version compilers like Turbo-C, modification is possible.
Question 11
Predict the output of the following program:

```
#include <stdio.h>
int main()
{
    char str[] = "%d %c", arr[] = "GeeksQuiz";
    printf(str, 0[arr], 2[arr + 3]);
    return 0;
}
```

AG Q
B71 81
C71 Q
DCompile-time error
String
Discuss it
Question 11 Explanation:

The statement *printf(str, 0[arr], 2[arr + 3]);* boils down to:
printf("%d %c, 0["GeeksQUiz"], 2["GeeksQUiz" + 3]);
Which is further interpreted as:
*printf("%d %c, *(0 + "GeeksQUiz"), *(2 + "GeeksQUiz" + 3));*
Which prints the ascii value of *'G'* and character *'Q'*.

Question 12
Output of following program

```
#include <stdio.h>
int fun(char *p)
{
    if (p == NULL || *p == '\0') return 0;
    int current = 1, i = 1;
    while (*(p+current))
    {
        if (p[current] != p[current-1])
        {
            p[i] = p[current];
            i++;
        }
        current++;
    }
    *(p+i)='\0';
    return i;
}

int main()
{
    char str[] = "geeksskeeg";
    fun(str);
    puts(str);
    return 0;
}
```

Agekskeg
Bgeeksskeeg
Cgeeks
DGarbage Values
String
Discuss it
Question 12 Explanation:
The function mainly replaces more than once consecutive occurrences of a character with one one occurrence.
Question 13

```
int main()
{
    char p[] = "geeksquiz";
    char t;
    int i, j;
    for(i=0,j=strlen(p); i<j; i++)
    {
        t = p[i];
        p[i] = p[j-i];
        p[j-i] = t;
    }
    printf("%s", p);
    return 0;
}
```

Output?
Aziuqskeeg
BNothing is printed on the screen
Cgeeksquiz
Dgggggggg
String
Discuss it
Question 13 Explanation:
The string termination character '\0' is assigned to first element of array p[]
Question 14
Assume that a character takes 1 byte. Output of following program?

```
#include<stdio.h>
int main()
{
    char str[20] = "GeeksQuiz";
    printf ("%d", sizeof(str));
    return 0;
}
```

A9
B10
C20
DGarbage Value
String
Discuss it
Question 14 Explanation:
Note that the sizeof() operator would return size of array. To get size of string stored in array, we need to use strlen(). The following program prints 9.

```
#include <stdio.h>
#include <string.h>
int main()
{
    char str[20] = "GeeksQuiz";
    printf ("%d", strlen(str));
    return 0;
}
```

Question 15

Predict the output of following program, assume that a character takes 1 byte and pointer takes 4 bytes.

```
#include <stdio.h>
int main()
{
   char *str1 = "GeeksQuiz";
   char str2[] = "GeeksQuiz";

   printf("sizeof(str1) = %d, sizeof(str2) = %d",
         sizeof(str1), sizeof(str2));

   return 0;
}
```

Asizeof(str1) = 10, sizeof(str2) = 10
Bsizeof(str1) = 4, sizeof(str2) = 10
Csizeof(str1) = 4, sizeof(str2) = 4
Dsizeof(str1) = 10, sizeof(str2) = 4

String

Discuss it

Question 15 Explanation:
str1 is a pointer and str2 is an array.

Question 16

The output of following C program is

```
#include <stdio.h>
char str1[100];

char *fun(char str[])
{
   static int i = 0;
   if (*str)
   {
      fun(str+1);
      str1[i] = *str;
      i++;
   }
   return str1;
}

int main()
{
   char str[] = "GATE CS 2015 Mock Test";
   printf("%s", fun(str));
   return 0;
}
```

AGATE CS 2015 Mock Test
BtseT kcoM 5102 SC ETAG
CNothing is printed on screen
DSegmentation Fault

String   **GATE-CS-2015 (Mock Test)**

Discuss it

Question 16 Explanation:
The function basically reverses the given string.

Question 17

Consider the following function written in the C programming language. The output of the above function on input “ABCD EFGH” is

```
void foo (char *a)
{
  if (*a && *a != ` `)
  {
    foo(a+1);
    putchar(*a);
  }
}
```

AABCD EFGH
BABCD
CHGFE DCBA
DDCBA

String **GATE-CS-2015 (Set 2)**
Discuss it
Question 17 Explanation:
The program prints all characters before ' ' or '\0' (whichever comes first) in reverse order. Let's assume that Base address of given Array is 1000. It can be represented as given below in the image
pranjul_25
Below is the complete Recursion Stack
pranjul_25_2
of the Given Code After calling foo(1004), recursion will return as character at 1004 is ‘ ’. foo(1004) will return to its caller which is foo(1003) and next line in foo(1003) will be executed which will print character at 1003 (‘D’). foo(1003) will then return to foo(1002) and character at 1002 (‘C’) will get printed and the process will continue till foo(1000). This solution is contributed by **Pranjul Ahuja**.
Question 18
Consider the following C program segment.

```
# include <stdio.h>
int main( )
{
   char s1[7] = "1234", *p;
   p = s1 + 2;
   *p = '0' ;
   printf ("%s", s1);
}
```

What will be printed by the program?
A12
B120400
C1204
D1034
String **GATE-CS-2015 (Set 3)**
Discuss it
Question 18 Explanation:

```
   char s1[7] = "1234", *p;
   p = s1 + 2;   // p holds address of character 3
   *p = '0' ;  // memory at s1 + 3 now becomes 0
   printf ("%s", s1); // All characters are printed
```

Question 19
Output of following C program? Assume that all necessary header files are included

```
int main()
{
   char *s1 = (char *)malloc(50);
   char *s2 = (char *)malloc(50);
   strcpy(s1, "Geeks");
```

```
    strcpy(s2, "Quiz");
    strcat(s1, s2);
    printf("%s", s1);
    return 0;
}
```

AGeeksQuiz
BGeeks
CGeeks Quiz
DQuiz
String
Discuss it
Question 19 Explanation:
strcpy puts \0 at the end. strcat starts from \0, concatenates string and puts \0 at the end.
There are 19 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Question 1
Output of following program?

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

A7 6 5
B5 6 7
C7 7 7
DCompiler Dependent
Functions
Discuss it
Question 1 Explanation:
When parameters are passed to a function, the value of every parameter is evaluated before being passed to the function. What is the order of evaluation of parameters - left-to-right or right-to-left? If evaluation order is left-to-right, then output should be 5 6 7 and if the evaluation order is right-to-left, then output should be 7 6 5. Unfortunately, there is no fixed order defined by C standard. A compiler may choose to evaluate either from left-to-right. So the output is compiler dependent.
Question 2
In C, parameters are always
APassed by value
BPassed by reference
CNon-pointer variables are passed by value and pointers are passed by reference
DPassed by value result
Functions
Discuss it
Question 2 Explanation:
In C, function parameters are always passed by value. Pass-by-reference is simulated in C by explicitly passing pointer values.
Question 3
Which of the following is true about return type of functions in C?
AFunctions can return any type
BFunctions can return any type except array and functions
CFunctions can return any type except array, functions and union

DFunctions can return any type except array, functions, function pointer and union
Functions
Discuss it
Question 3 Explanation:
In C, functions can return any type except arrays and functions. We can get around this limitation by returning pointer to array or pointer to function.
Question 4

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

AAddress of main function
BCompiler Error
CRuntime Error
DSome random value
Functions
Discuss it
Question 4 Explanation:
Explanation: Name of the function is actually a pointer variable to the function and prints the address of the function. Symbol table is implemented like this.

```
struct
{
   char *name;
   int (*funcptr)();
}
symtab[] = {
   "func", func,
   "anotherfunc", anotherfunc,
};
```

Question 5
Output?

```
#include <stdio.h>

int main()
{
    int (*ptr)(int ) = fun;
    (*ptr)(3);
    return 0;
}

int fun(int n)
{
  for(;n > 0; n--)
    printf("GeeksQuiz ");
  return 0;
}
```

AGeeksQuiz GeeksQuiz GeeksQuiz
BGeeksQuiz GeeksQuiz
CCompiler Error
DRuntime Error
Functions
Discuss it
Question 5 Explanation:
The only problem with program is ***fun*** is not declared/defined before it is assigned to ptr. The following program works fine and prints "GeeksQuiz GeeksQuiz GeeksQuiz "

```
int fun(int n);

int main()
{
    // ptr is a pointer to function fun()
    int (*ptr)(int ) = fun;

    // fun() called using pointer
    (*ptr)(3);
    return 0;
}

int fun(int n)
{
  for(;n > 0; n--)
```

```
    printf("GeeksQuiz ");
}
```

Question 6
Output of following program?

```
#include<stdio.h>

void dynamic(int s, ...)
{
    printf("%d ", s);
}

int main()
{
    dynamic(2, 4, 6, 8);
    dynamic(3, 6, 9);
    return 0;
}
```

