C C++

1. How will you print numbers from 1 to 100 without using loop?

Here is a solution that prints numbers using recursion.

Other alternatives for loop statements are recursion and goto statement, but use of goto is not suggestible as a general programming practice as goto statement changes the normal program execution sequence and makes it difficult to undestand and maintain.

```
#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.

2. How can we sum the digits of a given number in single statement?

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.

3. Write a C program to calculate pow(x,n)

Below solution divides the problem into subproblems of size y/2 and call the subproblems recursively.

```
#include<stdio.h>

/* Function to calculate x raised to the power y */
int power(int x, unsigned int y)
{
    if( y == 0)
        return 1;
    else if (y%2 == 0)
        return power(x, y/2)*power(x, y/2);
    else
```

```
        return x*power(x, y/2)*power(x, y/2);

}

/* Program to test function power */
int main()
{
    int x = 2;
    unsigned int y = 3;

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

Time Complexity: O(n)
Space Complexity: O(1)
Algorithmic Paradigm: Divide and conquer.

Above function can be optimized to O(logn) by calculating power(x, y/2) only once and storing it.

```
/* Function to calculate x raised to the power y in O(logn)*/
int power(int x, unsigned int y)
{
    int temp;
    if( y == 0)
        return 1;
    temp = power(x, y/2);
    if (y%2 == 0)
        return temp*temp;
    else
        return x*temp*temp;
}
```

Time Complexity of optimized solution: O(logn)
Let us extend the pow function to work for negative y and float x.

```
/* Extended version of power function that can work
 for float x and negative y*/
#include<stdio.h>

float power(float x, int y)
{
    float temp;
    if( y == 0)
       return 1;
    temp = power(x, y/2);
    if (y%2 == 0)
        return temp*temp;
    else
    {
        if(y > 0)
```

```
        return x*temp*temp;
    else
        return (temp*temp)/x;
  }
}

/* Program to test function power */
int main()
{
  float x = 2;
  int y = -3;
  printf("%f", power(x, y));
  getchar();
  return 0;
}
```

4. 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.

5. 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.

6. 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)

7. 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”) < 0) or
if (printf(“Hello”) < 1) etc

Please comment if you find more solutions of this.

8. 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 < n; i--)
      printf("*");
   getchar();
   return 0;
}
```

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 < by + 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;
}
```

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.

9. 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.

10. 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;
}
```

11. 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!

12. 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.

13. 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.

14. 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**.

C C++

15. 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", "%");
```

16. 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!

17. 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.

18. 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

19. What all is inherited from parent class in C++?

Following are the things which a derived class inherits from its parent.
1) Every data member defined in the parent class (although such members may not always be
accessible in the derived class!)
2) Every ordinary member function of the parent class (although such members may not always be
accessible in the derived class!)
3) The same initial data layout as the base class.

Following are the things which a derived class doesn't inherits from its parent :
1) The base class's constructors and destructor.
2) The base class's friends

20. 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

21. 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/DOCU00
http://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html#The-sizeof-Operator

22. 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();
}
```

23. 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.

24. 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.

25. 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.

*26. How does "void *" differ in C and C++?*

C allows a void pointer to be assigned to any pointer type without a cast, whereas C++ does not; this idiom appears often in C code using malloc memory allocation. For example, the following is valid in C but not C++:*

```
void* ptr;
```

```
int *i = ptr; /* Implicit conversion from void* to int* */
```

or similarly:

```
int *j = malloc(sizeof(int) * 5); /* Implicit conversion from void* to int* */
```

In order to make the code compile in both C and C++, one must use an explicit cast:

```
void* ptr;
int *i = (int *) ptr;
int *j = (int *) malloc(sizeof(int) * 5);
```

Source:
http://en.wikipedia.org/wiki/Compatibility_of_C_and_C%2B%2B

27. When is copy constructor called?

In C++, a Copy Constructor may be called in following cases:
1. When an object of the class is returned by value.
2. When an object of the class is passed (to a function) by value as an argument.
3. When an object is constructed based on another object of the same class.
4. When compiler generates a temporary object.

It is however, not guaranteed that a copy constructor will be called in all these cases, because the C++ Standard allows the compiler to optimize the copy away in certain cases, one example being the return value optimization (sometimes referred to as RVO).

References:
http://www.fredosaurus.com/notes-cpp/oop-condestructors/copyconstructors.html
http://en.wikipedia.org/wiki/Copy_constructor

28. What are the operators that cannot be overloaded in C++?

In C++, following operators can not be overloaded:

. (Member Access or Dot operator)
?: (Ternary or Conditional Operator)
:: (Scope Resolution Operator)
. (Pointer-to-member Operator)*
sizeof (Object size Operator)
typeid (Object type Operator)

References:
http://en.wikibooks.org/wiki/C++_Programming/Operators/Operator_Overloading

C C++

29. Static functions in C

In C, functions are global by default. The â€œstaticâ€ 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

```
/* linside file2.c  */
int main(void)
{
  fun1();
  getchar();
  return 0;
}
```

Now, if we compile the above code with command â€œgcc file2.c file1.câ€, we get the error â€œundefined reference to `fun1'â€ . 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.

30. 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

31. 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 < 100 or j < 10) while the FIRST does 11 + 1010 comparisons (i < 10 or j < 100).
c) The SECOND executes 1100 increment operations (i++ or j++) while the FIRST executes 1010 increment operation.

Below C++ code counts the number of increment operations executed in FIRST and

SECOND, and prints the counts.

```
/* program to count number of increment operations in FIRST and SECOND */
#include<iostream>

using namespace std;

int main()
{
  int c1 = 0, c2 = 0;

  /* FIRST */
  for(int i=0;i<10;i++,c1++)
    for(int j=0;j<100;j++, c1++);
   //do something



  /* SECOND */
  for(int i=0; i<100; i++, c2++)
   for(int j=0; j<10; j++, c2++);
  //do something

  cout << " Count in FIRST = " <<c1 << endl;
  cout << " Count in SECOND  = " <<c2 << endl;

  getchar();
  return 0;
}
```

Below C++ code counts the number of comparison operations executed by FIRST and SECOND

```
/* Program to count the number of comparison operations executed by FIRST and SECOND */
#include<iostream>

using namespace std;

int main()
{
   int c1 = 0, c2 = 0;

   /* FIRST */
   for(int i=0; ++c1&&i<10; i++)
      for(int j=0; ++c1&&j<100;j++);
  //do something

   /* SECOND */
   for(int i=0; ++c2&&i<100; i++)
      for(int j=0; ++c2&&j<10; j++);
   //do something

   cout << " Count fot FIRST  " <<c1 << endl;
```

```
  cout << " Count fot SECOND  " <<c2 << endl;
  getchar();
  return 0;
}
```

Thanks to 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.

32. 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
(*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 an
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.