A2 3
BCompiler Error
C4 3
D3 2
Functions
Discuss it
Question 6 Explanation:
In c three continuous dots is known as ellipsis which is variable number of arguments of function. The values to parameters are assigned one by one. Now the question is how to access other arguments. See this for details.
Question 7
Predict the output?

```
#include <stdio.h>
int main()
{
    void demo();
    void (*fun)();
    fun = demo;
    (*fun)();
    fun();
    return 0;
}

void demo()
{
    printf("GeeksQuiz ");
}
```

AGeeksQuiz
BGeeksQuiz GeeksQuiz
CCompiler Error
DBlank Screen
Functions
Discuss it
Question 7 Explanation:
This is a simple program with function pointers. fun is assigned to point to demo. So the two statements "(*fun)();" and "fun();" mean the same thing.
Question 8
What is the meaning of using extern before function declaration? For example following function sum is made extern

```
extern int sum(int x, int y, int z)
{
    return (x + y + z);
}
```

AFunction is made globally available
Bextern means nothing, sum() is same without extern keyword.
CFunction need not to be declared before its use
DFunction is made local to the file.
Functions
Discuss it

Question 8 Explanation:
extern keyword is used for global variables. Functions are global anyways, so adding extern doesn't add anything.
Question 9
What is the meaning of using static before function declaration? For example following function sum is made static

```
static int sum(int x, int y, int z)
{
   return (x + y + z);
}
```

AStatic means nothing, sum() is same without static keyword.
BFunction need not to be declared before its use
CAccess to static functions is restricted to the file where they are declared
DStatic functions are made inline
Functions
Discuss it
Question 9 Explanation:
In C, functions are global by default. Unlike global functions, access to static functions is restricted to the file where they are declared. We can have file level encapsulation using static variables/functions in C because when we make a global variable static, access to the variable becomes limited to the file in which it is declared.
Question 10
In C, what is the meaning of following function prototype with empty parameter list

```
void fun()
{
  /* .... */
}
```

AFunction can only be called without any parameter
BFunction can be called with any number of parameters of any types
CFunction can be called with any number of integer parameters.
DFunction can be called with one integer parameter.
Functions
Discuss it
Question 10 Explanation:
Empty list in C mean that the parameter list is not specified and function can be called with any parameters. In C, to declare a function that can only be called without any parameter, we should use "void fun(void)" As a side note, in C++, empty list means function can only be called without any parameter. In C++, both void fun() and void fun(void) are same.
Question 11

```
#include <stdio.h>
#include <stdarg.h>
int fun(int n, ...)
{
   int i, j = 1, val = 0;
   va_list p;
   va_start(p, n);
   for (; j < n; ++j)
   {
      i = va_arg(p, int);
      val += i;
   }
   va_end(p);
   return val;
}
int main()
{
   printf("%d\n", fun(4, 1, 2, 3));
   return 0;
}
```

A3
B5
C6
D10
Functions
Discuss it
Question 11 Explanation:
The function receives variable number of arguments as there are three dots after first argument.   The firs argument is count of all arguments including first. The function mainly returns sum of all remaining arguments. See http://www.geeksforgeeks.org/how-to-count-variable-numbers-of-arguments-in-c for details.

Question 12
Consider the following C-program:

```
void foo(int n, int sum)
{
  int k = 0, j = 0;
  if (n == 0) return;
    k = n % 10;
  j = n / 10;
  sum = sum + k;
  foo (j, sum);
  printf ("%d,", k);
}

int main ()
{
  int a = 2048, sum = 0;
  foo (a, sum);
  printf ("%d\n", sum);

  getchar();
}
```

What does the above program print?
A8, 4, 0, 2, 14
B8, 4, 0, 2, 0
C2, 0, 4, 8, 14
D2, 0, 4, 8, 0
Functions GATE-CS-2005
Discuss it
Question 12 Explanation:
See Question 5 of http://www.geeksforgeeks.org/c-language-set-3/
Question 13
Consider the following C-program:

```
double foo (double); /* Line 1 */

int main()
{

    double da, db;

    // input da

    db = foo(da);

}

double foo(double a)
{
    return a;
}
```

The above code compiled without any error or warning. If Line 1 is deleted, the above code will show:
Ano compile warning or error
Bsome compiler-warnings not leading to unintended results
Csome compiler-warnings due to type-mismatch eventually leading to unintended results
Dcompiler errors
Functions GATE-CS-2005
Discuss it
Question 13 Explanation:
Refer What happens when a function is called before its declaration in C?
Question 14
Consider the following C function

```
void swap (int a, int b)
{
   int temp;
   temp = a;
   a = b;
   b = temp;
}
```

In order to exchange the values of two variables x and y.
ACall swap (x, y)
BCall swap (&x, &y)
Cswap(x,y) cannot be used as it does not return any value
Dswap(x,y) cannot be used as the parameters are passed by value
Functions   GATE-CS-2004
Discuss it
Question 14 Explanation:
See question 3 of http://www.geeksforgeeks.org/c-language-set-2/
Question 15
The value of j at the end of the execution of the following C program.

```
int incr(int i)
{
   static int count = 0;
   count = count + i;
   return (count);
}
main()
{
   int i,j;
   for (i = 0; i <=4; i++)
      j = incr(i);
}
```

A10
B4
C6
D7
Functions   GATE-CS-2000
Discuss it
Question 15 Explanation:
See question 2 of http://www.geeksforgeeks.org/c-language-set-1/
Question 16
The output of the following C program is __________.

```
void f1 (int a, int b)
{
  int c;
  c=a; a=b; b=c;
}
void f2 (int *a, int *b)
{
  int c;
  c=*a; *a=*b;*b=c;
}
int main()
{
  int a=4, b=5, c=6;
  f1(a, b);
  f2(&b, &c);
  printf (“%d”, c-a-b);
  return 0;
}
```

A-5
B-4
C5
D3
Functions   GATE-CS-2015 (Set 1)
Discuss it
Question 16 Explanation:
The function call to to f1(a, b) won't have any effect as the values are passed by value. The function call f2(&b, &c) swaps values of b and c. So b becomes 6 and c becomes 5. Value of c-a-b becomes 5-4-6 which is -5.
Question 17
What’s going to happen when we compile and run the following C program snippet?

```
#include "stdio.h"
int main()
{
 int a = 10;
 int b = 15;

 printf("=%d",(a+1),(b=a+2));
 printf(" %d=",b);

 return 0;
}
```

A=11 15=
B=11 12=
CCompiler Error due to (b=a+2) in the first printf().
DNo compile error but output would be =11 X= where X would depend on compiler implementation.
Functions   **C Quiz - 103**
Discuss it
Question 17 Explanation:
As per C standard C11, all the arguments of printf() are evaluated irrespective of whether they get printed or not. That's why (b=a+2) would also be evaluated and value of b would be 12 after first printf(). That's why correct answer is B.
Question 18
What's going to happen when we compile and run the following C program snippet?

```
#include "stdio.h"
int main()
{
 int a = 10;

 printf("=%d %d=",(a+1));

 return 0;
}
```

A=11 0=
B=11 X= where X would depend on Compiler implementation
CUndefined behaviour
DCompiler Error due to missing argument for second %d
Functions   **C Quiz - 103**
Discuss it
Question 18 Explanation:
In the context of printf() and fprintf(), as per C standard C11 clause 7.21.6.1, "If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored." Some implementation can choose to print =10 0= while other implementations can choose to print =11 X=. That's why the output of above program varies depending on the compiler and platform. Hence, correct answer is C).
Question 19
Assuming int size is 4 bytes, what is going to happen when we compile and run the following program?

```
#include “stdio.h”
int main()
{
  printf(“GeeksQuiz\n”);
  main();
  return 0;
}
```

AWe can't use main() inside main() and compiler will catch it by showing compiler error.
BGeeksQuiz would be printed in 2147483647 times i.e. (2 to the power 31) - 1.
CIt'll print GeeksQuiz infinite times i.e. the program will continue to run forever until it's terminated by other means such as CTRL+C or CTRL+Z etc.

DGeeksQuiz would be printed only once. Because when main() is used inside main(), it's ignored by compiler at run time. This is to make sure that main() is called only once.
EGeeksQuiz would be printed until stack overflow happens for this program.
Functions   **C Quiz - 106**
Discuss it
Question 19 Explanation:
First of all, there's no restriction of main() calling main() i.e. recursion can happen for main() as well. But there's no explicit termination condition mentioned here for this recursion. So main() would be calling main() after printing GeeksQuiz. This will go on until Stack of the program would be filled completely. Please note that stack (internal to a running program) stores the calling function sequence i.e. which function has called which function so that the control can be returned when called function returns. That's why here in program, main() would continue to call main() until complete stack is over i.e. stack-overflow occurs.
Question 20
Both of the following declarations for function pointers are equivalent. Second one (i.e. with typedef) looks cleaner.

```
/* First Declaration */
int (*funPtr1)(int), (*funPtr2)(int);

/* Second Declaration*/
typedef int (*funPtr)(int);
funPtr funPtr1, funPtr2;
```

ATRUE
BFALSE
Functions   C Quiz - 108
Discuss it
Question 20 Explanation:
Usually data type of function pointers tends to be cryptic and that's why it's used in conjunction with *typedef*. Think of a function pointer which is pointing to a function that accepts a function pointer and that returns a function pointer. This can be used simplified using typedef otherwise it's going to very difficult to read/understand!
Question 21
Pick the best statement for the following program.

```
#include "stdio.h"

int foo(int a)
{
 printf("%d",a);
 return 0;
}

int main()
{
 foo;
 return 0;
}
```

AIt'll result in compile error because foo is used without parentheses.
BNo compile error and some garbage value would be passed to foo function. This would make foo to be executed with output "garbage integer".
CNo compile error but foo function wouldn't be executed. The program wouldn't print anything.
DNo compile error and ZERO (i.e. 0) would be passed to foo function. This would make foo to be executed with output 0.
Functions   C Quiz - 109
Discuss it
Question 21 Explanation:
In C, if a function name is used without parentheses, the reference to the function name simply generates a pointer to the function, which is then discarded. So the above program would compile but won't print anything.
Question 22
What is the output of the following program?

```
#include <stdio.h>
int funcf (int x);
int funcg (int y);

main()
{
   int x = 5, y = 10, count;
   for (count = 1; count <= 2; ++count)
   {
      y += funcf(x) + funcg(x);
      printf ("%d ", y);
   }
}

funcf(int x)
{
   int y;
   y = funcg(x);
   return (y);
}

funcg(int x)
{
   static int y = 10;
   y += 1;
   return (y+x);
}
```

A43 80
B42 74
C33 37
D32 32
Functions GATE-IT-2004
Discuss it
Question 22 Explanation:
The count=1 and it goes till two,so following statement will be executed twice.

```
y += funcf(x) + funcg(x);
```

1st call- funcg(x); // y = 11 y+x= 16. 2nd call funcg(x); // y= 12 y+x= 17. First iteration-> main()->y = 16+17 **+10 =** **43** Second iteration-> main() y= 18+19 **+43 =80** **So the Answer is A** See more about static variables at: http://www.geeksforgeeks.org/g-fact-80/
Question 23
What is the output printed by the following program?

```
#include<stdio.h>
int f(int n, int k)
{
   if (n == 0)
      return 0;
   else if (n % 2)
      return f(n/2, 2*k) + k;
   else return f(n/2, 2*k) - k;
}
int main ()
{
   printf("%d", f(20, 1));
   return 0;
}
```

A5
B8
C9
D20
Functions Gate IT 2005
Discuss it
Question 23 Explanation:

```
f(20,1) = 9.
f(10,2) - 1 = 9
f(5,4) - 2 = 10
f(2,8) + 4 = 12
f(1,16) - 8 = 8
f(0,32) + 16 = 16
```

Question 24
Consider the following C program.

```
void f(int, short);
void main()
{
  int i = 100;
  short s = 12;
  short *p = &s;
  __________ ;  // call to f()
}
```

Which one of the following expressions, when placed in the blank above, will NOT result in a type checking error?
Af(s, *s)
Bi = f(i,s)
Cf(i,*s)
Df(i,*p)
Functions GATE-CS-2016 (Set 1)
Discuss it
Question 24 Explanation:
i is integer and *p is value of a pointer to short. 1) Option 1 is wrong because we are passing "*S" as second argument check that S is not

a pointer variable .So error 2) Second option is we are trying to store the value of f(i,s) into i but look at the function definition outside main it has no return type. It is simply void so that assignment is wrong. So error 3) Option 3 is wrong because of the same reason why option 1 is wrong 4) So option d is correct.

Question 25

Consider the following program:

```
int f(int *p, int n)
{
   if (n <= 1) return 0;
   else return max(f(p+1,n-1),p[0]-p[1]);
}
int main()
{
   int a[] = {3,5,2,6,4};
   printf("%d", f(a,5));
}
```

Note: max(x,y) returns the maximum of x and y. The value printed by this program is

A2

B3

C4

D5

Functions **GATE-CS-2016 (Set 2)**

Discuss it

Question 25 Explanation:

Look at the recursion stack of the given code in the below image. Assuming that the base address of array starts from 1000 and an integer

pranjul_16

takes 4 Bytes.

pranjul_16_1

After the last recursive call f(1016,1) returns, in the previous call we will have return max(0,2) and then return max(2,-4) and then return max(2,3) and then finally return max(3,-2) = 3. This solution is contributed by **Pranjul Ahuja**

Question 26

Given a boolean function f (x_1, x_2, ..., x_n), which of the following equations is NOT true

Af (x1, x2, ..., xn) = x1'f(x1, x2, ..., xn) + x1f(x1, x2, ..., xn)

Bf (x1, x2, ..., xn) = x2f(x1, x2, …, xn) + x2'f(x1, x2, …,xn)

Cf (x1, x2, ..., xn) = xn'f(x1, x2, …, 0) + xnf(x1, x2, …,1)

Df (x1, x2, ..., xn) = f(0, x2, …, xn) + f(1, x2, .., xn)

Functions **Set Theory & Algebra** **GATE IT 2006**

Discuss it

Question 26 Explanation:
Option A: f (x1, x2, …, xn) = x1'f(x1, x2, …, xn) + x1f(x1, x2, …, xn) **Case 1**: taking x1=0 RHS = 1.f(x1, x2, …, xn) + 0.f(x1, x2, …, xn) RHS =f(x1, x2, …, xn). **Case 2**: taking x1=1 RHS = 0.f(x1, x2, …, xn) + 1.f(x1, x2, …, xn) RHS =f(x1, x2, …, xn). In both cases RHS=LHS, so, (A) is true **Option B**: f (x1, x2, …, xn) = x2f(x1, x2, …, xn) + x2'f(x1, x2, …, xn) **Case 1**: taking x2=0 RHS= 0.f(x1, x2, …, xn) + 1.f(x1, x2…,xn) RHS =f(x1, x2, …, xn). **Case 2**: taking x2=1 RHS = 1.f(x1, x2, …, xn) + 0.f(x1, x2, …, xn) RHS =f(x1, x2, …, xn). In both cases RHS=LHS, so, (B) is true. **Option C**: f (x1, x2, …, xn) = xn'f(x1, x2, …, 0) + xnf(x1, x2, …,1) **Case 1**: taking xn=0 RHS= 1.f(x1, x2, …, 0) + 0.f(x1, x2, …, 1) RHS =f(x1, x2, …, 0) **Case 2**: taking xn=1 RHS = 0.f(x1, x2, …, 0) + 1.f(x1, x2, …, 1) RHS =f(x1, x2, …, 1)In both cases RHS=LHS, so, (C) is true. **Option D**: f (x1, x2, …, xn) = f(0, x2, …, xn) + f(1, x2, .., xn) Here, no way to equate LHS and RHS so **'NOT true'**. NO term depends on value of 'x1'.   This solution is contributed by **Sandeep pandey.**
There are 26 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Pointers store address of variables or a memory location.

```
// General syntax
datatype *var_name;

// An example pointer "ptr" that holds
// address of an integer variable or holds
// address of a memory whose value(s) can
// be accessed as integer values through "ptr"
int *ptr;
```

Read More…

Question 1
What is the output of following program?

```
# include <stdio.h>
void fun(int x)
{
    x = 30;
}

int main()
{
  int y = 20;
  fun(y);
  printf("%d", y);
  return 0;
}
```

A30
B20
CCompiler Error
DRuntime Error
Pointer Basics
Discuss it
Question 1 Explanation:
Parameters are always passed by value in C. Therefore, in the above code, value of y is not modified using the function fun(). So how do we modify the value of a local variable of a function inside another function. Pointer is the solution to such problems. Using pointers, we can modify a local variable of a function inside another function. See the next question. Note that everything is passed by value in C. We only get the effect of pass by reference using pointers.
Question 2
Output of following program?

```
# include <stdio.h>
void fun(int *ptr)
{
    *ptr = 30;
}
```

```
int main()
{
  int y = 20;
  fun(&y);
  printf("%d", y);

  return 0;
}
```

A20
B30
CCompiler Error
DRuntime Error
Pointer Basics
Discuss it
Question 2 Explanation:
The function fun() expects a pointer ptr to an integer (or an address of an integer). It modifies the value at the address ptr. The dereference operator * is used to access the value at an address. In the statement '*ptr = 30', value at address ptr is changed to 30. The address operator & is used to get the address of a variable of any data type. In the function call statement 'fun(&y)', address of y is passed so that y can be modified using its address.
Question 3
Output of following program?

```
#include <stdio.h>

int main()
{
    int *ptr;
    int x;

    ptr = &x;
    *ptr = 0;

    printf(" x = %d\n", x);
    printf(" *ptr = %d\n", *ptr);

    *ptr += 5;
    printf(" x  = %d\n", x);
    printf(" *ptr = %d\n", *ptr);

    (*ptr)++;
    printf(" x = %d\n", x);
    printf(" *ptr = %d\n", *ptr);

    return 0;
}
```

```
Ax = 0
  *ptr = 0
  x = 5
  *ptr = 5
  x = 6
  *ptr = 6
Bx = garbage value
  *ptr = 0
  x = garbage value
  *ptr = 5
  x = garbage value
  *ptr = 6
Cx = 0
  *ptr = 0
  x = 5
  *ptr = 5
  x = garbage value
  *ptr = garbage value

Dx = 0
  *ptr = 0
  x = 0
  *ptr = 0
  x = 0
```

  *ptr = 0

Pointer Basics

Discuss it

Question 3 Explanation:
See the comments below for explanation.

```
int *ptr;  /* Note: the use of * here is not for dereferencing,
           it is for data type int */
int x;

ptr = &x;   /* ptr now points to x (or ptr is equal to address of x) */
*ptr = 0;   /* set value ate ptr to 0 or set x to zero */

printf(" x = %d\n", x);   /* prints x =  0 */
printf(" *ptr = %d\n", *ptr);  /* prints *ptr =  0 */


*ptr += 5;        /* increment the value at ptr by 5 */
printf(" x  = %d\n", x);  /* prints x = 5 */
printf(" *ptr = %d\n", *ptr); /* prints *ptr =  5 */


(*ptr)++;         /* increment the value at ptr by 1 */
printf(" x  = %d\n", x);  /* prints x = 6 */
printf(" *ptr = %d\n", *ptr);  /* prints *ptr =  6 */
```

Question 4
Consider a compiler where int takes 4 bytes, char takes 1 byte and pointer takes 4 bytes.

```
#include <stdio.h>

int main()
{
    int arri[] = {1, 2 ,3};
    int *ptri = arri;

    char arrc[] = {1, 2 ,3};
    char *ptrc = arrc;

    printf("sizeof arri[] = %d ", sizeof(arri));
    printf("sizeof ptri = %d ", sizeof(ptri));

    printf("sizeof arrc[] = %d ", sizeof(arrc));
    printf("sizeof ptrc = %d ", sizeof(ptrc));

    return 0;
}
```

Asizeof arri[] = 3 sizeof ptri = 4 sizeof arrc[] = 3 sizeof ptrc = 4
Bsizeof arri[] = 12 sizeof ptri = 4 sizeof arrc[] = 3 sizeof ptrc = 1
Csizeof arri[] = 3 sizeof ptri = 4 sizeof arrc[] = 3 sizeof ptrc = 1
Dsizeof arri[] = 12 sizeof ptri = 4 sizeof arrc[] = 3 sizeof ptrc = 4

Pointer Basics

Discuss it

Question 4 Explanation:
Size of an array is number of elements multiplied by the type of element, that is why we get sizeof arri as 12 and sizeof arrc as 3. Size of a pointer is fixed for a compiler. All pointer types take same number of bytes for a compiler. That is why we get 4 for both ptri and ptrc.
Question 5
Assume that float takes 4 bytes, predict the output of following program.

```
#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;
}
```

A90.500000 3
B90.500000 12
C10.000000 12
D0.500000 3
Pointer Basics
Discuss it
Question 5 Explanation:
When we add a value x to a pointer p, the value of the resultant expression is p + x*sizeof(*p) where sizeof(*p) means size of data type pointed by p. That is why ptr2 is incremented to point to arr[3] in the above code. Same rule applies for subtraction. Note that only integral values can be added or subtracted from a pointer. We can also subtract or compare two pointers of same type.
Question 6

```
#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;
}
```

Assume that an int variable takes 4 bytes and a char variable takes 1 byte
ANumber of elements between two pointer are: 5. Number of bytes between two pointers are: 20
BNumber of elements between two pointer are: 20. Number of bytes between two pointers are: 20
CNumber of elements between two pointer are: 5. Number of bytes between two pointers are: 5
DCompiler Error
ERuntime Error

Pointer Basics
Discuss it
Question 6 Explanation:
Array name gives the address of first element in array. So when we do '*ptr1 = arr;', ptr1 starts holding the address of element 10. 'arr + 5' gives the address of 6th element as arithmetic is done using pointers. So 'ptr2-ptr1' gives 5. When we do '(char *)ptr2', ptr2 is type-casted to char pointer and size of character is one byte, pointer arithmetic happens considering character pointers. So we get 5*sizeof(int)/sizeof(char) as a difference of two pointers.
Question 7

```
#include<stdio.h>
int main()
{
   int a;
   char *x;
   x = (char *) &a;
   a = 512;
   x[0] = 1;
   x[1] = 2;
   printf("%d\n",a);
   return 0;
}
```

What is the output of above program?
AMachine dependent
B513
C258
DCompiler Error
Pointer Basics
Discuss it
Question 7 Explanation:
Output is 513 in a little endian machine. To understand this output, let integers be stored using 16 bits. In a little endian machine, when we do x[0] = 1 and x[1] = 2, the number a is changed to 00000001 00000010 which is representation of 513 in a little endian machine.
Question 8

```
int main()
{
 char *ptr = "GeeksQuiz";
 printf("%c\n", *&*&*ptr);
 return 0;
}
```

ACompiler Error
BGarbage Value
CRuntime Error
DG
Pointer Basics
Discuss it
Question 8 Explanation:
The operator * is used for dereferencing and the operator & is used to get the address. These operators cancel out effect of each other when used one after another. We can apply them alternatively any no. of times. In the above code, ptr is a pointer to first character of string g. *ptr gives us g, &*ptr gives address of g, *&*ptr again g, &*&*ptr address of g, and finally *&*&*ptr gives 'g' Now try below

```
int main()
{
 char *ptr = "GeeksQuiz";
 printf("%s\n", *&*&ptr);
 return 0;
}
```

Question 9

```
#include<stdio.h>
void fun(int arr[])
{
  int i;
  int arr_size = sizeof(arr)/sizeof(arr[0]);
  for (i = 0; i < arr_size; i++)
      printf("%d ", arr[i]);
}

int main()
{
  int i;
  int arr[4] = {10, 20 ,30, 40};
  fun(arr);
  return 0;
}
```

A10 20 30 40
BMachine Dependent
C10 20
DNorthing
Pointer Basics
Discuss it
Question 9 Explanation:
In C, array parameters are always treated as pointers. So following two statements have the same meaning.

```
void fun(int arr[])
void fun(int *arr)
```

[] is used to make it clear that the function expects an array, it doesn't change anything though. People use it only for readability so that the reader is clear about the intended parameter type. The bottom line is, sizeof should never be used for array parameters, a separate parameter for array size (or length) should be passed to fun(). So, **in the given program, arr_size contains ration of pointer size and integer size, this ration= is compiler dependent.** 1
There are 31 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Question 1
The most appropriate matching for the following pairs (GATE CS 2000)

```
X: m=malloc(5); m= NULL;        1: using dangling pointers
```

```
Y: free(n); n->value=5;        2: using uninitialized pointers
Z: char *p; *p = 'a';          3. lost memory is:
```

AX—1 Y—3 Z-2
B(X—2 Y—1 Z-3
CX—3 Y—2 Z-1
DX—3 Y—1 Z-2
Dynamic Memory Allocation
Discuss it
Question 1 Explanation:
X -> A pointer is assigned to NULL without freeing memory so a clear example of memory leak Y -> Trying to retrieve value after freeing it so dangling pointer. Z -> Using uninitialized pointers
Question 2
Consider the following three C functions :

```
[PI] int * g (void)
{
  int x= 10;
  return (&x);
}

[P2] int * g (void)
{
  int * px;
  *px= 10;
  return px;
}

[P3] int *g (void)
{
  int *px;
  px = (int *) malloc (sizeof(int));
  *px= 10;
  return px;
}
```

Which of the above three functions are likely to cause problems with pointers? (GATE 2001)
AOnly P3
BOnly P1 and P3
COnly P1 and P2
DP1, P2 and P3
Dynamic Memory Allocation
Discuss it
Question 2 Explanation:
In P1, pointer variable x is a local variable to g(), and g() returns pointer to this variable. x may vanish after g() has returned as x exists on stack. So, &x may become invalid. In P2, pointer variable px is being assigned a value without allocating memory to it. P3 works perfectly fine. Memory is allocated to pointer variable px using malloc(). So, px exists on heap, it's existence will remain in memory even after return of g() as it is on heap.
Question 3
Output?

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

void fun(int *a)
{
    a = (int*)malloc(sizeof(int));
}

int main()
{
    int *p;
    fun(p);
    *p = 6;
    printf("%d\n",*p);
    return(0);
}
```

AMay not work
BWorks and prints 6
Dynamic Memory Allocation

Discuss it

Question 3 Explanation:

The program is not valid. Try replacing "int *p;" with "int *p = NULL;" and it will try to dereference a null pointer. This is because fun() makes a copy of the pointer, so when malloc() is called, it is setting the copied pointer to the memory location, not p. p is pointing to random memory before and after the call to fun(), and when you dereference it, it will crash. If you want to add memory to a pointer from a function, you need to pass the address of the pointer (ie. double pointer).

Question 4

Which of the following is/are true

Acalloc() allocates the memory and also initializes the allocates memory to zero, while memory allocated using malloc() has random data.

Bmalloc() and memset() can be used to get the same effect as calloc().

Ccalloc() takes two arguments, but malloc takes only 1 argument.

DBoth malloc() and calloc() return 'void *' pointer.

EAll of the above

Dynamic Memory Allocation

Discuss it

Question 4 Explanation:

All of the given options are true. See http://www.geeksforgeeks.org/calloc-versus-malloc/ for details.

Question 5

What is the return type of malloc() or calloc()

Avoid *

BPointer of allocated memory type

Cvoid **

Dint *

Dynamic Memory Allocation

Discuss it

Question 5 Explanation:

malloc() and calloc() return void *. We may get warning in C if we don't type cast the return type to appropriate pointer.

Question 6

Which of the following is true?

A"ptr = calloc(m, n)" is equivalent to following
  ptr = malloc(m * n);

B"ptr = calloc(m, n)" is equivalent to following
  ptr = malloc(m * n); memset(ptr, 0, m * n);

C"ptr = calloc(m, n)" is equivalent to following
  ptr = malloc(m); memset(ptr, 0, m);

D"ptr = calloc(m, n)" is equivalent to following
  ptr = malloc(n); memset(ptr, 0, n);

Dynamic Memory Allocation

Discuss it

Question 6 Explanation:

See calloc() versus malloc() for details.

Question 7

What is the problem with following code?

```
#include<stdio.h>
int main()
{
    int *p = (int *)malloc(sizeof(int));

    p = NULL;

    free(p);
}
```

ACompiler Error: free can't be applied on NULL pointer

BMemory Leak

CDangling Pointer

DThe program may crash as free() is called for NULL pointer.

Dynamic Memory Allocation

Discuss it

Question 7 Explanation:

free() can be called for NULL pointer, so no problem with free function call. The problem is memory leak, p is allocated some memory which is not freed, but the pointer is assigned as NULL. The correct sequence should be following:

```
    free(p);
    p = NULL;
```

Question 8

Consider the following program, where are i, j and k are stored in memory?

```
int i;
int main()
{
   int j;
   int *k = (int *) malloc (sizeof(int));
}
```

Ai, j and k are stored in stack segment
Bi and j are stored in stack segment. k is stored on heap.
Ci is stored in BSS part of data segment, j is stored in stack segment. k is stored on heap.
Dj is stored in BSS part of data segment, i is stored in stack segment. k is stored on heap.
Dynamic Memory Allocation
Discuss it
Question 8 Explanation:
i is global variable and it is uninitialized so it is stored on BSS part of Data Segment (http://en.wikipedia.org/wiki/.bss) j is local in main() so it is stored in stack frame (http://en.wikipedia.org/wiki/Call_stack) k is dynamically allocated so it is stored on Heap Segment. See following article for more details. Memory Layout of C Programs
There are 8 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Question 1
In the C language (GATE CS 2002)
AAt most one activation record exists between the current activation record and the activation record for the main
BThe number of activation records between the current activation record and the activation record for the main depends on the actual function calling sequence.
CThe visibility of global variables depends on the actual function calling sequence.
DRecursion requires the activation record for the recursive function to be saved on a different stack before the recursive function can be called.
Misc
Discuss it
Question 1 Explanation:
a) –> There is no such restriction in C language b) –> True c) –> False. In C, variables are statically scoped, not dynamically. c) –> False. The activation records are stored on the same stack.
Question 2
The C language is. (GATE CS 2002)
AA context free language
BA context sensitive language
CA regular language
DParsable fully only by a Turing machine
Misc
Discuss it
Question 2 Explanation:
Most programming languages including C, C++, Java and Pascal can be approximated by context-free grammar and compilers for them have been developed based on properties of context-free languages.
Question 3
The number of tokens in the following C statement is (GATE 2000)

```
printf("i = %d, &i = %x", i, &i);
```

A3
B26
C10
D21
Misc **Lexical analysis**
Discuss it
Question 3 Explanation:
In a C source program, the basic element recognized by the compiler is the “token.” A token is source-program text that the compiler does not break down into component elements. There are 6 types of C tokens : identifiers, keywords, constants, operators, string literals and other separators. There are total 10 tokens in the above printf statement. Below are tokens in above program.

```
printf
(
"i = %d, &i = %x"
,
i
,
&
i
)
;
```

Question 4
Which of the following best describes C language
AC is a low level language
BC is a high level language with features that support low level programming
CC is a high level language
DC is a very high level language
Misc
Discuss it
Question 4 Explanation:
See http://geeksquiz.com/c-language-set-1-introduction/
Question 5
Assume that the size of an integer is 4 bytes. Predict the output?

```
#include <stdio.h>
int fun()
{
   puts(" Hello ");
   return 10;
}

int main()
{
   printf("%d", sizeof(fun()));
   return 0;
}
```

A4
BHello 4
C4 Hello
DCompiler Error
Misc
Discuss it
Question 5 Explanation:
sizeof() is an operator, not a function. It looks like a function though. The operands of operators need not to be evaluated. That is why fun() is not called.
Question 6

```
#include <stdio.h>
int main()
{
   int a[][3] = {1, 2, 3, 4, 5, 6};
   int (*ptr)[3] = a;
   printf("%d %d ", (*ptr)[1], (*ptr)[2]);
   ++ptr;
   printf("%d %d\n", (*ptr)[1], (*ptr)[2]);
   return 0;
}
```

A2 3 5 6
B2 3 4 5
C4 5 0 0
Dnone of the above
Misc
Discuss it
Question 7
Consider line number 3 of the following C- program.

```
int main ( ) {                /* Line 1 */
  int I, N;                   /* Line 2 */
  fro (I = 0, I < N, I++);      /* Line 3 */
}
```

Identify the compiler's response about this line while creating the object-module
ANo compilation error
BOnly a lexical error
COnly syntactic errors
DBoth lexical and syntactic errors
Misc GATE-CS-2005
Discuss it
Question 7 Explanation:
Note that there is 'fro' instead of 'for'. This is not a lexical error as lexical analysis typically involves Tokenization.
Question 8
Consider a program P that consists of two source modules M1 and M2 contained in two different files. If M1 contains a reference to a function defined in M2, the reference will be resolved at
AEdit-time
BCompile-time
CLink-time
DLoad-time
Misc GATE-CS-2004
Discuss it
Question 8 Explanation:
Note: Static linking is done at link-time, dynamic linking or shared libraries are brought in only at runtime. (A) Edit-time : Function references can never be given/determined at edit-time or code-writing time. Function references are different from function names. Function names are used at edit-time and function references are determined at linker-time for static libraries or at runtime for dynamic libraries. (B) Compile-time : Compile time binding is done for functions present in the same file or module. (C) Link-time : Link time binding is done in the linker stage, where functions present in separate files or modules are referenced in the executable. (D) Load-time : Function referencing is not done at load-time. Hence, correct answer would be (C). This solution is contributed by **Vineet Purswani**.
Question 9
Assume the following C variable declaration

```
int *A [10], B [10][10];
```

Of the following expressions

```
I.  A [2]
II.  A [2][3]
III. B [1]
IV.  B [2][3]
```

which will not give compile-time errors if used as left hand sides of assignment statements in a C program ?
AI, II and IV only
BII, III and IV only
CII and IV only
DIV only
Misc
Discuss it
Question 9 Explanation:
See http://geeksquiz.com/c-arrays-question-6/ for explanation.
Question 10
Consider the following C program:

```
# include <stdio.h>
int main( )
{
  int i, j, k = 0;
  j = 2 * 3 / 4 + 2.0 / 5 + 8 / 5;
  k  -= --j;
  for (i = 0; i < 5; i++)
  {
    switch(i + k)
    {
      case 1:
      case 2: printf("\n%d", i + k);
      case 3: printf("\n%d", i + k);
      default: printf("\n%d", i + k);
    }
  }
  return 0;
}
```

The number of times printf statement is executed is __________.
A8
B9
C10
D11
Misc GATE-CS-2015 (Set 3)
Discuss it
Question 10 Explanation:
The following statement makes j = 2

```
j = 2 * 3 / 4 + 2.0 / 5 + 8 / 5;
```

The following statement makes k = -1.

```

k  -= --j;
```

There is one important thing to note in switch is, there is no break. Let count of printf statements be 'count'

```
For i = 0, the value of i+k becomes -1, default block
         is executed, count = 1.
For i = 1, the value of i+k becomes 0, default block
         is executed, count = 2.
For i = 2, the value of i+k becomes 1, all blocks are
         executed as there is no break, count = 5
For i = 3, the value of i+k becomes 2, three blocks
         after case 1: are executed, count = 8
For i = 4, the value of i+k becomes 3, two blocks
         are executed, count = 10
```

Question 11
Output of Below C Code? Assume that int takes 4 bytes.

```
#include<stdio.h>
int x = 5;
int main()
{
 int arr[x];
 static int x = 0;
 x = sizeof(arr);
 printf("%d", x<<2);
 return 0;
}
```

Thanks to Gokul Kumar for contributing this question.
ACompiler error in line "static int x = 0"
B7
C80
D20
Misc
Discuss it
Question 11 Explanation:
Size of gives size of arr * int in bytes = 20 Left shift two times gives you 80.
Question 12

```
#include <stdio.h>
#include <string.h>
int main()
{
   char a[] = {'G','E','E','K','S','Q','U','I','Z'};
   char b[] = "QUIZ";
   char c[] = "GEEKS";
   char d[] = "1234";
   int l = strlen(a);
   int o = printf("%d", sizeof((sizeof(l)+(c[5]+d[0]+a[1]+b[2]))) );
   printf("%c", a[o]);
   return 0;
}
```

Thanks to Gokul for contributing this question.
A4E
B8E

C1234Q
DCompiler Dependent
Misc
Discuss it
Question 12 Explanation:
The output seems is compiler dependent. It depends on the size of return type of sizeof. The return type of sizeof is std::size_t. The size of size_t is 4 bytes in some compilers and 8 bytes in some other.
Question 13
In the context of the following printf() in C, pick the best statement.

```
i) printf("%d",8);
ii) printf("%d",090);
iii) printf("%d",00200);
iv) printf("%d",0007000);
```

AOnly i) would compile. And it will print 8.
BBoth i) and ii) would compile. i) will print 8 while ii) will print 90
CAll i), ii), iii) and iv) would compile successfully and they will print 8, 90, 200 & 7000 respectively.
DOnly i), iii) and iv) would compile successfully. They will print 8, 128 and 3584 respectively.
Misc C Quiz - 102
Discuss it
Question 13 Explanation:
As per C standard, "*An octal constant consists of the prefix 0 optionally followed by a sequence of the digits 0 through 7 only.*"

So 090 isn't valid because 0 prefix is used for octal but 9 isn't valid octal-digit.

Question 14
As per C language standard, which of the followings is/are not keyword(s)? Pick the best statement. **auto make main sizeof elseif**
ANone of the above is keywords in C.
Bmake main elseif
Cmake main
Dauto make
Esizeof elseif make
Misc C Quiz - 103
Discuss it
Question 14 Explanation:
"auto" is a storage specifier defined in C language. "sizeof" is unary operator defined in C language. Both of these are reserved words i.e. keywords as per C language standard. All the other 3 i.e. main make and elseif aren't keywords. In fact, you can use *int main, make, elseif;* in your C program.
Question 15
When a coin is tossed, the probability of getting a Head is p,0<p<1. Let N be the random variable denoting the number of tosses till the first Head appears, including the toss where the Head appears. Assuming that successive tosses are independent, the expected value of N is
A1/p
B1/(1–p)
C1/p2
D1/(1–p2)
Misc Probability GATE IT 2006
Discuss it
Question 15 Explanation:
For a continuous variable X ranging over all the real numbers, the expectation is defined by E(X)= ∫ xf(x) dx probability for head = p probability for tail = 1-p if in first time, head appears, probability will be 1*p if firstly tail occurs,and then head occurs, then the probability will be (1-p)*p and so on.... for the nth time, probaility will be (1-p) n-1 * p E= 1*p + 2*(1–p)*p + 3*(1–p)*(1–p)*p + ... equation(1) multiply both side with (1–p): E*(1-p) = 1*p*(1-p) + 2*(1-p)*(1-p)*p + 3*(1-p)*(1-p)*(1-p)*p +............ equation (2) Subtract equation 2 from equation 1: E–E*(1–p)= 1*p+ (1–p)*p+ (1–p)*(1–p)*p +... E*p =p[1+ (1-p) + (1-p)*(1-p) + ...] It's a infinite geometric progression. E = 1/(1-(1-p)) = 1/p E=1/p correct answer is A. This solution is contributed by **Nitika Bansal**.
Question 16
The minimum positive integer p such that 3^p modulo 17 = 1 is
A5
B8
C12
D16
Misc Network Security Gate IT 2007
Discuss it
Question 16 Explanation:
Either use Fermat's Little Theorem (reference)
Or put the value of p and get the answer
Question 17
The minimum frame size required for a CSMA/CD based computer network running at 1 Gbps on a 200m cable with a link speed of 2 × 10^8m/s is
A125 bytes

B250 bytes
C500 bytes
DNone of these
Data Link Layer **Misc** **Gate IT 2008**
Discuss it
Question 17 Explanation:
2008_65_sol

There are 17 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Pointers store address of variables or a memory location.

```
// General syntax
datatype *var_name;

// An example pointer "ptr" that holds
// address of an integer variable or holds
// address of a memory whose value(s) can
// be accessed as integer values through "ptr"
int *ptr;
```

Read More..

Question 1

```
void fun(int *p)
{
  int q = 10;
  p = &q;
}

int main()
{
  int r = 20;
  int *p = &r;
  fun(p);
  printf("%d", *p);
  return 0;
}
```

A10
B20
CCompiler error
DRuntime Error
Advanced Pointer
Discuss it
Question 1 Explanation:
Inside fun(), q is a copy of the pointer p. So if we change q to point something else then p remains uneffected. If we want to change a local pointer of one function inside another function, then we must pass pointer to the pointer. By passing the pointer to the pointer, we can change pointer to point to something else. See the following program as an example.

```
void fun(int **pptr)
{
  static int q = 10;
  *pptr = &q;
}

int main()
{
  int r = 20;
  int *p = &r;
  fun(&p);
  printf("%d", *p);
  return 0;
}
```

In the above example, the function fun() expects a double pointer (pointer to a pointer to an integer). Fun() modifies the value at address pptr. The value at address pptr is pointer p as we pass adderess of p to fun(). In fun(), value at pptr is changed to address of q. Therefore, pointer p of main() is changed to point to a new variable q. Also, note that the program won't cause any out of scope problem because q is a static variable. Static variables exist in memory even after functions return. For an auto variable, we might have seen some unexpected output because auto variable may not exist in memory after functions return.
Question 2
Assume sizeof an integer and a pointer is 4 byte. Output?

```
#include <stdio.h>

#define R 10
#define C 20

int main()
{
   int (*p)[R][C];
   printf("%d",  sizeof(*p));
   getchar();
   return 0;
}
```

A200
B4
C800
D80
Advanced Pointer
Discuss it
Question 2 Explanation:
Output is 10*20*sizeof(int) which is "800″ for compilers with integer size as 4 bytes. When a pointer is de-referenced using *, it yields type of the object being pointed. In the present case, it is an array of array of integers. So, it prints R*C*sizeof(int).
Question 3

```
#include <stdio.h>
int main()
{
    int a[5] = {1,2,3,4,5};
    int *ptr = (int*)(&a+1);
    printf("%d %d", *(a+1), *(ptr-1));
    return 0;
}
```

A2 5
BGarbage Value
CCompiler Error
DSegmentation Fault
Advanced Pointer
Discuss it
Question 3 Explanation:
The program prints "2 5″. Since compilers convert array operations in pointers before accessing the array elements, (a+1) points to 2. The expression (&a + 1) is actually an address just after end of array (after address of 5) because &a contains address of an item of size 5*integer_size and when we do (&a + 1) the pointer is incremented by 5*integer_size. ptr is type-casted to int * so when we do ptr -1, we get address of 5
Question 4

```
#include <stdio.h>

char *c[] = {"GeksQuiz", "MCQ", "TEST", "QUIZ"};
char **cp[] = {c+3, c+2, c+1, c};
char ***cpp = cp;
```

```
int main()
{
    printf("%s ", **++cpp);
    printf("%s ", *--*++cpp+3);
    printf("%s ", *cpp[-2]+3);
    printf("%s ", cpp[-1][-1]+1);
    return 0;
}
```

ATEST sQuiz Z CQ
BMCQ Quiz Z CQ
CTEST Quiz Z CQ
D*GarbageValue* sQuiz Z CQ
Advanced Pointer
Discuss it
Question 4 Explanation:
Let us first consider **++cpp. Precedence of prefix increment and de-reference is same and associativity of both of them is right to left. So the expression is evaluated as **(++cpp). So cpp points to c+2. So we get "TEST" as output. Note the de-reference operator twice. Similarly, you may try other expressions yourself with the help of precedence table.
Question 5
Predict the output

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

void fun(char** str_ref)
{
    str_ref++;
}

int main()
{
    char *str = (void *)malloc(100*sizeof(char));
    strcpy(str, "GeeksQuiz");
    fun(&str);
    puts(str);
    free(str);
    return 0;
}
```

AGeeksQuiz
BeeksQuiz
CGarbage Value
DCompiler Error
Advanced Pointer
Discuss it
Question 5 Explanation:
Note that str_ref is a local variable to fun(). When we do str_ref++, it only changes the local variable str_ref. We can change str pointer using dereference operator *. For example, the following program prints "eeksQuiz"

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

void fun(char** str_ref)
{
    (*str_ref)++;
}

int main()
{
    char *str = (void *)malloc(100*sizeof(char));
    strcpy(str, "GeeksQuiz");
    fun(&str);
    puts(str);
    free(str);
    return 0;
}
```

Question 6
Assume that the size of int is 4.

```
#include <stdio.h>
void f(char**);
int main()
{
   char *argv[] = { "ab", "cd", "ef", "gh", "ij", "kl" };
   f(argv);
   return 0;
}
void f(char **p)
{
   char *t;
   t = (p += sizeof(int))[-1];
   printf("%s\n", t);
}
```

Aab
Bcd
Cef
Dgh
Advanced Pointer
Discuss it
Question 6 Explanation:
The expression (p += sizeof(int))[-1] can be written as (p += 4)[-1] which can be written as (p = p+4)[-] which returns address p+3 which is address of fourth element in argv[].
Question 7

```
#include <stdio.h>
int main()
{
   int a[][3] = {1, 2, 3, 4, 5, 6};
   int (*ptr)[3] = a;
   printf("%d %d ", (*ptr)[1], (*ptr)[2]);
   ++ptr;
   printf("%d %d\n", (*ptr)[1], (*ptr)[2]);
   return 0;
}
```

A2 3 5 6
B2 3 4 5
C4 5 0 0
Dnone of the above
Advanced Pointer
Discuss it
Question 8

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

int main(void)
{
   int i;
   int *ptr = (int *) malloc(5 * sizeof(int));

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

   printf("%d ", *ptr++);
   printf("%d ", (*ptr)++);
   printf("%d ", *ptr);
   printf("%d ", *++ptr);
   printf("%d ", ++*ptr);
}
```

ACompiler Error
B0 1 2 2 3
C0 1 2 3 4
D1 2 3 4 5
Advanced Pointer

Discuss it

Question 8 Explanation:

The important things to remember for handling such questions are **1)** Prefix ++ and * operators have same precedence and right to left associativity. **2)** Postfix ++ has higher precedence than the above two mentioned operators and associativity is from left to right. We can apply the above two rules to guess all *ptr++ is treated as *(ptr++) *++ptr is treated as *(++ptr) ++*ptr is treated as ++(*ptr)

Question 9

Output of following program

```
#include <stdio.h>
int fun(int arr[]) {
   arr = arr+1;
   printf("%d ", arr[0]);
}
int main(void) {
   int arr[2] = {10, 20};
   fun(arr);
   printf("%d", arr[0]);
   return 0;
}
```

ACompiler Error

B20 10

C20 20

D10 10

Advanced Pointer

Discuss it

Question 9 Explanation:

In C, array parameters are treated as pointers (See http://www.geeksforgeeks.org/why-c-treats-array-parameters-as-pointers/ for details). So the variable *arr* represents an array in main(), but a pointer in fun().

Question 10

What is printed by the following C program?

```
$include <stdio.h>
int f(int x, int *py, int **ppz)
{
  int y, z;
  **ppz += 1;
   z  = **ppz;
  *py += 2;
   y = *py;
   x += 3;
   return x + y + z;
}

void main()
{
   int c, *b, **a;
   c = 4;
   b = &c;
   a = &b;
   printf( "%d", f(c,b,a));
   getchar();
}
```

A18

B19

C21

D22

Advanced Pointer   **GATE CS 2008**

Discuss it

Question 10 Explanation:

See Question 1 of http://www.geeksforgeeks.org/c-language-set-5/

Question 11

What is the output of the following C code? Assume that the address of x is 2000 (in decimal) and an integer requires four bytes of memory.

```
#include <stdio.h>
int main()
{
   unsigned int x[4][3] = {{1, 2, 3}, {4, 5, 6},
                          {7, 8, 9}, {10, 11, 12}};
```

```
  printf("%u, %u, %u", x+3, *(x+3), *(x+2)+3);
}
```

A2036, 2036, 2036
B2012, 4, 2204
C2036, 10, 10
D2012, 4, 6
Advanced Pointer **GATE-CS-2015 (Set 1)**
Discuss it
Question 11 Explanation:

```
x = 2000

Since x is considered as a pointer to an
array of 3 integers and an integer takes 4
bytes, value of x + 3 = 2000 + 3*3*4 = 2036

The expression, *(x + 3) also prints same
address as x is 2D array.


The expression *(x + 2) + 3 = 2000 + 2*3*4 + 3*4
                        = 2036
```

Question 12
Consider the following C program.

```
# include
int main( )
{
  static int a[] = {10, 20, 30, 40, 50};
  static int *p[] = {a, a+3, a+4, a+1, a+2};
  int **ptr = p;
  ptr++;
  printf("%d%d", ptr - p, **ptr};
}
```

The output of the program is _________
A140
B120
C100
D40
Advanced Pointer **GATE-CS-2015 (Set 3)**
Discuss it
Question 12 Explanation:
In order to simplify programs involving complex operations on pointers, we suggest you to draw proper diagrams in order to avoid silly mistakes. Let's assume that integer is of 4 Bytes and Pointer size is also 4 Bytes. Let's assume array a Base address is 1000. Array pranjul_36 name actually holds the array base address. pranjul_36_1 Let's assume array p Base address is 2000. pranjul_36_2 Double Pointer ptr Base Address is 3000. Now ptr is actually pointing to the first element of array p. ptr++ will make it point to the next element of array p. Its value will then change to 2004. One of the Rule of Pointer Arithmetic is that When you subtract two pointers, as long as they point into the same array, the result is the number of elements separating them. ptr is

pointing to the second element and p is pointing to the first element so ptr-p will be equal to 1(Excluding the element to which ptr is pointing). Now ptr = 2004 -----> *(2004) = 1012 ----> *(1012) ----> 40 which is the final answer. This solution is contributed by **Pranjul Ahuja**. .
Question 13

```
#include "stdlib.h"
int main()
{
 int *pInt;
 int **ppInt1;
 int **ppInt2;

 pInt = (int*)malloc(sizeof(int));
 ppInt1 = (int**)malloc(10*sizeof(int*));
 ppInt2 = (int**)malloc(10*sizeof(int*));

 free(pInt);
 free(ppInt1);
 free(*ppInt2);
 return 0;
}
```

Choose the correct statement w.r.t. above C program.
Amalloc() for ppInt1 and ppInt2 isn’t correct. It’ll give compile time error.
Bfree(*ppInt2) is not correct. It’ll give compile time error.
Cfree(*ppInt2) is not correct. It’ll give run time error.
DNo issue with any of the malloc() and free() i.e. no compile/run time error.
Advanced Pointer **C Quiz - 101**
Discuss it
Question 13 Explanation:
ppInt2 is pointer to pointer to int. *ppInt2 is pointer to int. So long as the argument of free() is pointer, there's no issue. That's why B) and C) both are not correct. Allocation of both ppInt1 and ppInt2 is fine as per their data type. So A) is also not correct. The correct statement is D).
Question 14

```
#include "stdio.h"
int main()
{
 void *pVoid;
 pVoid = (void*)0;
 printf("%lu",sizeof(pVoid));
 return 0;
}
```

Pick the best statement for the above C program snippet.
AAssigning (void *)0 to pVoid isn’t correct because memory hasn’t been allocated. That’s why no compile error but it'll result in run time error.
BAssigning (void *)0 to pVoid isn’t correct because a hard coded value (here zero i.e. 0) can’t assigned to any pointer. That’s why it'll result in compile error.
CNo compile issue and no run time issue. And the size of the void pointer i.e. pVoid would equal to size of int.
Dsizeof() operator isn’t defined for a pointer of void type.
Advanced Pointer **C Quiz - 101**
Discuss it
Question 14 Explanation:
(void *)0 is basically NULL pointer which is used for many purposes in C. Please note that no matter what is the type of pointer, each pointer holds some address and the size of every pointer is equal to sizeof(int). So D) isn't correct. An absolute address can be assigned to any pointer though it might result in issues at run time if the address is illegal. Since 0 is a legal address, assigning (void *)0 to pVoid is fine. So B) isn't correct. We aren't doing any illegal operation with pVoid here. So it'll not result in any compile/run time error. So A) isn't correct. For example, if we perform illegal operation over pVoid e.g. de-referencing of void pointer i.e. *pVoid, it'll result in error. The above program will compile/run without any issue. So C) is correct.
Question 15
In the context of the below program snippet, pick the best answer.

```
#include "stdio.h"
int arr[10][10][10];
int main()
{
 arr[5][5][5] = 123;
 return 0;
}
```

Which of the given printf statement(s) would be able to print arr[5][5][5]

```
(i) printf("%d",arr[5][5][5]);
(ii) printf("%d",*(*(*(arr+5)+5)+5));
(iii) printf("%d",(*(*(arr+5)+5))[5]);
(iv) printf("%d",*((*(arr+5))[5]+5));
```

Aonly (i) would compile and print 123.
Bboth (i) and (ii) would compile and both would print 123.
Conly (i), (ii) and (iii) would compile but only (i) and (ii) would print 123.
Donly (i), (ii) and (iii) would compile and all three would print 123.
Eall (i), (ii), (iii) and (iv) would compile but only (i) and (ii) would print 123.
Fall (i), (ii), (iii) and (iv) would compile and all would print 123.
Advanced Pointer   **C Quiz - 102**
Discuss it
Question 15 Explanation:
For arrays, we can convert array subscript operator [] to pointer deference operator * with proper offset. It means that arr[x] is equal to *(arr+x). Basically, these two are interchangeable. The same concept can be applied in multi-dimensional arrays as well. That's why all of the above 4 printf are referring to the same element i.e. *arr[5][5][5]*
Question 16
Find out the correct statement for the following program.

```
#include "stdio.h"

typedef int (*funPtr)(int);

int inc(int a)
{
 printf("Inside inc() %d\n",a);
 return (a+1);
}

int main()
{

 funPtr incPtr1 = NULL, incPtr2 = NULL;

 incPtr1 = &inc; /* (1) */
 incPtr2 = inc; /* (2) */

 (*incPtr1)(5); /* (3) */
 incPtr2(5); /* (4)*/

 return 0;
}
```

ALine with comment (1) will give compile error.
BLine with comment (2) will give compile error.
CLines with (1) & (3) will give compile error.
DLines with (2) & (4) will give compile error.
ENo compile error and program will run without any issue.

Advanced Pointer   **C Quiz - 109**
Discuss it
Question 16 Explanation:
While assigning any function to function pointer, & is optional. Same way, while calling a function via function pointer, * is optional.
There are 16 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Question 1
Which of the following true about FILE *fp
AFILE is a keyword in C for representing files and fp is a variable of FILE type.

BFILE is a structure and fp is a pointer to the structure of FILE type
CFILE is a stream
DFILE is a buffered stream
File Handling
Discuss it
Question 1 Explanation:
fp is a pointer of FILE type and FILE is a structure that store following information about opened file. 1
Question 2
When fopen() is not able to open a file, it returns
AEOF
BNULL
CRuntime Error
DCompiler Dependent
File Handling
Discuss it
Question 2 Explanation:
fopen() returns NULL if it is not able to open the given file due to any of the reasons like file not present, inappropriate permissions, etc.
Question 3
getc() returns EOF when
AEnd of files is reached
BWhen getc() fails to read a character
CBoth of the above
DNone of the above
File Handling
Discuss it
Question 3 Explanation:
See EOF, getc() and feof() in C
Question 4
In fopen(), the open mode "wx" is sometimes preferred "w" because. 1) Use of wx is more efficient. 2) If w is used, old contents of file are erased and a new empty file is created. When wx is used, fopen() returns NULL if file already exists.
AOnly 1
BOnly 2
CBoth 1 and 2
DNeither 1 nor 2
File Handling
Discuss it
Question 4 Explanation:
See http://www.geeksforgeeks.org/fopen-for-an-existing-file-in-write-mode/
Question 5
fseek() should be preferred over rewind() mainly because
Arewind() doesn't work for empty files
Brewind() may fail for large files
CIn rewind, there is no way to check if the operations completed successfully
DAll of the above
File Handling
Discuss it
Question 5 Explanation:
Please see fseek() vs rewind() in C
There are 5 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Commonly Asked C Programming Interview Questions | Set 1

What is the difference between declaration and definition of a variable/function

Ans: Declaration of a variable/function simply declares that the variable/function exists somewhere in the program but the memory is not allocated for them. But the declaration of a variable/function serves an important role. And that is the type of the variable/function. Therefore, when a variable is declared, the program knows the data type of that variable. In case of function declaration, the program knows what are the arguments to that functions, their data types, the order of arguments and the return type of the function. So that’s all about declaration. Coming to the definition, when we define a variable/function, apart from the role of declaration, it also allocates memory for that variable/function. Therefore, we can think of definition as a super set of declaration. (or declaration as a subset of definition). From this explanation, it should be obvious that a variable/function can be declared any number of times but it can be defined only once.

(Remember the basic principle that you can't have two locations of the same variable/function).

```
 // 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;
```

What are different storage class specifiers in C?
Ans: auto, register, static, extern

What is scope of a variable? How are variables scoped in C?
Ans: Scope of a variable is the part of the program where the variable may directly be accessible. In C, all identifiers are lexically (or statically) scoped. See this for more details.

How will you print "Hello World" without semicolon?
Ans:

```
int main(void)
{
   if (printf("Hello World")) ;
}
```

See print "Geeks for Geeks" without using a semicolon for answer.

When should we use pointers in a C program?
1. To get address of a variable
2. *For achieving pass by reference in C:* Pointers allow different functions to share and modify their local variables.
3. *To pass large structures* so that complete copy of the structure can be avoided.
C
4. *To implement "linked" data structures* like linked lists and binary trees.

What is NULL pointer?
Ans: NULL is used to indicate that the pointer doesn't point to a valid location. Ideally, we should initialize pointers as NULL if we don't know their value at the time of declaration. Also, we should make a pointer NULL when memory pointed by it is deallocated in the middle of a program.

What is Dangling pointer?
Ans: Dangling Pointer is a pointer that doesn't point to a valid memory location. Dangling pointers arise when an object is deleted or deallocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the deallocated memory. Following are examples.

```
// EXAMPLE 1
int *ptr = (int *)malloc(sizeof(int));
............
............
free(ptr);

// ptr is a dangling pointer now and operations like following are invalid
*ptr = 10;  // or printf("%d", *ptr);
```

```
// EXAMPLE 2
int *ptr = NULL
{
  int x  = 10;
  ptr = &x;
}
// x goes out of scope and memory allocated to x is free now.
// So ptr is a dangling pointer now.
```

What is memory leak? Why it should be avoided
Ans: Memory leak occurs when programmers create a memory in heap and forget to delete it. Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminate.

```
/* Function with memory leak */
#include <stdlib.h>
```

```
void f()
{
  int *ptr = (int *) malloc(sizeof(int));

  /* Do some work */

  return; /* Return without freeing ptr*/
}
```

What are local static variables? What is their use?

Ans:A local static variable is a variable whose lifetime doesn't end with a function call where it is declared. It extends for the lifetime of complete program. All calls to the function share the same copy of local static variables. Static variables can be used to count the number of times a function is called. Also, static variables get the default value as 0. For example, the following program prints "0 1″

```
#include <stdio.h>
void fun()
{
   // static variables get the default value as 0.
   static int x;
   printf("%d ", x);
   x = x + 1;
}

int main()
{
   fun();
   fun();
   return 0;
}
// Output: 0 1
```

What are static functions? What is their use?

Ans: In C, functions are global by default. The "static" keyword before a function name makes it static. Unlike global functions in C, access to static functions is restricted to the file where they are declared. Therefore, when we want to restrict access to functions, we make them static. Another reason for making functions static can be reuse of the same function name in other files. See this for examples and more details.

- Commonly Asked C Programming Interview Questions | Set 2
- Practices Quizzes on C
- C articles

You may also like:

Commonly Asked C Programming Interview Questions | Set 2
Commonly Asked Java Programming Interview Questions | Set 1
Amazon's most asked interview questions
Microsoft's most asked interview questions
Accenture's most asked Interview Questions
Commonly Asked OOP Interview Questions
Commonly Asked C++ Interview Questions
Commonly asked DBMS interview questions | Set 1
Commonly asked DBMS interview questions | Set 2
Commonly Asked Operating Systems Interview Questions | Set 1
Commonly Asked Data Structure Interview Questions.
Commonly Asked Algorithm Interview Questions
Commonly asked Computer Networks Interview Questions
Top 10 algorithms in Interview Questions

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

Commonly Asked C Programming Interview Questions | Set 2

This post is second set of Commonly Asked C Programming Interview Questions | Set 1

What are main characteristics of C language?

C is a procedural language. The main features of C language include low-level access to memory, simple set of keywords, and clean style. These features make it suitable for system programming like operating system or compiler development.

What is difference between i++ and ++i?

1) The expression 'i++' returns the old value and then increments i. The expression ++i increments the value and returns new value.
2) Precedence of postfix ++ is higher than that of prefix ++.
3) Associativity of postfix ++ is left to right and associativity of prefix ++ is right to left.
4) In C++, ++i can be used as l-value, but i++ cannot be. In C, they both cannot be used as l-value.
See Difference between ++*p, *p++ and *++p for more details.

What is l-value?

l-value or location value refers to an expression that can be used on left side of assignment operator. For example in expression "a = 3", a is l-value and 3 is r-value.
l-values are of two types:
"nonmodifiable l-value" represent a l-value that can not be modified. const variables are "nonmodifiable l-value".
"modifiable l-value" represent a l-value that can be modified.

What is the difference between array and pointer?

See Array vs Pointer

How to write your own sizeof operator?

```
#define my_sizeof(type) (char *)(&type+1)-(char*)(&type)
```

See Implement your own sizeof for more details.

How will you print numbers from 1 to 100 without using loop?

We can use recursion for this purpose.

```
/* Prints numbers from 1 to n */
void printNos(unsigned int n)
{
  if(n > 0)
  {
    printNos(n-1);
    printf("%d ",  n);
  }
}
```

What is volatile keyword?

The volatile keyword is intended to prevent the compiler from applying any optimizations on objects that can change in ways that cannot be determined by the compiler.
Objects declared as volatile are omitted from optimization because their values can be changed by code outside the scope of current code at any time. See Understanding "volatile" qualifier in C for more details.

Can a variable be both const and volatile?

yes, the const means that the variable cannot be assigned a new value. The value can be changed by other code or pointer. For example the following program works fine.

```
int main(void)
{
    const volatile int local = 10;
    int *ptr = (int*) &local;
    printf("Initial value of local : %d \n", local);
    *ptr = 100;
    printf("Modified value of local: %d \n", local);
    return 0;
}
```

- Practices Quizzes on C
- C articles

We will soon be publishing more sets of commonly asked C programming questions.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner