

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 understand 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 $O(\log n)$ by calculating $\text{power}(x, y/2)$ only once and storing it.

```

/* Function to calculate x raised to the power y in  $O(\log n)$  */
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: $O(\log n)$

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 "HelloWorld"

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 arguments.

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](#).

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 :
```

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*/
    }
    getch();
    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 inherit 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/~dgl/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/D/

<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 allocated 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

29. 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.

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 something

/* 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;

    getch();
    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 FI
#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;
    getch();
    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 strcat() and strcmp().

```

/* my_strcat(dest, src) copies data of src to dest. To do so, it first
(*dest++ = *src++)? my_strcat(dest, src). */
void my_strcat(char *dest, char *src)
{
    (*dest)? my_strcat(++dest, src): (*dest++ = *src++)? my_strcat(dest,
}

/* driver function to test above function */
int main()
{
    char dest[100] = "geeksfor";
    char *src = "geeks";
    my_strcat(dest, src);
    printf(" %s ", dest);
    getch();
}

```

The function `my_strncmp()` is simple compared to `my_strcmp()`.

```
/* my_strcmp(a, b) returns 0 if strings a and b are same, otherwise 1.
int my_strcmp(char *a, char *b)
{
    return (*a == *b && *b == '\0')? 0 : (*a == *b)? my_strcmp(++a, ++b)
}

/* 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.

33. 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

34. Can we use function on left side of an expression in C and C++?

In C, it might not be possible to have function names on left side of an expression, but it's possible in C++.

```
#include<iostream>
```

```
using namespace std;
```

```
/* such a function will not be safe if x is non static variable of it */  
int &fun()  
{
```

```
    static int x;  
    return x;  
}
```

```
int main()  
{
```

```
    fun() = 10;
```

```
    /* this line prints 10 on screen */
```

```
    printf(" %d ", fun());
```

```
    getchar();
```

```
    return 0;
```

```
}
```

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

35. When should we write our own copy constructor?

Don't write a copy constructor if shallow copies are ok: In C++, If an object has no pointers or any run time allocation of resource like file handle, a network connection..etc, a shallow copy is probably sufficient. Therefore the default copy constructor, default assignment operator, and default destructor are ok and you don't need to write your own.

Source: <http://www.fredosaurus.com/notes-cpp/oop-condestructors/copyconstructors.html>

36. Can we access global variable if there is a local variable with

same name?

In C, we cannot access a global variable if we have a local variable with same name, but it is possible in C++ using **scope resolution operator (::)**.

```
#include<iostream>

using namespace std;

int x; // Global x

int main()
{
    int x = 10; // Local x
    cout<<"Value of global x is "<<::x<<endl;
    cout<<"Value of local x is "<<x;
    getch();
    return 0;
}
```

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

37. Can references refer to invalid location in C++?

In C++, **Reference variables** are safer than pointers because reference variables must be initialized and they cannot be changed to refer to something else once they are initialized. But there are exceptions where we can have invalid references.

1) Reference to value at uninitialized pointer.

```
int *ptr;
int &ref = *ptr; // Reference to value at some random memory location
```

2) Reference to a local variable is returned.

```
int& fun()
{
    int a = 10;
    return a;
}
```

Once fun() returns, the space allocated to it on stack frame will be taken back. So the reference to a local variable will not be valid.

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

38. Does compiler create default constructor when we write our own?

In C++, compiler by default creates default constructor for every class. But, if we define our own constructor, compiler doesn't create the default constructor.

For example, program 1 compiles without any error, but compilation of program 2 fails with error "no matching function for call to `myInteger::myInteger()' "

Program 1

```
#include<iostream>

using namespace std;

class myInteger
{
    private:
        int value;

        //...other things in class
};

int main()
{
    myInteger I1;
    getchar();
    return 0;
}
```

Program 2

```

#include<iostream>

using namespace std;

class myInteger
{
    private:
        int value;
    public:
        myInteger(int v)  // parametrized constructor
        { value = v; }

        //...other things in class
};

int main()
{
    myInteger I1;
    getchar();
    return 0;
}

```

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

References:

http://en.wikipedia.org/wiki/Default_constructor

[http://publib.boulder.ibm.com/infocenter/lnxpcomp/v8v101/index.jsp?](http://publib.boulder.ibm.com/infocenter/lnxpcomp/v8v101/index.jsp?topic=/com.ibm.xlcpp8l.doc/language/ref/cplr375.htm)

[topic=/com.ibm.xlcpp8l.doc/language/ref/cplr375.htm](http://publib.boulder.ibm.com/infocenter/lnxpcomp/v8v101/index.jsp?topic=/com.ibm.xlcpp8l.doc/language/ref/cplr375.htm)

39. 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. You are not allowed to use any comparison (==, <, >..etc) or conditional (if, else, switch, ternary operator,..etc) statement.

Method 1

Below is a tricky code that 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

40. Can we call an undeclared function in C++?

Calling an undeclared function is poor style in C (See [this](#)) and illegal in C++. So is passing arguments to a function using a declaration that doesn't list argument types:

If we save the below program in a .c file and compile it, it works without any error. But, if we save the same in a .cpp file, it doesn't compile.

```
#include<stdio.h>

void f(); /* Argument list is not mentioned */

int main()
{
    f(2); /* Poor style in C, invalid in C++ */
    getchar();
    return 0;
}

void f(int x)
{
    printf("%d", x);
}

Source: http://www2.research.att.com/~bs/bs\_faq.html#C-is-subset
```

41. 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.

42. Can static functions be virtual in C++?

In C++, a *static* member function of a class cannot be *virtual*. For example, below program gives compilation error.

```
#include<iostream>

using namespace std;

class Test
{
    public:
        // Error: Virtual member functions cannot be static
        virtual static void fun() { }
};
```

Also, *static* member function cannot be *const* and *volatile*. Following code also fails in compilation.

```
#include<iostream>

using namespace std;

class Test
{
    public:
        // Error: Static member function cannot be const
        static void fun() const { }
};
```

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

43. malloc() vs new

Following are the differences between malloc() and operator new.

1) new calls constructors, while malloc() does not. In fact primitive data types (char, int, float.. etc) can also be initialized with new. For example, below program prints 10.

```
#include<iostream>

using namespace std;

int main()
{
    int *n = new int(10); // initialization with new()
    cout<<*n;
    getch();
    return 0;
}
```

2) new is an operator, while malloc() is a function.

3) new returns exact data type, while malloc() returns void *.

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

44. 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
                      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 equivalent 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?](http://publib.boulder.ibm.com/infocenter/comphelp/v101v121/index.jsp?topic=/com.ibm.xlcpp101.aix.doc/language_ref/co.html)

[topic=/com.ibm.xlcpp101.aix.doc/language_ref/co.html](http://msdn.microsoft.com/en-us/library/zs06xbxh.aspx)

<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.

45. delete and free() in C++

In C++, delete operator should only be used either for the pointers pointing to the memory allocated using new operator or for a NULL pointer, and free() should only be used either for the pointers pointing to the memory allocated using malloc() or for a NULL pointer.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int x;
    int *ptr1 = &x;
    int *ptr2 = (int *)malloc(sizeof(int));
    int *ptr3 = new int;
    int *ptr4 = NULL;

    /* delete Should NOT be used like below because x is allocated
       on stack frame */
    delete ptr1;

    /* delete Should NOT be used like below because x is allocated
       using malloc() */
    delete ptr2;

    /* Correct uses of delete */
    delete ptr3;
    delete ptr4;

    getchar();
    return 0;
}
```

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

46. “delete this” in C++

Ideally *delete* operator should not be used for *this* pointer. However, if used, then following points must be considered.

1) *delete* operator works only for objects allocated using operator *new* (See <http://geeksforgeeks.org/?p=8539>). If the object is created using *new*, then we can do *delete this*, otherwise behavior is undefined.

```

class A
{
public:
    void fun()
    {
        delete this;
    }
};

int main()
{
    /* Following is Valid */
    A *ptr = new A;
    ptr->fun();
    ptr = NULL // make ptr NULL to make sure that things are not accessed

    /* And following is Invalid: Undefined Behavior */
    A a;
    a.fun();

    getchar();
    return 0;
}

```

2) Once *delete this* is done, any member of the deleted object should not be accessed after deletion.

```

#include<iostream>
using namespace std;

class A
{
    int x;
public:
    A() { x = 0; }
    void fun() {
        delete this;

        /* Invalid: Undefined Behavior */
        cout<<x;
    }
};

```

The best thing is to not do *delete this* at all.

Thanks to [Shekhu](#) for providing above details.

References:

<https://www.securecoding.cert.org/confluence/display/cplusplus/OOP05-CPP.+Aavoid+deleting+this>
http://en.wikipedia.org/wiki/This_%28computer_science%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.

47. 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.

48. Initialization of data members

In C++, class variables are initialized in the same order as they appear in the class declaration.

Consider the below code.

```

#include<iostream>

using namespace std;

class Test {
private:
    int y;
    int x;
public:
    Test() : x(10), y(x + 10) {}
    void print();
};

void Test::print()
{
    cout<<"x = "<<x<<" y = "<<y;
}

int main()
{
    Test t;
    t.print();
    getchar();
    return 0;
}

```

The program prints correct value of x, but some garbage value for y, because y is initialized before x as it appears before in the class declaration.

So one of the following two versions can be used to avoid the problem in above code.

```

// First: Change the order of declaration.
class Test {
private:
    int x;
    int y;
public:
    Test() : x(10), y(x + 10) {}
    void print();
};

// Second: Change the order of initialization.
class Test {
private:
    int y;
    int x;
public:
    Test() : x(y-10), y(20) {}
    void print();
};

```

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

49. RTTI (Run-time type information) in C++

In C++, **RTTI (Run-time type information)** is available only for the classes which have at least one virtual function.

For example, `dynamic_cast` uses RTTI and following program fails with error “*cannot dynamic_cast `b' (of type `class B*)' to type `class D*' (source type is not polymorphic)*” because there is no virtual function in the base class B.

```
#include<iostream>

using namespace std;

class B { };
class D: public B {};

int main()
{
    B *b = new D;
    D *d = dynamic_cast<D*>(b);
    if(d != NULL)
        cout<<"works";
    else
        cout<<"cannot cast B* to D*";
    getchar();
    return 0;
}
```

Adding a virtual function to the base class B makes it working.

```
#include<iostream>

using namespace std;

class B { virtual void fun() {} };
class D: public B { };

int main()
{
    B *b = new D;
    D *d = dynamic_cast<D*>(b);
    if(d != NULL)
        cout<<"works";
    else
        cout<<"cannot cast B* to D*";
    getchar();
    return 0;
}
```

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

50. Inheritance and friendship

In C++, friendship is not inherited. If a base class has a friend function, then the function doesn't become a friend of the derived class(es).

For example, following program prints error because *show()* which is a friend of base class *A* tries to access private data of derived class *B*.

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

class A
{
    protected:
        int x;
    public:
        A() { x = 0;}
        friend void show();
};

class B: public A
{
    public:
        B() : y (0) {}
    private:
        int y;
};

void show()
{
    B b;
    cout << "The default value of A::x = " << b.x;

    // Can't access private member declared in class 'B'
    cout << "The default value of B::y = " << b.y;
}

int main()
{
    show();
    getchar();
    return 0;
}
```

Thanks to [Venki](#) for the above code and explanation.

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

51. What is conversion constructor in C++?

In C++, if a class has a constructor which can be called with a single argument, then this constructor becomes conversion constructor because such a constructor allows automatic conversion to the class being constructed.

```

#include<iostream>

using namespace std;
class Test
{
    private:
        int x;
    public:
        Test(int i) {x = i;}
        void show() { cout<<" x = "<<x<<endl; }
};

int main()
{
    Test t(20);
    t.show();
    t = 30; // conversion constructor is called here.
    t.show();
    getchar();
    return 0;
}

```

The above program prints:

x = 20

x = 30

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

52. Virtual functions in derived classes

In C++, once a member function is declared as a virtual function in a base class, it becomes virtual in every class derived from that base class. In other words, it is not necessary to use the keyword virtual in the derived class while declaring redefined versions of the virtual base class function.

Source: <http://www.umsl.edu/~subramaniana/virtual2.html>

For example, the following program prints "C::fun() called" as B::fun() becomes virtual automatically.

```

#include<iostream>

using namespace std;

class A {
public:
    virtual void fun()
    { cout<<"\n A::fun() called ";}
};

class B: public A {
public:
    void fun()
    { cout<<"\n B::fun() called "; }
};

class C: public B {
public:
    void fun()
    { cout<<"\n C::fun() called "; }
};

int main()
{
    C c; // An object of class C
    B *b = &c; // A pointer of type B* pointing to c
    b->fun(); // this line prints "C::fun() called"
    getch();
    return 0;
}

```

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

53. Virtual Destructor

Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor.

Source: <https://www.securecoding.cert.org/confluence/display/cplusplus/OOP34-CPP.+Ensure+the+proper+destructor+is+called+for+polymorphic+objects>

For example, following program results in undefined behavior. Although the output of following program may be different on different compilers, when compiled using Dev-CPP, it prints following.

Constructing base

Constructing derived

Destructing base

```

// A program without virtual destructor causing undefined behavior
#include<iostream>

using namespace std;

class base {
public:
    base()
    { cout<<"Constructing base \n"; }
    ~base()
    { cout<<"Destructing base \n"; }
};

class derived: public base {
public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};

int main(void)
{
    derived *d = new derived();
    base *b = d;
    delete b;
    getchar();
    return 0;
}

```

Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called. For example, following program prints:

Constructing base

Constructing derived

Destructing derived

Destructing base

```

// A program with virtual destructor
#include<iostream>

using namespace std;

class base {
public:
    base()
    { cout<<"Constructing base \n"; }
    virtual ~base()
    { cout<<"Destructing base \n"; }
};

class derived: public base {
public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};

int main(void)
{
    derived *d = new derived();
    base *b = d;
    delete b;
    getchar();
    return 0;
}

```

As a guideline, any time you have a virtual function in a class, you should immediately add a virtual destructor (even if it does nothing). This way, you ensure against any surprises later.

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

54. Self assignment check in assignment operator

In C++, assignment operator should be overloaded with self assignment check.

For example, consider the following class *Array* and overloaded assignment operator function without self assignment check.

```

// A sample class
class Array {
private:
    int *ptr;
    int size;
public:
    Array& operator = (const Array &rhs);
    // constructors and other functions of class.....
};

// Overloaded assignment operator for class Array (without self
// assignment check)
Array& Array::operator = (const Array &rhs)
{
    // Deallocate old memory
    delete [] ptr;

    // allocate new space
    ptr = new int [rhs.size];

    // copy values
    size = rhs.size;
    for(int i = 0; i < size; i++)
        ptr[i] = rhs.ptr[i];

    return *this;
}

```

If we have an object say a1 of type Array and if we have a line like a1 = a1 somewhere, the program results in unpredictable behavior because there is no self assignment check in the above code. To avoid the above issue, self assignment check must be there while overloading assignment operator. For example, following code does self assignment check.

```

// Overloaded assignment operator for class Array (with self
// assignment check)
Array& Array::operator = (const Array &rhs)
{
    /* SELF ASSIGNMENT CHECK */
    if(this != &rhs)
    {
        // Deallocate old memory
        delete [] ptr;

        // allocate new space
        ptr = new int [rhs.size];

        // copy values
        size = rhs.size;
        for(int i = 0; i < size; i++)
            ptr[i] = rhs.ptr[i];
    }

    return *this;
}

```

References:

<http://www.cs.caltech.edu/courses/cs11/material/cpp/donnie/cpp-ops.html>

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

55. Rules for operator overloading

In C++, following are the general rules for operator overloading.

- 1) Only built-in operators can be overloaded. New operators can not be created.
- 2) **Arity of the operators** cannot be changed.
- 3) Precedence and associativity of the operators cannot be changed.
- 4) Overloaded operators cannot have default arguments except the function call operator () which can have default arguments.
- 5) Operators cannot be overloaded for built in types only. At least one operand must be used defined type.
- 6) Assignment (=), subscript ([]), function call ("()"), and member selection (->) operators must be defined as member functions
- 7) Except the operators specified in point 6, all other operators can be either member functions or a non member functions.
- 8) Some operators like (assignment)=, (address)& and comma (,) are by default overloaded.

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

56. 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.

57. 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.

58. 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.

59. 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

60. Type difference of character literals in C and C++

Every literal (constant) in C/C++ will have a type information associated with it.

In both C and C++, numeric literals (e.g. 10) will have *int* as their type. It means *sizeof(10)* and *sizeof(int)* will return same value.

However, character literals (e.g. 'V') will have different types, *sizeof('V')* returns different values in C and C++. In C, a character literal is treated as *int* type where as in C++, a character literal is treated as *char* type (*sizeof('V')* and *sizeof(char)* are same in C++ but not in C).

```
int main()
{
    printf("sizeof('V') = %d sizeof(char) = %d", sizeof('V'), sizeof(char));
    return 0;
}
```

Result of above program:

C result – *sizeof('V') = 4 sizeof(char) = 1*

C++ result – *sizeof('V') = 1 sizeof(char) = 1*

Such behaviour is required in C++ to support function overloading. An example will make it more clear. Predict the output of the following C++ program.

```
void foo(char c)
{
    printf("From foo: char");
}
void foo(int i)
{
    printf("From foo: int");
}

int main()
{
    foo('V');
    return 0;
}
```

The compiler must call

```
void foo(char);
```

since 'V' type is *char*.

Article contribution by Venki and Geeksforgeeks team.

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

61. Can a C++ class have an object of self type?

A class declaration can contain static object of self type, it can also have pointer to self type, but it cannot have a non-static object of self type.

For example, following program works fine.

```
// A class can have a static member of self type
#include<iostream>

using namespace std;

class Test {
    static Test self; // works fine

    /* other stuff in class*/
};

int main()
{
    Test t;
    getchar();
    return 0;
}
```

And following program also works fine.

```
// A class can have a pointer to self type
#include<iostream>

using namespace std;

class Test {
    Test * self; //works fine

    /* other stuff in class*/
};

int main()
{
    Test t;
    getchar();
    return 0;
}
```

But following program generates compilation error "*field 'self' has incomplete type*"

```
// A class cannot have non-static object(s) of self type.
#include<iostream>

using namespace std;

class Test {
    Test self; // Error

    /* other stuff in class*/
};

int main()
{
    Test t;
    getch();
    return 0;
}
```

If a non-static object is member then declaration of class is incomplete and compiler has no way to find out size of the objects of the class.

Static variables do not contribute to the size of objects. So no problem in calculating size with static variables of self type.

For a compiler, all pointers have a fixed size irrespective of the data type they are pointing to, so no problem with this also.

Thanks to Manish Jain and Venki for their contribution to this post. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

62. C/C++ Ternary Operator – Some Interesting Observations

Predict the output of following C++ program.

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

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

    return 0;
}
```

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

First character 1
Second character 49

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

Ternary Operator (C/C++):

A ternary operator has the following form,

$exp_1 ? exp_2 : exp_3$

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

Side Effects:

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

Return Type:

It is another interesting fact. The ternary operator has return type. The return type depends on exp_2 , and *convertibility* of exp_3 into exp_2 as per usual/overloaded conversion rules. If they are not convertible, the compiler throws an error. See the examples below,

The following program compiles without any error. The return type of ternary expression is expected to be *float* (as that of exp_2) and exp_3 (i.e. literal zero – *int* type) is implicitly convertible to *float*.

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

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

    return 0;
}
```

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

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

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

    return 0;
}
```

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

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

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

    return 0;
}
```

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

Best Practice:

It is the power of C++ type system that avoids such bugs. Make sure both the expressions exp_2 and exp_3 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.

63. 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'
    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.

64. 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.

65. 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.

66. Catching base and derived classes as exceptions

Exception Handling – catching base and derived classes as exceptions:

If both base and derived classes are caught as exceptions then catch block of derived class must appear before the base class.

If we put base class first then the derived class catch block will never be reached. For example, following C++ code prints *“Caught Base Exception”*

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

class Base {};
class Derived: public Base {};
int main()
{
    Derived d;
    // some other stuff
    try {
        // Some monitored code
        throw d;
    }
    catch(Base b) {
        cout<<"Caught Base Exception";
    }
    catch(Derived d) { //This catch block is NEVER executed
        cout<<"Caught Derived Exception";
    }
    getchar();
    return 0;
}
```

In the above C++ code, if we change the order of catch statements then both catch statements become reachable. Following is the modified program and it prints *“Caught Derived Exception”*

```

#include<iostream>
using namespace std;

class Base {};
class Derived: public Base {};
int main()
{
    Derived d;
    // some other stuff
    try {
        // Some monitored code
        throw d;
    }
    catch(Derived d) {
        cout<<"Caught Derived Exception";
    }
    catch(Base b) {
        cout<<"Caught Base Exception";
    }
    getchar();
    return 0;
}

```

In Java, catching a base class exception before derived is not allowed by the compiler itself. In C++, compiler might give warning about it, but compiles the code.

For example, following Java code fails in compilation with error message *“exception Derived has already been caught”*

```

//filename Main.java
class Base extends Exception {}
class Derived extends Base {}
public class Main {
    public static void main(String args[]) {
        try {
            throw new Derived();
        }
        catch(Base b) {}
        catch(Derived d) {}
    }
}

```

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

67. Stack Unwinding in C++

The process of removing function entries from function call stack at run time is called **Stack Unwinding**. Stack Unwinding is generally related to Exception Handling. In C++,

when an exception occurs, the function call stack is linearly searched for the exception handler, and all the entries before the function with exception handler are removed from the function call stack. So exception handling involves Stack Unwinding if exception is not handled in same function (where it is thrown).

For example, output of the following program is:

```
f3() Start
f2() Start
f1() Start
Caught Exception: 100
f3() End
```

```
#include <iostream>
```

```
using namespace std;
```

```
// A sample function f1() that throws an int exception
```

```
void f1() throw (int) {
    cout<<"\n f1() Start ";
    throw 100;
    cout<<"\n f1() End ";
}
```

```
// Another sample function f2() that calls f1()
```

```
void f2() throw (int) {
    cout<<"\n f2() Start ";
    f1();
    cout<<"\n f2() End ";
}
```

```
// Another sample function f3() that calls f2() and handles exception
```

```
void f3() {
    cout<<"\n f3() Start ";
    try {
        f2();
    }
    catch(int i) {
        cout<<"\n Caught Exception: "<<i;
    }
    cout<<"\n f3() End";
}
```

```
// A driver function to demonstrate Stack Unwinding process
```

```
int main() {
    f3();

    getchar();
    return 0;
}
```

In the above program, when f1() throws exception, its entry is removed from the function call stack (because it f1() doesn't contain exception handler for the thrown exception), then next entry in call stack is looked for exception handler. The next entry is f2(). Since f2() also doesn't have handler, its entry is also removed from function call stack. The next entry in function call stack is f3(). Since f3() contains exception handler, the catch

block inside f3() is executed, and finally the code after catch block is executed. Note that the following lines inside f1() and f2() are not executed at all.

```
//inside f1()
cout<<"\n f1() End ";

//inside f2()
cout<<"\n f2() End ";
```

On a side note, if there were some local class objects inside f1() and f2(), destructors for those local objects would have been called in Stack Unwinding process.

Stack Unwinding also happens in Java when exception is not handled in same function.

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

68. A Puzzle on C/C++ R-Value Expressions

What will be the output of following program?

```
#include <stdio.h>
int main()
{
    int i = 0xAA;
    ~i, printf("%X\n", i);

    return 0;
}
```

Output: 0xAA

No change in *i* value, the emphasis is on l-value and r-value expressions. The expression *~i* is an r-value, it has to be assigned to an l-value to retain the change.

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

69. Templates and Static variables in C++

Function templates and static variables:

Each instantiation of function template has its own copy of local static variables. For example, in the following program there are two instances: *void fun(int)* and *void*

fun(double)). So two copies of static variable *i* exist.

```
#include <iostream>

using namespace std;

template <typename T>
void fun(const T& x)
{
    static int i = 10;
    cout << ++i;
    return;
}

int main()
{
    fun<int>(1); // prints 11
    cout << endl;
    fun<int>(2); // prints 12
    cout << endl;
    fun<double>(1.1); // prints 11
    cout << endl;
    getchar();
    return 0;
}
```

Output of the above program is:

```
11
12
11
```

Class templates and static variables:

The rule for class templates is same as function templates

Each instantiation of class template has its own copy of member static variables. For example, in the following program there are two instances *Test* and *Test*. So two copies of static variable *count* exist.

```

#include <iostream>

using namespace std;

template <class T> class Test
{
private:
    T val;
public:
    static int count;
    Test()
    {
        count++;
    }
    // some other stuff in class
};

template<class T>
int Test<T>::count = 0;

int main()
{
    Test<int> a; // value of count for Test<int> is 1 now
    Test<int> b; // value of count for Test<int> is 2 now
    Test<double> c; // value of count for Test<double> is 1 now
    cout << Test<int>::count << endl; // prints 2
    cout << Test<double>::count << endl; //prints 1

    getchar();
    return 0;
}

```

Output of the above program is:

```

2
1

```

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

70. 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 need 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.

71. 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 linked 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](#)

72. Can namespaces be nested in C++?

In C++, namespaces can be nested, and resolution of namespace variables is hierarchical. For example, in the following code, namespace *inner* is created inside namespace *outer*, which is inside the global namespace. In the line “*int z = x*”, *x* refers to *outer::x*. If *x* would not have been in *outer* then this *x* would have referred to *x* in global namespace.

```
#include <iostream>

int x = 20;
namespace outer {
    int x = 10;
    namespace inner {
        int z = x; // this x refers to outer::x
    }
}

int main()
{
    std::cout<<outer::inner::z; //prints 10
    getchar();
    return 0;
}
```

Output of the above program is 10.

On a side note, unlike C++ namespaces, Java packages are not hierarchical.

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

73. 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.

74. 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.

75. 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 p2
    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.

76. Declare a C/C++ function returning pointer to array of integer pointers

Declare “a function with argument of *int** which returns pointer to an array of 4 integer pointers”.

At the first glance it may look complex, we can declare the required function with a series of decomposed statements.

1. We need, a function with argument *int **,

```
function(int *)
```

2. a function with argument *int **, returning pointer to

```
(*function(int *))
```

3. a function with argument *int **, returning pointer to array of 4

```
(*function(int *)) [4]
```

4. a function with argument *int **, returning pointer to array of 4 integer pointers

```
int *(*function(int *)) [4];
```

How can we ensure that the above declaration is correct? The following program can

cross checks our declaration,

```
#include<stdio.h>

// Symbolic size
#define SIZE_OF_ARRAY (4)

// pointer to array of (SIZE_OF_ARRAY) integers
typedef int *(*p_array_t)[SIZE_OF_ARRAY];

// Declaration : compiler should throw error
// if not matched with definition
int *(*function(int *arg))[4];

// Definition : 'function' returning pointer to an
// array of integer pointers
p_array_t function(int *arg)
{
    // array of integer pointers
    static int *arr[SIZE_OF_ARRAY] = {NULL};

    // return this
    p_array_t pRet = &arr;

    return pRet;
}

int main()
{
}
```

The macro `SIZE_OF_ARRAY` is used for symbolic representation of array size. `p_array_t` is typedefined as “pointer to an array of 4 integers”. If our declaration is wrong, the program throws an error at the *‘function’* definition.

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

77. 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.

78. Nested Classes in C++

A nested class is a class which is declared in another enclosing class. A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed.

For example, program 1 compiles without any error and program 2 fails in compilation.

Program 1

```
#include<iostream>

using namespace std;

/* start of Enclosing class declaration */
class Enclosing {
    int x;

    /* start of Nested class declaration */
    class Nested {
        int y;
        void NestedFun(Enclosing *e) {
            cout<<e->x; // works fine: nested class can access
                       // private members of Enclosing class
        }
    }; // declaration Nested class ends here
}; // declaration Enclosing class ends here

int main()
{
}
```

Program 2


```

#include<iostream>

using namespace std;

/* start of Enclosing class declaration */
class Enclosing {

    int x;

    /* start of Nested class declaration */
    class Nested {
        int y;
    }; // declaration Nested class ends here

    void EnclosingFun(Nested *n) {
        cout<<n->y; // Compiler Error: y is private in Nested
    }
}; // declaration Enclosing class ends here

int main()
{

}

```

References:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1905.pdf>

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

79. 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));

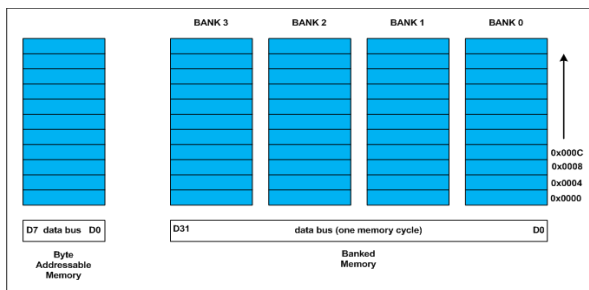
    return 0;
}

```

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:

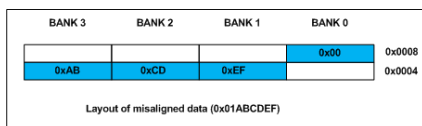
Every data type in C/C++ will have alignment requirement (infact it is mandated by processor architecture, not by language). A processor will have processing word length as that of data bus size. On a 32 bit machine, the processing word size will be 4 bytes.



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 $\text{sizeof(char)} + 1$ (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 $\text{sizeof(char)} + 7$ byte padding + $\text{sizeof(double)} + \text{sizeof(int)} = 1 + 7 + 8 + 4 = 20$ 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.

In order 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,

```
// Dereferencing 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.

In fact 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.

80. Order of operands for logical operators

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

In mathematics, logical AND, OR, EXOR, 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.

81. Function overloading and return type

In C++ and Java, functions can not be overloaded if they differ only in the return type.

For example, the following program C++ and Java programs fail in compilation.

C++ Program


```

#include<iostream>
int foo() {
    return 10;
}

char foo() { // compiler error; new declaration of foo()
    return 'a';
}

int main()
{
    char x = foo();
    getchar();
    return 0;
}

```

Java Program

```

// filename Main.java
public class Main {
    public int foo() {
        return 10;
    }
    public char foo() { // compiler error: foo() is already defined
        return 'a';
    }
    public static void main(String args[])
    {
    }
}

```

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

82. 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.

83. Some interesting facts about static member functions in C++

1) static member functions do not have [this pointer](#).

For example following program fails in compilation with error “*this’ is unavailable for static member functions*”

```

#include<iostream>
class Test {
    static Test * fun() {
        return this; // compiler error
    }
};

int main()
{
    getchar();
    return 0;
}

```

2) A static member function cannot be virtual (See [this G-Fact](#))

3) Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration.

For example, following program fails in compilation with error “*void Test::fun()’ and `static void Test::fun()’ cannot be overloaded*”

```
#include<iostream>
class Test {
    static void fun() {}
    void fun() {} // compiler error
};

int main()
{
    getchar();
    return 0;
}
```

4) A static member function can not be declared *const*, *volatile*, or *const volatile*. For example, following program fails in compilation with error “*static member function ‘static void Test::fun()’ cannot have ‘const’ method qualifier*”

```
#include<iostream>
class Test {
    static void fun() const { // compiler error
        return;
    }
};

int main()
{
    getchar();
    return 0;
}
```

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

References:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1905.pdf>

84. Structure vs class in C++

In C++, a structure is same as class except the following differences:

1) Members of a class are private by default and members of struct are public by default.

For example program 1 fails in compilation and program 2 works fine.

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

class Test {
    int x; // x is private
};
int main()
{
    Test t;
    t.x = 20; // compiler error because x is private
    getchar();
    return 0;
}
```

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

struct Test {
    int x; // x is public
};
int main()
{
    Test t;
    t.x = 20; // works fine because x is public
    getchar();
    return 0;
}
```

2) When deriving a struct from a class/struct, default access-specifier for a base class/struct is public. And when deriving a class, default access specifier is private. For example program 3 fails in compilation and program 4 works fine.

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

class Base {
public:
    int x;
};

class Derived : Base { }; // is equivalent to class Derived : private Base

int main()
{
    Derived d;
    d.x = 20; // compiler error because inheritance is private
    getchar();
    return 0;
}
```

```
// Program 4
#include <stdio.h>

class Base {
public:
    int x;
};

struct Derived : Base { }; // is equivalent to struct Derived : public
Base

int main()
{
    Derived d;
    d.x = 20; // works fine because inheritance is public
    getchar();
    return 0;
}
```

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

85. Is assignment operator inherited?

In C++, like other functions, assignment operator function is inherited in derived class.

For example, in the following program, base class assignment operator function can be accessed using the derived class object.

```
#include<iostream>

using namespace std;

class A {
public:
    A & operator= (A &a) {
        cout<<" base class assignment operator called ";
        return *this;
    }
};

class B: public A { };

int main()
{
    B a, b;
    a.A::operator=(b); //calling base class assignment operator function
                        // using derived class
    getchar();
    return 0;
}
```

Output: *base class assignment operator called*

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

86. Type of 'this' pointer in C++

In C++, *this pointer* is passed as a hidden argument to all non-static member function calls. The type of *this* depends upon function declaration. If the member function of a class *X* is declared *const*, the type of *this* is *const X** (see code 1 below), if the member function is declared *volatile*, the type of *this* is *volatile X** (see code 2 below), and if the member function is declared *const volatile*, the type of *this* is *const volatile X** (see code 3 below).

Code 1

```
#include<iostream>
class X {
    void fun() const {
        // this is passed as hidden argument to fun().
        // Type of this is const X*
    }
};
```

Code 2

```
#include<iostream>
class X {
    void fun() volatile {
        // this is passed as hidden argument to fun().
        // Type of this is volatile X*
    }
};
```

Code 3

```
#include<iostream>
class X {
    void fun() const volatile {
        // this is passed as hidden argument to fun().
        // Type of this is const volatile X*
    }
};
```

References:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1905.pdf>

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

87. Copy constructor vs assignment operator in C++

Difficulty Level: Rookie

Consider the following C++ program.

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

using namespace std;

class Test
{
public:
    Test() {}
    Test(const Test &t)
    {
        cout<<"Copy constructor called "<<endl;
    }
    Test& operator = (const Test &t)
    {
        cout<<"Assignment operator called "<<endl;
    }
};

int main()
{
    Test t1, t2;
    t2 = t1;
    Test t3 = t1;
    getchar();
    return 0;
}
```

Output:

Assignment operator called

Copy constructor called

Copy constructor is called when a new object is created from an existing object, as a copy of the existing object (see [this](#) G-Fact). And assignment operator is called when an already initialized object is assigned a new value from another existing object.

```
t2 = t1; // calls assignment operator, same as "t2.operator=(t1);"
Test t3 = t1; // calls copy constructor, same as "Test t3(t1);"
```

References:

http://en.wikipedia.org/wiki/Copy_constructor

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

88. 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++, |
    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.

89. 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.

90. 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.

91. 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 an 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>

92. 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;
}
```

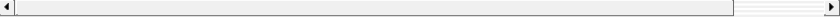
Program 2

```
#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
    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.

93. 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.

94. 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.

95. fork() and Binary Tree

Given a program on **fork()** system call.

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    fork();
    fork() && fork() || fork();
    fork();

    printf("forked\n");
    return 0;
}
```

How many processes will be spawned after executing the above program?

A *fork()* system call spawn processes as leaves of growing binary tree. If we call *fork()* twice, it will spawn $2^2 = 4$ processes. All these 4 processes forms the leaf children of binary tree. In general if we are level *l*, and *fork()* called unconditionally, we will have 2^l processes at level **(l+1)**. It is equivalent to number of maximum child nodes in a binary tree at level **(l+1)**.

As another example, assume that we have invoked *fork()* call 3 times unconditionally. We can represent the spawned process using a full binary tree with 3 levels. At level 3, we will have $2^3 = 8$ child nodes, which corresponds to number of processes running.

A note on C/C++ logical operators:

The logical operator && has more precedence than ||, and have left to right associativity. After executing left operand, the final result will be estimated and execution of right operand depends on outcome of left operand as well as type of operation.

In case of AND (&&), after evaluation of left operand, right operand will be evaluated only if left operand evaluates to **non-zero**. In case of OR (||), after evaluation of left operand, right operand will be evaluated only if left operand evaluates to **zero**.

Return value of fork():

The man pages of fork() cites the following excerpt on return value,

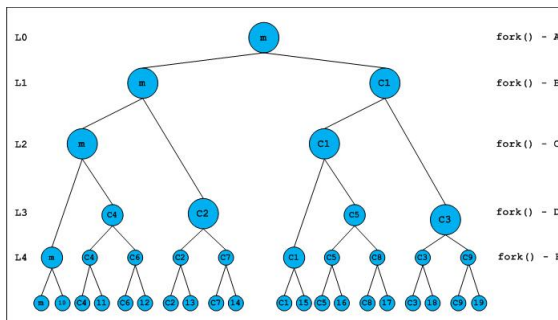
“On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.”

A PID is like handle of process and represented as *unsigned int*. We can conclude, the fork() will return a non-zero in parent and zero in child. Let us analyse the program. For easy notation, label each fork() as shown below,

```
#include <stdio.h>
int main()
{
    fork(); /* A */
    ( fork() /* B */ &&
      fork() /* C */ ) || /* B and C are grouped according to precedence */
    fork(); /* D */
    fork(); /* E */

    printf("forked\n");
    return 0;
}
```

The following diagram provides pictorial representation of fork-ing new processes. All newly created processes are propagated on right side of tree, and parents are propagated on left side of tree, in consecutive levels.



The first two fork() calls are called unconditionally.

At level 0, we have only main process. The main (m in diagram) will create child C1 and both will continue execution. The children are numbered in increasing order of their creation.

At level 1, we have m and C1 running, and ready to execute fork() – B. (Note that B, C and D named as operands of && and || operators). The initial expression B will be executed in every children and parent process running at this level.

At level 2, due to `fork()` – B executed by m and C1, we have m and C1 as parents and, C2 and C3 as children.

The return value of `fork()` – B is non-zero in parent, and zero in child. Since the first operator is `&&`, because of zero return value, the children C2 and C3 **will not** execute next expression (`fork()` – C). Parents processes m and C1 will continue with `fork()` – C. The children C2 and C3 will directly execute `fork()` – D, to evaluate value of logical OR operation.

At level 3, we have m, C1, C2, C3 as running processes and C4, C5 as children. The expression is now simplified to `((B && C) || D)`, and at this point the value of `(B && C)` is obvious. In parents it is non-zero and in children it is zero. Hence, the parents aware of outcome of overall `B && C || D`, will skip execution of `fork()` – D. Since, in the children `(B && C)` evaluated to zero, they will execute `fork()` – D. We should note that children C2 and C3 created at level 2, will also run `fork()` – D as mentioned above.

At level 4, we will have m, C1, C2, C3, C4, C5 as running processes and C6, C7, C8 and C9 as child processes. All these processes unconditionally execute `fork()` – E, and spawns one child.

At level 5, we will have 20 processes running. The program (on Ubuntu Maverick, GCC 4.4.5) printed “forked” 20 times. Once by root parent (main) and rest by children. Overall there will be 19 processes spawned.

A note on order of evaluation:

The evaluation order of expressions in binary operators is unspecified. For details read the post [Evaluation order of operands](#). However, the logical operators are an exception. They are guaranteed to evaluate from left to right.

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

96. Functions that cannot be overloaded in C++

In C++, following function declarations **cannot** be overloaded.

1) Function declarations that differ only in the return type. For example, the following program fails in compilation.

```
#include<iostream>
int foo() {
    return 10;
}

char foo() {
    return 'a';
}

int main()
{
    char x = foo();
    getchar();
    return 0;
}
```

2) Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration. For example, following program fails in compilation.

```
#include<iostream>
class Test {
    static void fun(int i) {}
    void fun(int i) {}
};

int main()
{
    Test t;
    getchar();
    return 0;
}
```

3) Parameter declarations that differ only in a pointer * versus an array [] are equivalent. That is, the array declaration is adjusted to become a pointer declaration. Only the second and subsequent array dimensions are significant in parameter types. For example, following two function declarations are equivalent.

```
int fun(int *ptr);
int fun(int ptr[]); // redeclaration of fun(int *ptr)
```

4) Parameter declarations that differ only in that one is a function type and the other is a pointer to the same function type are equivalent.

```
void h(int ());
void h(int (*)( )); // redeclaration of h(int())
```

5) Parameter declarations that differ only in the presence or absence of const and/or volatile are equivalent. That is, the const and volatile type-specifiers for each parameter type are ignored when determining which function is being declared, defined, or called. For example, following program fails in compilation with error “redefinition of ‘int f(int)’ “

Example:

```

#include<iostream>
#include<stdio.h>

using namespace std;

int f ( int x) {
    return x+10;
}

int f ( const int x) {
    return x+10;
}

int main() {
    getchar();
    return 0;
}

```

Only the const and volatile type-specifiers at the outermost level of the parameter type specification are ignored in this fashion; const and volatile type-specifiers buried within a parameter type specification are significant and can be used to distinguish overloaded function declarations. In particular, for any type T, “pointer to T,” “pointer to const T,” and “pointer to volatile T” are considered distinct parameter types, as are “reference to T,” “reference to const T,” and “reference to volatile T.” For example, see the example in [this comment](#) posted by Venki.

6) Two parameter declarations that differ only in their default arguments are equivalent. For example, following program fails in compilation with error “*redefinition of ‘int f(int, int)’*”

```

#include<iostream>
#include<stdio.h>

using namespace std;

int f ( int x, int y) {
    return x+10;
}

int f ( int x, int y = 10) {
    return x+y;
}

int main() {
    getchar();
    return 0;
}

```

References:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1905.pdf>

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

97. C++ default constructor | Built-in types

Predict the output of following program?

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

int main() {
    cout << int() << endl;
    return 0;
}
```

A constructor without any arguments or with default values for every argument, is treated as *default constructor*. It will be called by the compiler when in need (precisely code will be generated for default constructor based on need).

C++ allows even built-in type (primitive types) to have default constructors. The function style cast `int()` is analogous to casting 0 to required type. The program prints 0 on console.

The initial content of the article triggered many discussions, given below is consolidation.

It is worth to be cognizant of reference vs. value semantics in C++ and the concept of Plain Old Data types. From Wiki, primitive types and POD types have no user-defined copy assignment operator, no user-defined destructor, and no non-static data members that are not themselves PODs. Moreover, a POD class must be an aggregate, meaning it has no user-declared constructors, no private nor protected non-static data, no base classes and no virtual functions.

An excerpt (from a mail note) from the creator of C++, "I think you mix up 'actual constructor calls' with conceptually having a constructor. Built-in types are considered to have constructors".

The code snippet above mentioned `int()` is considered to be conceptually having constructor. However, there will not be any code generated to make an *explicit constructor* call. But when we observe assembly output, code will be generated to initialize the identifier using value semantics. For more details refer section 8.5 of [this document](#).

Thanks to [Prasoon Saurav](#) for initiating the discussion, providing various references and correcting lacuna in my understanding.

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

References:

1. The C++ Programming Language, 3e.
2. Latest C++ standard – working draft section 8.5.

98. Calculate Logn in one line

Write a one line C function that calculates and returns $\lfloor \log_2(n) \rfloor$. 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 version of the problem. Write a one line function Logn(n,r) which returns $\lfloor \log_r(n) \rfloor$. 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.

99. Advanced C++ | Conversion Operators

In C++, the programmer abstracts real world objects using classes as concrete types. Sometimes it is required to convert one concrete type to another concrete type or primitive type implicitly. Conversion operators play smart role in such situations.

For example consider the following class

```
#include <iostream>
#include <cmath>

using namespace std;

class Complex
{
private:
    double real;
    double imag;

public:
    // Default constructor
    Complex(double r = 0.0, double i = 0.0) : real(r), imag(i)
    {}

    // magnitude : usual function style
    double mag()
    {
        return getMag();
    }

    // magnitude : conversion operator
    operator double ()
    {
        return getMag();
    }

private:
    // class helper to get magnitude
    double getMag()
    {
        return sqrt(real * real + imag * imag);
    }
};

int main()
{
    // a Complex object
    Complex com(3.0, 4.0);

    // print magnitude
    cout << com.mag() << endl;
    // same can be done like this
    cout << com << endl;
}
```

We are printing the magnitude of Complex object in two different ways.

Note that usage of such smart (over smart ?) techniques are discouraged. The compiler

will have more control in calling an appropriate function based on type, rather than what the programmer expects. It will be good practice to use other techniques like class/object specific member function (or making use of C++ Variant class) to perform such conversions. At some places, for example in making compatible calls with existing C library, these are unavoidable.

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

100. 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.

101. 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>

102. Multidimensional Pointer Arithmetic in C/C++

In C/C++, arrays and pointers have similar semantics, except on type information.

As an example, given a 3D array

```
int buffer[5][7][6];
```

An element at location `[2][1][2]` can be accessed as `buffer[2][1][2]` or `*(*(*(buffer + 2) + 1) + 2)`.

Observe the following declaration

```
T *p; // p is a pointer to an object of type T
```

When a pointer *p* is pointing to an object of type T, the expression **p* is of type T. For example *buffer* is of type array of 5 two dimensional arrays. The type of the expression **buffer* is “array of arrays (i.e. two dimensional array)”.

Based on the above concept translating the expression `*(*(*(buffer + 2) + 1) + 2)` step-by-step makes it more clear.

1. *buffer* – An array of 5 two dimensional arrays, i.e. its type is “array of 5 two dimensional arrays”.
2. *buffer + 2* – displacement for 3rd element in the array of 5 two dimensional arrays.
3. **(buffer + 2)* – dereferencing, i.e. its type is now two dimensional array.
4. **(buffer + 2) + 1* – displacement to access 2nd element in the array of 7 one dimensional arrays.
5. **(*(buffer + 2) + 1)* – dereferencing (accessing), now the type of expression “**(*(buffer + 2) + 1)*” is an array of integers.
6. **(*(buffer + 2) + 1) + 2* – displacement to get element at 3rd position in the single dimension array of integers.
7. **(*(*(buffer + 2) + 1) + 2)* – accessing the element at 3rd position (the overall expression type is *int* now).

The compiler calculates an “offset” to access array element. The “offset” is calculated based on dimensions of the array. In the above case, $offset = 2 * (7 * 6) + 1 * (6) + 2$. Those in blue colour are dimensions, *note that the higher dimension is not used in offset calculation*. During compile time the compiler is aware of dimensions of array. Using offset we can access the element as shown below,

```
element_data = *( (int *)buffer + offset );
```

It is not always possible to declare dimensions of array at compile time. Sometimes we need to interpret a buffer as multidimensional array object. For instance, when we are processing 3D image whose dimensions are determined at run-time, usual array subscript rules can't be used. It is due to lack of fixed dimensions during compile time. Consider the following example,

```
int *base;
```

Where *base* is pointing large image buffer that represents 3D image of dimension *l x b x h* where *l*, *b* and *h* are variables. If we want to access an element at location (2, 3, 4) we need to calculate offset of the element as

$offset = 2 * (b \times h) + 3 * (h) + 4$ and the element located at *base + offset*.

Generalizing further, given start address (say *base*) of an array of size [***l x b x h***] dimensions, we can access the element at an arbitrary location (***a, b, c***) in the following way,

```
data = *(base + a * (b x h) + b * (h) + c); // Note that we haven't used the higher dimension l.
```

The same concept can be applied to any number of dimensions. We don't need the higher dimension to calculate offset of any element in the multidimensional array. *It is the reason behind omitting the higher dimension when we pass multidimensional arrays to functions.* The higher dimension is needed only when the programmer iterating over limited number of elements of higher dimension.

A C/C++ puzzle, predict the output of following program

```
int main()
{
    char arr[5][7][6];
    char (*p)[5][7][6] = &arr;

    /* Hint: &arr - is of type const pointer to an array of
       5 two dimensional arrays of size [7][6] */

    printf("%d\n", (&arr + 1) - &arr);
    printf("%d\n", (char *)&arr + 1 - (char *)&arr);
    printf("%d\n", (unsigned)(arr + 1) - (unsigned)arr);
    printf("%d\n", (unsigned)(p + 1) - (unsigned)p);

    return 0;
}
```

Output:

1

210

42

210

Thanks to [student](#) for pointing an error.

— [Venki](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

103. When do we pass arguments by reference or pointer?

In C++, variables are passed by reference due to following reasons:

1) To modify local variables of the caller function: A reference (or pointer) allows called function to modify a local variable of the caller function. For example, consider the following example program where *fun()* is able to modify local variable *x* of *main()*.

```
void fun(int &x) {
    x = 20;
}

int main() {
    int x = 10;
    fun(x);
    cout<<"New value of x is "<<x;
    return 0;
}
```

Output:

New value of x is 20

2) For passing large sized arguments: If an argument is large, passing by reference (or pointer) is more efficient because only an address is really passed, not the entire object. For example, let us consider the following *Employee* class and a function *printEmpDetails()* that prints Employee details.

```
class Employee {
private:
    string name;
    string desig;

    // More attributes and operations
};

void printEmpDetails(Employee emp) {
    cout<<emp.getName();
    cout<<emp.getDesig();

    // Print more attributes
}
```

The problem with above code is: every time *printEmpDetails()* is called, a new Employee object is constructed that involves creating a copy of all data members. So a better implementation would be to pass Employee as a reference.

```
void printEmpDetails(const Employee &emp) {
    cout<<emp.getName();
    cout<<emp.getDesig();

    // Print more attributes
}
```

This point is valid only for struct and class variables as we don't get any efficiency advantage for basic types like int, char.. etc.

3) To avoid Object Slicing: If we pass an object of subclass to a function that expects an object of superclass then the passed object is **sliced** if it is pass by value. For example, consider the following program, it prints “This is Pet Class”.

```
#include <iostream>
#include<string>

using namespace std;

class Pet {
public:
    virtual string getDescription() const {
        return "This is Pet class";
    }
};

class Dog : public Pet {
public:
    virtual string getDescription() const {
        return "This is Dog class";
    }
};

void describe(Pet p) { // Slices the derived class object
    cout<<p.getDescription()<<endl;
}

int main() {
    Dog d;
    describe(d);
    return 0;
}
```

Output:

This is Pet Class

If we use pass by reference in the above program then it correctly prints “This is Dog Class”. See the following modified program.

```

#include <iostream>
#include<string>

using namespace std;

class Pet {
public:
    virtual string getDescription() const {
        return "This is Pet class";
    }
};

class Dog : public Pet {
public:
    virtual string getDescription() const {
        return "This is Dog class";
    }
};

void describe(const Pet &p) { // Doesn't slice the derived class object
    cout<<p.getDescription()<<endl;
}

int main() {
    Dog d;
    describe(d);
    return 0;
}

```

Output:

This is Dog Class

This point is also not valid for basic data types like int, char, .. etc.

4) To achieve Run Time Polymorphism in a function

We can make a function polymorphic by passing objects as reference (or pointer) to it. For example, in the following program, print() receives a reference to the base class object. print() calls the base class function show() if base class object is passed, and derived class function show() if derived class object is passed.

```

#include<iostream>
using namespace std;

class base {
public:
    virtual void show() { // Note the virtual keyword here
        cout<<"In base \n";
    }
};

class derived: public base {
public:
    void show() {
        cout<<"In derived \n";
    }
};

// Since we pass b as reference, we achieve run time polymorphism here
void print(base &b) {
    b.show();
}

int main(void) {
    base b;
    derived d;
    print(b);
    print(d);
    return 0;
}

```

Output:

In base

In derived

Thanks to [Venki](#) for adding this point.

As a side note, it is a recommended practice to make reference arguments const if they are being passed by reference only due to reason no. 2 or 3 mentioned above. This is recommended to avoid unexpected modifications to the objects.

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

104. When are Constructors Called?

When are the constructors called for different types of objects like global, local, static local, dynamic?

1) Global objects: For a global object, constructor is called before main() is called. For

example, see the following program and output:

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

class Test
{
public:
    Test();
};

Test::Test() {
    cout << "Constructor Called \n";
}

Test t1;

int main() {
    cout << "main() started\n";
    return 0;
}
/* OUTPUT:
    Constructor Called
    main() started
*/
```

2) Function or Block Scope (automatic variables and constants) For a non-static local object, constructor is called when execution reaches point where object is declared. For example, see the following program and output:

```
using namespace std;

class Test
{
public:
    Test();
};

Test::Test() {
    cout << "Constructor Called \n";
}

void fun() {
    Test t1;
}

int main() {
    cout << "Before fun() called\n";
    fun();
    cout << "After fun() called\n";
    return 0;
}
/* OUTPUT:
    Before fun() called
    Constructor Called
    After fun() called
*/
```

For a local static object, the **first** time (and only the first time) execution reaches point where object is declared. For example, output of the following program is:

```

#include<iostream>
using namespace std;

class Test
{
public:
    Test();
};

Test::Test() {
    cout << "Constructor Called \n";
}

void fun() {
    static Test t1;
}

int main() {
    cout << "Before fun() called\n";
    fun();
    cout << "After fun() called\n";
    fun(); //constructor is not called this time.
    return 0;
}
/* OUTPUT
    Before fun() called
    Constructor Called
    After fun() called
*/

```

3) Class Scope: When an object is created, compiler makes sure that constructors for all of its subobjects (its member and inherited objects) are called. If members have default constructors or constructor without parameter then these constructors are called automatically, otherwise parameterized constructors can be called using [Initializer List](#). For example, see PROGRAM 1 and PROGRAM 2 and their output.

```
// PROGRAM 1: Constructor without any parameter
```

```
#include<iostream>
```

```
using namespace std;
```

```
class A
```

```
{
```

```
public:
```

```
    A();
```

```
};
```

```
A::A() {
```

```
    cout << "A's Constructor Called \n";
```

```
}
```

```
class B
```

```
{
```

```
    A t1;
```

```
public:
```

```
    B();
```

```
};
```

```
B::B() {
```

```
    cout << "B's Constructor Called \n";
```

```
}
```

```
int main() {
```

```
    B b;
```

```
    return 0;
```

```
}
```

```
/* OUTPUT:
```

```
    A's Constructor Called
```

```
    B's Constructor Called
```

```
*/
```

```

// PROGRAM 2: Constructor with parameter (using initializer list)
#include <iostream>
using namespace std;

class A
{
public:
    int i;
    A(int );
};

A::A(int arg)
{
    i = arg;
    cout << "A's Constructor called: Value of i: " << i << endl;
}

// Class B contains object of A
class B
{
    A a;
public:
    B(int );
};

B::B(int x):a(x)
{
    cout << "B's Constructor called";
}

int main()
{
    B obj(10);
    return 0;
}

/* OUTPUT
    A's Constructor called: Value of i: 10
    B's Constructor called
*/

```

4) Dynamic objects: For a dynamically allocated object, constructor is invoked by new operator. For example, see the following program and output.

```

#include<iostream>

using namespace std;

class Test
{
public:
    Test();
};

Test::Test() {
    cout << "Constructor Called \n";
}

int main()
{
    cout << "Before new called\n";
    Test *t1 = new Test;
    cout << "After new called\n";
    return 0;
}
/* OUTPUT
    Before new called
    Constructor Called
    After new called
*/

```

References:

http://web.cs.wpi.edu/~cs2303/c10/Protected/Lectures-C10/Week5_MoreClasses.ppt

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

105. When do we use Initializer List in C++?

Initializer List is used to initialize data members of a class. The list of members to be initialized is indicated with constructor as a comma separated list followed by a colon. Following is an example that uses initializer list to initialize x and y of Point class.

```

#include<iostream>
using namespace std;

class Point {
private:
    int x;
    int y;
public:
    Point(int i = 0, int j = 0):x(i), y(j) {}
    /* The above use of Initializer list is optional as the
       constructor can also be written as:
       Point(int i = 0, int j = 0) {
           x = i;
           y = j;
       }
    */

    int getX() const {return x;}
    int getY() const {return y;}
};

int main() {
    Point t1(10, 15);
    cout<<"x = "<<t1.getX()<<" , ";
    cout<<"y = "<<t1.getY();
    return 0;
}

/* OUTPUT:
   x = 10, y = 15
*/

```

The above code is just an example for syntax of Initializer list. In the above code, x and y can also be easily initialed inside the constructor. But there are situations where initialization of data members inside constructor doesn't work and Initializer List must be used. Following are such cases:

1) For initialization of non-static const data members:

const data members must be initialized using Initializer List. In the following example, "t" is a const data member of Test class and is initialized using Initializer List.

```

#include<iostream>
using namespace std;

class Test {
    const int t;
public:
    Test(int t):t(t) {} //Initializer list must be used
    int getT() { return t; }
};

int main() {
    Test t1(10);
    cout<<t1.getT();
    return 0;
}

/* OUTPUT:
10
*/

```

2) For initialization of reference members:

Reference members must be initialized using Initializer List. In the following example, “t” is a reference member of Test class and is initialized using Initializer List.

```

// Initialization of reference data members
#include<iostream>
using namespace std;

class Test {
    int &t;
public:
    Test(int &t):t(t) {} //Initializer list must be used
    int getT() { return t; }
};

int main() {
    int x = 20;
    Test t1(x);
    cout<<t1.getT()<<endl;
    x = 30;
    cout<<t1.getT()<<endl;
    return 0;
}

/* OUTPUT:
20
30
*/

```

3) For initialization of member objects which do not have default constructor:

In the following example, an object “a” of class “A” is data member of class “B”, and “A” doesn’t have default constructor. Initializer List must be used to initialize “a”.

```

#include <iostream>
using namespace std;

class A {
    int i;
public:
    A(int );
};

A::A(int arg) {
    i = arg;
    cout << "A's Constructor called: Value of i: " << i << endl;
}

// Class B contains object of A
class B {
    A a;
public:
    B(int );
};

B::B(int x):a(x) { //Initializer list must be used
    cout << "B's Constructor called";
}

int main() {
    B obj(10);
    return 0;
}

/* OUTPUT:
    A's Constructor called: Value of i: 10
    B's Constructor called
*/

```

If class A had both default and parameterized constructors, then Initializer List is not must if we want to initialize “a” using default constructor, but it is must to initialize “a” using parameterized constructor.

4) For initialization of base class members : Like point 3, parameterized constructor of base class can only be called using Initializer List.


```

#include <iostream>
using namespace std;

class A {
    int i;
public:
    A(int );
};

A::A(int arg) {
    i = arg;
    cout << "A's Constructor called: Value of i: " << i << endl;
}

// Class B is derived from A
class B: A {
public:
    B(int );
};

B::B(int x):A(x) { //Initializer list must be used
    cout << "B's Constructor called";
}

int main() {
    B obj(10);
    return 0;
}

```

5) When constructor's parameter name is same as data member

If constructor's parameter name is same as data member name then the data member must be initialized either using **this pointer** or Initializer List. In the following example, both member name and parameter name for A() is "i".

```

#include <iostream>
using namespace std;

class A {
    int i;
public:
    A(int );
    int getI() const { return i; }
};

A::A(int i):i(i) { } // Either Initializer list or this pointer must be used
/* The above constructor can also be written as
A::A(int i) {
    this->i = i;
}
*/

int main() {
    A a(10);
    cout<<a.getI();
    return 0;
}
/* OUTPUT:
10
*/

```

6) For Performance reasons:

It is better to initialize all class variables in Initializer List instead of assigning values inside body. Consider the following example:

```
// Without Initializer List
class MyClass {
    Type variable;
public:
    MyClass(Type a) { // Assume that Type is an already
                      // declared class and it has appropriate
                      // constructors and operators
        variable = a;
    }
};
```

Here compiler follows following steps to create an object of type MyClass

1. Type's constructor is called first for "a".
2. The assignment operator of "Type" is called inside body of MyClass() constructor to assign

```
variable = a;
```

3. And then finally destructor of "Type" is called for "a" since it goes out of scope.

Now consider the same code with MyClass() constructor with Initializer List

```
// With Initializer List
class MyClass {
    Type variable;
public:
    MyClass(Type a):variable(a) { // Assume that Type is an already
                                  // declared class and it has appropriate
                                  // constructors and operators
    }
};
```

With the Initializer List, following steps are followed by compiler:

1. Copy constructor of "Type" class is called to initialize : variable(a). The arguments in initializer list are used to copy construct "variable" directly.
2. Destructor of "Type" is called for "a" since it goes out of scope.

As we can see from this example if we use assignment inside constructor body there are three function calls: constructor + destructor + one addition assignment operator call.

And if we use Initializer List there are only two function calls: copy constructor + destructor call. See [this](#) post for a running example on this point.

This assignment penalty will be much more in "real" applications where there will be many such variables. Thanks to *ptr* for adding this point.

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

106. 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.

107. C++ Internals | Default Constructors | Set 1

A constructor without any arguments or with default value for every argument, is said to be **default constructor**. What is the significance of default constructor? Will the code be generated for every default constructor? Will there be any code inserted by compiler to the user implemented default constructor behind the scenes?

The compiler will implicitly *declare* default constructor if not provided by programmer, will *define* it when in need. Compiler defined default constructor is required to do certain initialization of class internals. It will not touch the data members or plain old data types (aggregates like array, structures, etc...). However, the compiler generates code for default constructor based on situation.

Consider a class derived from another class with default constructor, or a class containing another class object with default constructor. The compiler needs to insert code to call the default constructors of base class/embedded object.

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

class Base
{
public:
    // compiler "declares" constructor
};

class A
{
public:
    // User defined constructor
    A()
    {
        cout << "A Constructor" << endl;
    }

    // uninitialized
    int size;
};

class B : public A
{
    // compiler defines default constructor of B, and
    // inserts stub to call A constructor

    // compiler won't initialize any data of A
};

class C : public A
{
public:
    C()
    {
        // User defined default constructor of C
        // Compiler inserts stub to call A's constructor
        cout << "B Constructor" << endl;

        // compiler won't initialize any data of A
    }
};

class D
{
public:
    D()
    {
        // User defined default constructor of D
        // a - constructor to be called, compiler inserts
        // stub to call A constructor
        cout << "D Constructor" << endl;

        // compiler won't initialize any data of 'a'
    }
};
```

```

private:
    A a;
};

int main()
{
    Base base;

    B b;
    C c;
    D d;

    return 0;
}

```

There are different scenarios in which compiler needs to insert code to ensure some necessary initialization as per language requirement. We will have them in upcoming posts. Our objective is to be aware of C++ internals, not to use them incorrectly.

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

108. Hiding of all overloaded methods with same name in base class

In C++, if a derived class redefines base class member method then all the base class methods with same name become hidden in derived class.

For example, the following program doesn't compile. In the following program, Derived redefines Base's method fun() and this makes fun(int i) hidden.

```

#include<iostream>

using namespace std;

class Base
{
public:
    int fun()
    {
        cout<<"Base::fun() called";
    }
    int fun(int i)
    {
        cout<<"Base::fun(int i) called";
    }
};

class Derived: public Base
{
public:
    int fun()
    {
        cout<<"Derived::fun() called";
    }
};

int main()
{
    Derived d;
    d.fun(5); // Compiler Error
    return 0;
}

```

Even if the signature of the derived class method is different, all the overloaded methods in base class become hidden. For example, in the following program, `Derived::fun(char)` makes both `Base::fun()` and `Base::fun(int)` hidden.

```

#include<iostream>

using namespace std;

class Base
{
public:
    int fun()
    {
        cout<<"Base::fun() called";
    }
    int fun(int i)
    {
        cout<<"Base::fun(int i) called";
    }
};

class Derived: public Base
{
public:
    int fun(char c) // Makes Base::fun() and Base::fun(int ) hidden
    {
        cout<<"Derived::fun(char c) called";
    }
};

int main()
{
    Derived d;
    d.fun(); // Compiler Error
    return 0;
}

```

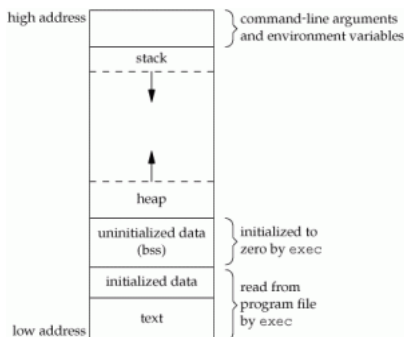
Note that the above facts are true for both static and nonstatic methods.

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

109. 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>

110. Templates and Default Arguments

Default parameters for templates in C++:

Like function default arguments, templates can also have default arguments. For

example, in the following program, the second parameter U has the default value as char.

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

template<class T, class U = char> class A
{
public:
    T x;
    U y;
};

int main()
{
    A<char> a;
    A<int, int> b;
    cout<<sizeof(a)<<endl;
    cout<<sizeof(b)<<endl;
    return 0;
}
```

Output: (char takes 1 byte and int takes 4 bytes)

2

8

Also, similar to default function arguments, if one template parameter has a default argument, then all template parameters following it must also have default arguments. For example, the compiler will not allow the following program:

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

template<class T = char, class U, class V = int> class A // Error
{
    // members of A
};

int main()
{
}
```

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

111. 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.

112. 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 iss

    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 to
                         the compiler to "const int *" */

    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%20int%20ptr
// 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 */

    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.

113. Why is the size of an empty class not zero in C++?

Predict the output of following program?

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

class Empty {};

int main()
{
    cout << sizeof(Empty);
    return 0;
}
```

Output:

```
1
```

Size of an empty class is not zero. It is 1 byte generally. It is nonzero to ensure that the two different objects will have different addresses. See the following example.

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

class Empty { };

int main()
{
    Empty a, b;

    if (&a == &b)
        cout << "impossible " << endl;
    else
        cout << "Fine " << endl;

    return 0;
}
```

Output:

```
Fine
```

For the same reason (different objects should have different addresses), “new” always returns pointers to distinct objects. See the following example.

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

class Empty { };

int main()
{
    Empty* p1 = new Empty;
    Empty* p2 = new Empty;

    if (p1 == p2)
        cout << "impossible " << endl;
    else
        cout << "Fine " << endl;

    return 0;
}
```

Output:

```
Fine
```

Now guess the output of following program (This is tricky)

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

class Empty { };

class Derived: Empty { int a; };

int main()
{
    cout << sizeof(Derived);
    return 0;
}
```

Output (with GCC compiler. See [this](#)):

```
4
```

Note that the output is not greater than 4. There is an interesting rule that says that an empty base class need not be represented by a separate byte. So compilers are free to make optimization in case of empty base classes. As an exercise, try the following program on your compiler.

```
// Thanks to Venki for suggesting this code.
#include <iostream>
using namespace std;

class Empty
{
};

class Derived1 : public Empty
{
};

class Derived2 : virtual public Empty
{
};

class Derived3 : public Empty
{
    char c;
};

class Derived4 : virtual public Empty
{
    char c;
};

class Dummy
{
    char c;
};

int main()
{
    cout << "sizeof(Empty) " << sizeof(Empty) << endl;
    cout << "sizeof(Derived1) " << sizeof(Derived1) << endl;
    cout << "sizeof(Derived2) " << sizeof(Derived2) << endl;
    cout << "sizeof(Derived3) " << sizeof(Derived3) << endl;
    cout << "sizeof(Derived4) " << sizeof(Derived4) << endl;
    cout << "sizeof(Dummy) " << sizeof(Dummy) << endl;

    return 0;
}
```

Source:

http://www2.research.att.com/~bs/bs_faq2.html

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

114. Local Classes in C++

A class declared inside a function becomes local to that function and is called Local Class in C++. For example, in the following program, Test is a local class in fun().

```

#include<iostream>
using namespace std;

void fun()
{
    class Test // local to fun
    {
        /* members of Test class */
    };
}

int main()
{
    return 0;
}

```

Following are some interesting facts about local classes.

1) *A local class type name can only be used in the enclosing function.* For example, in the following program, declarations of t and tp are valid in fun(), but invalid in main().

```

#include<iostream>
using namespace std;

void fun()
{
    // Local class
    class Test
    {
        /* ... */
    };

    Test t; // Fine
    Test *tp; // Fine
}

int main()
{
    Test t; // Error
    Test *tp; // Error
    return 0;
}

```

2) *All the methods of Local classes must be defined inside the class only.* For example, program 1 works fine and program 2 fails in compilation.

```
// PROGRAM 1
#include<iostream>
using namespace std;

void fun()
{
    class Test // local to fun
    {
    public:

        // Fine as the method is defined inside the local class
        void method() {
            cout << "Local Class method() called";
        }
    };

    Test t;
    t.method();
}

int main()
{
    fun();
    return 0;
}
```

Output:

```
Local Class method() called
```

```
// PROGRAM 2
#include<iostream>
using namespace std;

void fun()
{
    class Test // local to fun
    {
    public:
        void method();
    };

    // Error as the method is defined outside the local class
    void Test::method()
    {
        cout << "Local Class method()";
    }
}

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

Output:

Compiler Error:

```
In function 'void fun()':
error: a function-definition is not allowed here before '{' token
```

3) A Local class cannot contain static data members. It may contain static functions though. For example, program 1 fails in compilation, but program 2 works fine.

```
// PROGRAM 1
#include<iostream>
using namespace std;

void fun()
{
    class Test // local to fun
    {
        static int i;
    };
}

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

Compiler Error:

```
In function 'void fun()':
error: local class 'class fun()::Test' shall not have static data member 'int fun
```

```
// PROGRAM 2
#include<iostream>
using namespace std;

void fun()
{
    class Test // local to fun
    {
    public:
        static void method()
        {
            cout << "Local Class method() called";
        }
    };

    Test::method();
}

int main()
{
    fun();
    return 0;
}
```

Output:

```
Local Class method() called
```

4) Member methods of local class can only access static and enum variables of the enclosing function. Non-static variables of the enclosing function are not accessible inside local classes. For example, the program 1 compiles and runs fine. But, program 2

fails in compilation.

```
// PROGRAM 1
#include<iostream>
using namespace std;

void fun()
{
    static int x;
    enum {i = 1, j = 2};

    // Local class
    class Test
    {
        public:
        void method() {
            cout << "x = " << x << endl; // fine as x is static
            cout << "i = " << i << endl; // fine as i is enum
        }
    };

    Test t;
    t.method();
}

int main()
{
    fun();
    return 0;
}
```

Output:

```
x = 0
i = 1
```

```
// PROGRAM 2
#include<iostream>
using namespace std;

void fun()
{
    int x;

    // Local class
    class Test
    {
    public:
        void method() {
            cout << "x = " << x << endl;
        }
    };

    Test t;
    t.method();
}

int main()
{
    fun();
    return 0;
}
```

Output:

```
In member function 'void fun()::Test::method()':
error: use of 'auto' variable from containing function
```

5) Local classes can access global types, variables and functions. Also, local classes can access other local classes of same function.. For example, following program works fine.

```

#include<iostream>
using namespace std;

int x;

void fun()
{
    // First Local class
    class Test1 {
    public:
        Test1() { cout << "Test1::Test1()" << endl; }
    };

    // Second Local class
    class Test2
    {
        // Fine: A local class can use other local classes of same
        Test1 t1;
    public:
        void method() {
            // Fine: Local class member methods can access global var
            cout << "x = " << x << endl;
        }
    };

    Test2 t;
    t.method();
}

int main()
{
    fun();
    return 0;
}

```

Output:

```

Test1::Test1()
x = 0

```

Also see [Nested Classes in C++](#)

References:

[Local classes \(C++ only\)](#)

[Local Classes](#)

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

Can we make a class constructor *virtual* in C++ to create polymorphic objects? No. C++ being static typed (the purpose of RTTI is different) language, it is meaningless to the C++ compiler to create an object polymorphically. The compiler must be aware of the class type to create the object. In other words, what type of object to be created is a compile time decision from C++ compiler perspective. If we make constructor virtual, compiler flags an error. In fact except *inline*, no other keyword is allowed in the declaration of constructor.

In practical scenarios we would need to create a derived class object in a class hierarchy based on some input. Putting in other words, *object creation and object type are tightly coupled which forces modifications to extended. The objective of virtual constructor is to decouple object creation from it's type.*

How can we create required type of object at runtime? For example, see the following sample program.

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

//// LIBRARY START
class Base
{
public:

    Base() { }

    virtual // Ensures to invoke actual object destructor
    ~Base() { }

    // An interface
    virtual void DisplayAction() = 0;
};

class Derived1 : public Base
{
public:
    Derived1()
    {
        cout << "Derived1 created" << endl;
    }

    ~Derived1()
    {
        cout << "Derived1 destroyed" << endl;
    }

    void DisplayAction()
    {
        cout << "Action from Derived1" << endl;
    }
};

class Derived2 : public Base
{
public:
    Derived2()
    {
        cout << "Derived2 created" << endl;
    }
};
```

```

        cout << "Derived2 created" << endl;
    }

    ~Derived2()
    {
        cout << "Derived2 destroyed" << endl;
    }

    void DisplayAction()
    {
        cout << "Action from Derived2" << endl;
    }
};

///// LIBRARY END

class User
{
public:
    // Creates Derived1
    User() : pBase(0)
    {
        // What if Derived2 is required? - Add an if-else ladder (see 1
        pBase = new Derived1();
    }

    ~User()
    {
        if( pBase )
        {
            delete pBase;
            pBase = 0;
        }
    }

    // Delegates to actual object
    void Action()
    {
        pBase->DisplayAction();
    }

private:
    Base *pBase;
};

int main()
{
    User *user = new User();

    // Need Derived1 functionality only
    user->Action();

    delete user;
}

```

In the above sample, assume that the hierarchy *Base*, *Derived1* and *Derived2* are part of library code. The class *User* is utility class trying to make use of the hierarchy. The *main* function is consuming *Base* hierarchy functionality via *User* class.

The *User* class constructor is creating *Derived1* object, always. If the *User's* consumer (the *main* in our case) needs *Derived2* functionality, *User* needs to create “***new Derived2()***” and it forces recompilation. Recompiling is bad way of design, so we can opt for the following approach.

Before going into details, let us answer, who will dictate to create either of *Derived1* or *Derived2* object? Clearly, it is the consumer of *User* class. The *User* class can make use of if-else ladder to create either *Derived1* or *Derived2*, as shown in the following sample,

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

//// LIBRARY START
class Base
{
public:
    Base() { }

    virtual // Ensures to invoke actual object destructor
    ~Base() { }

    // An interface
    virtual void DisplayAction() = 0;
};

class Derived1 : public Base
{
public:
    Derived1()
    {
        cout << "Derived1 created" << endl;
    }

    ~Derived1()
    {
        cout << "Derived1 destroyed" << endl;
    }

    void DisplayAction()
    {
        cout << "Action from Derived1" << endl;
    }
};

class Derived2 : public Base
{
public:
    Derived2()
    {
        cout << "Derived2 created" << endl;
    }

    ~Derived2()
    {
        cout << "Derived2 destroyed" << endl;
    }

    void DisplayAction()
    {
        cout << "Action from Derived2" << endl;
    }
};
```

```

    }
};

///// LIBRARY END

class User
{
public:
    // Creates Derived1 or Derived2 based on input
    User() : pBase(0)
    {
        int input; // ID to distinguish between
                   // Derived1 and Derived2

        cout << "Enter ID (1 or 2): ";
        cin >> input;

        while( (input != 1) && (input != 2) )
        {
            cout << "Enter ID (1 or 2 only): ";
            cin >> input;
        }

        if( input == 1 )
        {
            pBase = new Derived1;
        }
        else
        {
            pBase = new Derived2;
        }

        // What if Derived3 being added to the class hierarchy?
    }

    ~User()
    {
        if( pBase )
        {
            delete pBase;
            pBase = 0;
        }
    }

    // Delegates to actual object
    void Action()
    {
        pBase->DisplayAction();
    }

private:
    Base *pBase;
};

int main()
{
    User *user = new User();

    // Need either Derived1 or Derived2 functionality
    user->Action();
}

```

```
}    delete user;
```

The above code is *not* open for extension, an inflexible design. In simple words, if the library updates the *Base* class hierarchy with new class *Derived3*. How can the *User* class creates *Derived3* object? One way is to update the if-else ladder that creates *Derived3* object based on new input ID 3 as shown below,


```

#include <iostream>
using namespace std;

class User
{
public:
    User() : pBase(0)
    {
        // Creates Drived1 or Derived2 based on need

        int input; // ID to distinguish between
                  // Derived1 and Derived2

        cout << "Enter ID (1 or 2): ";
        cin >> input;

        while( (input != 1) && (input != 2) )
        {
            cout << "Enter ID (1 or 2 only): ";
            cin >> input;
        }

        if( input == 1 )
        {
            pBase = new Derived1;
        }
        else if( input == 2 )
        {
            pBase = new Derived2;
        }
        else
        {
            pBase = new Derived3;
        }
    }

    ~User()
    {
        if( pBase )
        {
            delete pBase;
            pBase = 0;
        }
    }

    // Delegates to actual object
    void Action()
    {
        pBase->DisplayAction();
    }

private:
    Base *pBase;
};

```

The above modification forces the users of *User* class to recompile, bad (inflexible) design! And won't close *User* class from further modifications due to *Base* extension.

The problem is with the creation of objects. Addition of new class to the hierarchy forcing dependents of *User* class to recompile. Can't we delegate the action of creating

objects to class hierarchy itself or to a function that behaves virtually? By delegating the object creation to class hierarchy (or to a static function) we can avoid the tight coupling between *User* and *Base* hierarchy. Enough theory, see the following code,

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

///// LIBRARY START
class Base
{
public:

    // The "Virtual Constructor"
    static Base *Create(int id);

    Base() { }

    virtual // Ensures to invoke actual object destructor
    ~Base() { }

    // An interface
    virtual void DisplayAction() = 0;
};

class Derived1 : public Base
{
public:
    Derived1()
    {
        cout << "Derived1 created" << endl;
    }

    ~Derived1()
    {
        cout << "Derived1 destroyed" << endl;
    }

    void DisplayAction()
    {
        cout << "Action from Derived1" << endl;
    }
};

class Derived2 : public Base
{
public:
    Derived2()
    {
        cout << "Derived2 created" << endl;
    }

    ~Derived2()
    {
        cout << "Derived2 destroyed" << endl;
    }

    void DisplayAction()
    {
        cout << "Action from Derived2" << endl;
    }
};
```

```

class Derived3 : public Base
{
public:
    Derived3()
    {
        cout << "Derived3 created" << endl;
    }

    ~Derived3()
    {
        cout << "Derived3 destroyed" << endl;
    }

    void DisplayAction()
    {
        cout << "Action from Derived3" << endl;
    }
};

// We can also declare "Create" outside Base
// But it is more relevant to limit it's scope to Base
Base *Base::Create(int id)
{
    // Just expand the if-else ladder, if new Derived class is created
    // User code need not be recompiled to create newly added class ob

    if( id == 1 )
    {
        return new Derived1;
    }
    else if( id == 2 )
    {
        return new Derived2;
    }
    else
    {
        return new Derived3;
    }
}

//// LIBRARY END

//// UTILITY START
class User
{
public:
    User() : pBase(0)
    {
        // Receives an object of Base heirarchy at runtime

        int input;

        cout << "Enter ID (1, 2 or 3): ";
        cin >> input;

        while( (input != 1) && (input != 2) && (input != 3) )
        {
            cout << "Enter ID (1, 2 or 3 only): ";
            cin >> input;
        }

        // Get object from the "Virtual Constructor"
        nBase = Base::Create(input);
    }
};

```

```

    }
    pBase = Base::Create(input);
}

~User()
{
    if( pBase )
    {
        delete pBase;
        pBase = 0;
    }
}

// Delegates to actual object
void Action()
{
    pBase->DisplayAction();
}

private:
    Base *pBase;
};

///// UTILITY END

///// Consumer of User (UTILITY) class
int main()
{
    User *user = new User();

    // Action required on any of Derived objects
    user->Action();

    delete user;
}

```

The *User* class is independent of object creation. It delegates that responsibility to *Base*, and provides an input in the form of ID. If the library adds new class *Derived4*, the library modifier will extend the if-else ladder inside *Create* to return proper object. Consumers of *User* need not recompile their code due to extension of *Base*.

Note that the function *Create* used to return different types of *Base* class objects at runtime. It acts like virtual constructor, also referred as *Factory Method* in pattern terminology.

Pattern world demonstrate different ways to implement the above concept. Also there are some potential design issues with the above code. Our objective is to provide some insights into virtual construction, creating objects dynamically based on some input. We have excellent books devoted to the subject, interested reader can refer them for more information.

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

116. Advanced C++ | Virtual Copy Constructor

In the **virtual constructor** idiom we have seen the way to construct an object whose type is not determined until runtime. Is it possible to create an object without knowing its exact class type? The *virtual copy constructor* address this question.

Sometimes we may need to construct an object from another existing object. Precisely the copy constructor does the same. The initial state of new object will be based on another existing object state. The compiler places call to copy constructor when an object being instantiated from another object. However, the compiler needs concrete type information to invoke appropriate copy constructor.

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

class Base
{
public:
    //
};

class Derived : public Base
{
public:
    Derived()
    {
        cout << "Derived created" << endl;
    }

    Derived(const Derived &rhs)
    {
        cout << "Derived created by deep copy" << endl;
    }

    ~Derived()
    {
        cout << "Derived destroyed" << endl;
    }
};

int main()
{
    Derived s1;

    Derived s2 = s1; // Compiler invokes "copy constructor"
                    // Type of s1 and s2 are concrete to compiler

    // How can we create Derived1 or Derived2 object
    // from pointer (reference) to Base class pointing Derived object?

    return 0;
}
```

What if we can't decide from which type of object, the copy construction to be made? For example, **virtual constructor** creates an object of class hierarchy at runtime based on some input. When we want to copy construct an object from another object created by

virtual constructor, we can't use usual copy constructor. We need a special cloning function that can duplicate the object at runtime.

As an example, consider a drawing application. You can select an object already drawn on the canvas and paste one more instance of the same object. From the programmer perspective, we can't decide which object will be copy-pasted as it is runtime decision. We need virtual copy constructor to help.

Similarly, consider clip board application. A clip board can hold different type of objects, and copy objects from existing objects, pastes them on application canvas. Again, what type of object to be copied is a runtime decision. Virtual copy constructor fills the gap here. See the example below,

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

//// LIBRARY SRART
class Base
{
public:
    Base() { }

    virtual // Ensures to invoke actual object destructor
        ~Base() { }

    virtual void ChangeAttributes() = 0;

    // The "Virtual Constructor"
    static Base *Create(int id);

    // The "Virtual Copy Constructor"
    virtual Base *Clone() = 0;
};

class Derived1 : public Base
{
public:
    Derived1()
    {
        cout << "Derived1 created" << endl;
    }

    Derived1(const Derived1& rhs)
    {
        cout << "Derived1 created by deep copy" << endl;
    }

    ~Derived1()
    {
        cout << "~Derived1 destroyed" << endl;
    }

    void ChangeAttributes()
    {
        cout << "Derived1 Attributes Changed" << endl;
    }

    Base *Clone()
    {
```

```

    {
        return new Derived1(*this);
    }
};

class Derived2 : public Base
{
public:
    Derived2()
    {
        cout << "Derived2 created" << endl;
    }

    Derived2(const Derived2& rhs)
    {
        cout << "Derived2 created by deep copy" << endl;
    }

    ~Derived2()
    {
        cout << "~Derived2 destroyed" << endl;
    }

    void ChangeAttributes()
    {
        cout << "Derived2 Attributes Changed" << endl;
    }

    Base *Clone()
    {
        return new Derived2(*this);
    }
};

class Derived3 : public Base
{
public:
    Derived3()
    {
        cout << "Derived3 created" << endl;
    }

    Derived3(const Derived3& rhs)
    {
        cout << "Derived3 created by deep copy" << endl;
    }

    ~Derived3()
    {
        cout << "~Derived3 destroyed" << endl;
    }

    void ChangeAttributes()
    {
        cout << "Derived3 Attributes Changed" << endl;
    }

    Base *Clone()
    {
        return new Derived3(*this);
    }
};

```

```

// We can also declare "Create" outside Base.
// But is more relevant to limit it's scope to Base
Base *Base::Create(int id)
{
    // Just expand the if-else ladder, if new Derived class is created
    // User need not be recompiled to create newly added class objects

    if( id == 1 )
    {
        return new Derived1;
    }
    else if( id == 2 )
    {
        return new Derived2;
    }
    else
    {
        return new Derived3;
    }
}
///// LIBRARY END

///// UTILITY SRART
class User
{
public:
    User() : pBase(0)
    {
        // Creates any object of Base heirarchey at runtime

        int input;

        cout << "Enter ID (1, 2 or 3): ";
        cin >> input;

        while( (input != 1) && (input != 2) && (input != 3) )
        {
            cout << "Enter ID (1, 2 or 3 only): ";
            cin >> input;
        }

        // Create objects via the "Virtual Constructor"
        pBase = Base::Create(input);
    }

    ~User()
    {
        if( pBase )
        {
            delete pBase;
            pBase = 0;
        }
    }

    void Action()
    {
        // Duplicate current object
        Base *pNewBase = pBase->Clone();

        // Change its attributes
        pNewBase->ChangeAttributes();
    }
}

```



```

        // Dispose the created object
        delete pNewBase;
    }

private:
    Base *pBase;
};

///// UTILITY END

///// Consumer of User (UTILITY) class
int main()
{
    User *user = new User();

    user->Action();

    delete user;
}

```

User class creating an object with the help of virtual constructor. The object to be created is based on user input. *Action()* is making duplicate of object being created and modifying it's attributes. The duplicate object being created with the help of *Clone()* virtual function which is also considered as *virtual copy constructor*. The concept behind *Clone()* method is building block of *prototype pattern*.

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

117. Template Metaprogramming in C++

Predict the output of following C++ program.

```

#include <iostream>
using namespace std;

template<int n> struct funStruct
{
    enum { val = 2*funStruct<n-1>::val };
};

template<> struct funStruct<0>
{
    enum { val = 1 };
};

int main()
{
    cout << funStruct<8>::val << endl;
    return 0;
}

```

Output:

256

The program calculates “2 raise to the power 8 (or 2^8)”. In fact, the structure *funStruct* can be used to calculate 2^n for any known n (or constant n). The special thing about above program is: **calculation is done at compile time**. So, it is compiler that calculates 2^8 . To understand how compiler does this, let us consider the following facts about templates and enums:

- 1) We can pass nontype parameters (parameters that are not data types) to class/function templates.
- 2) Like other const expressions, values of enumeration constants are evaluated at compile time.
- 3) When compiler sees a new argument to a template, compiler creates a new instance of the template.

Let us take a closer look at the original program. When compiler sees *funStruct<8>::val*, it tries to create an instance of *funStruct* with parameter as 8, it turns out that *funStruct<7>* must also be created as enumeration constant *val* must be evaluated at compile time. For *funStruct<7>*, compiler need *funStruct<6>* and so on. Finally, compiler uses *funStruct<1>::val* and compile time recursion terminates. So, using templates, we can write programs that do computation at compile time, such programs are called **template metaprograms**. Template metaprogramming is in fact **Turing-complete**, meaning that any computation expressible by a computer program can be computed, in some form, by a template metaprogram. Template Metaprogramming is generally not used in practical programs, it is an interesting concept though.

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

118. Understanding “volatile” qualifier in C

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 extension)
- 2) assembly code (having .s extension) 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 2016-11-19 16:19 volatile.s
[narendra@ubuntu]$
```

Let us compile same code with optimization option (i.e. -O option). In the 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 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 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.

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.

119. Simulating final class in C++

Ever wondered how can you design a class in C++ which can't be inherited. Java and C# programming languages have this feature built-in. You can use **final** keyword in java, **sealed** in C# to make a class non-extendable.

Below is a mechanism using which we can achieve the same behavior in C++. It makes use of private constructor, virtual inheritance and friend class.

In the following code, we make the *Final* class non-inheritable. When a class *Derived* tries to inherit from it, we get compilation error.

An extra class *MakeFinal* (whose default constructor is private) is used for our purpose. Constructor of *Final* can call private constructor of *MakeFinal* as *Final* is a friend of *MakeFinal*.

Note that *MakeFinal* is also a virtual base class. The reason for this is to call the constructor of *MakeFinal* through the constructor of *Derived*, not *Final* (The constructor of a virtual base class is not called by the class that inherits from it, instead the constructor is called by the constructor of the concrete class).

```
/* A program with compilation error to demonstrate that Final class can
   be inherited */
#include<iostream>
using namespace std;

class Final; // The class to be made final

class MakeFinal // used to make the Final class final
{
private:
    MakeFinal() { cout << "MakFinal constructor" << endl; }
friend class Final;
};

class Final : virtual MakeFinal
{
public:
    Final() { cout << "Final constructor" << endl; }
};

class Derived : public Final // Compiler error
{
public:
    Derived() { cout << "Derived constructor" << endl; }
};

int main(int argc, char *argv[])
{
    Derived d;
    return 0;
}
```

Output: *Compiler Error*

```
In constructor 'Derived::Derived()':
error: 'MakeFinal::MakeFinal()' is private
```

In the above example, *Derived*'s constructor directly invokes *MakeFinal*'s constructor, and the constructor of *MakeFinal* is private, therefore we get the compilation error.

You can create the object of *Final* class as it is friend class of *MakeFinal* and has access to its constructor. For example, the following program works fine.

```
/* A program without any compilation error to demonstrate that instance
the Final class can be created */
#include<iostream>
using namespace std;

class Final;

class MakeFinal
{
private:
    MakeFinal() { cout << "MakeFinal constructor" << endl; }
    friend class Final;
};

class Final : virtual MakeFinal
{
public:
    Final() { cout << "Final constructor" << endl; }
};

int main(int argc, char *argv[])
{
    Final f;
    return 0;
}
```

Output: *Compiles and runs fine*

```
MakeFinal constructor
Final constructor
```

This article is compiled by Gopal Gorthi 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.

120. Exception handling and object destruction | Set 1

Predict the output of following C++ program.

```

#include <iostream>
using namespace std;

class Test {
public:
    Test() { cout << "Constructing an object of Test " << endl; }
    ~Test() { cout << "Destructing an object of Test " << endl; }
};

int main() {
    try {
        Test t1;
        throw 10;
    } catch(int i) {
        cout << "Caught " << i << endl;
    }
}

```

Output:

```

Constructing an object of Test
Destructing an object of Test
Caught 10

```

When an exception is thrown, destructors of the objects (whose scope ends with the try block) is automatically called before the catch block gets executed. That is why the above program prints "Destructing an object of Test" before "Caught 10".

What happens when an exception is thrown from a constructor? Consider the following program.

```

#include <iostream>
using namespace std;

class Test1 {
public:
    Test1() { cout << "Constructing an Object of Test1" << endl; }
    ~Test1() { cout << "Destructing an Object of Test1" << endl; }
};

class Test2 {
public:
    // Following constructor throws an integer exception
    Test2() { cout << "Constructing an Object of Test2" << endl;
              throw 20; }
    ~Test2() { cout << "Destructing an Object of Test2" << endl; }
};

int main() {
    try {
        Test1 t1; // Constructed and destructed
        Test2 t2; // Partially constructed
        Test1 t3; // t3 is not constructed as this statement never gets e:
    } catch(int i) {
        cout << "Caught " << i << endl;
    }
}

```


Output:

```
Constructing an Object of Test1
Constructing an Object of Test2
Destructing an Object of Test1
Caught 20
```

Destructors are only called for the completely constructed objects. When constructor of an object throws an exception, destructor for that object is not called.

As an exercise, predict the output of following program.

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

class Test {
    static int count;
    int id;
public:
    Test() {
        count++;
        id = count;
        cout << "Constructing object number " << id << endl;
        if(id == 4)
            throw 4;
    }
    ~Test() { cout << "Destructing object number " << id << endl; }
};

int Test::count = 0;

int main() {
    try {
        Test array[5];
    } catch(int i) {
        cout << "Caught " << i << endl;
    }
}
```

We will be covering more of this topic in a separate post. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

121. 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 highlighted in red colour.

```
[narendra@ubuntu]$ valgrind -leak-check=full ./free
==1238== LEAK SUMMARY:

==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
```

```
[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.

122. Catch block and type conversion in C++

Predict the output of following C++ program.

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

int main()
{
    try
    {
        throw 'x';
    }
    catch(int x)
    {
        cout << " Caught int " << x;
    }
    catch(...)
    {
        cout << "Defaule catch block";
    }
}
```

```
Defaule catch block
```

In the above program, a character 'x' is thrown and there is a catch block to catch an int. One might think that the int catch block could be matched by considering ASCII value of 'x'. But such conversions are not performed for catch blocks. Consider the following program as another example where conversion constructor is not called for thrown object.

```

#include <iostream>
using namespace std;

class MyExcept1 {};

class MyExcept2
{
public:
    // Conversion constructor
    MyExcept2 (const MyExcept1 &e )
    {
        cout << "Conversion constructor called";
    }
};

int main()
{
    try
    {
        MyExcept1 myexp1;
        throw myexp1;
    }
    catch(MyExcept2 e2)
    {
        cout << "Caught MyExcept2 " << endl;
    }
    catch(...)
    {
        cout << " Defaule catch block " << endl;
    }
    return 0;
}

```

Defaule catch block

As a side note, the derived type objects are converted to base typr when a derived object is thrown and there is a catch block to catch base type. See [this GFact](#) for more details.

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

123. 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.

124. 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 expression 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 returning 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.

125. Initialization of variable 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.

126. 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 square 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 occurrence of that character. For example, given below scanset will read all characters but stops after first occurrence 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() function 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 idea 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.

127. Use of explicit keyword in C++

Predict the output of following C++ program.

```

#include <iostream>

using namespace std;

class Complex
{
private:
    double real;
    double imag;

public:
    // Default constructor
    Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}

    // A method to compare two Complex numbers
    bool operator == (Complex rhs) {
        return (real == rhs.real && imag == rhs.imag)? true : false;
    }
};

int main()
{
    // a Complex object
    Complex com1(3.0, 0.0);

    if (com1 == 3.0)
        cout << "Same";
    else
        cout << "Not Same";
    return 0;
}

```

Output: The program compiles fine and produces following output.

```
Same
```

As discussed in [this GFact](#), in C++, if a class has a constructor which can be called with a single argument, then this constructor becomes conversion constructor because such a constructor allows conversion of the single argument to the class being constructed. *We can avoid such implicit conversions as these may lead to unexpected results. We can make the constructor explicit with the help of [explicit keyword](#).* For example, if we try the following program that uses explicit keyword with constructor, we get compilation error.

```

#include <iostream>

using namespace std;

class Complex
{
private:
    double real;
    double imag;

public:
    // Default constructor
    explicit Complex(double r = 0.0, double i = 0.0) : real(r), imag(i)

    // A method to compare two Complex numbers
    bool operator== (Complex rhs) {
        return (real == rhs.real && imag == rhs.imag)? true : false;
    }
};

int main()
{
    // a Complex object
    Complex com1(3.0, 0.0);

    if (com1 == 3.0)
        cout << "Same";
    else
        cout << "Not Same";
    return 0;
}

```

Output: Compiler Error

```
no match for 'operator==' in 'com1 == 3.0e+0'
```

We can still typecast the double values to Complex, but now we have to explicitly typecast it. For example, the following program works fine.

```

#include <iostream>

using namespace std;

class Complex
{
private:
    double real;
    double imag;

public:
    // Default constructor
    explicit Complex(double r = 0.0, double i = 0.0) : real(r), imag(i)

    // A method to compare two Complex numbers
    bool operator== (Complex rhs) {
        return (real == rhs.real && imag == rhs.imag)? true : false;
    }
};

int main()
{
    // a Complex object
    Complex com1(3.0, 0.0);

    if (com1 == (Complex)3.0)
        cout << "Same";
    else
        cout << "Not Same";
    return 0;
}

```

Output: The program compiles fine and produces following output.

```
Same
```

Also see [Advanced C++ | Conversion Operators](#)

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

128. 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.

129. 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.

130. 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,
    // 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
    // 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://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.

131. 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

132. 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 'width', 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

Copy elision (or Copy omission) is a compiler optimization technique that avoids unnecessary copying of objects. Now a days, almost every compiler uses it. Let us understand it with the help of an example.

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

class B
{
public:
    B(const char* str = "\\0") //default constructor
    {
        cout << "Constructor called" << endl;
    }

    B(const B &b) //copy constructor
    {
        cout << "Copy constructor called" << endl;
    }
};

int main()
{
    B ob = "copy me";
    return 0;
}
```

The output of above program is:

```
Constructor called
```

Why copy constructor is not called?

According to theory, when the object "ob" is being constructed, one argument constructor is used to convert "copy me" to a temporary object & that temporary object is copied to the object "ob". So the statement

```
B ob = "copy me";
```

should be broken down by the compiler as

```
B ob = B("copy me");
```

However, most of the C++ compilers avoid such overheads of creating a temporary object & then copying it.

The modern compilers break down the statement

```
B ob = "copy me"; //copy initialization
```

as

```
B ob("copy me"); //direct initialization
```

and thus eliding call to copy constructor.

However, if we still want to ensure that the compiler doesn't elide the call to copy constructor [disable the copy elision], we can compile the program using "-fno-elide-

constructors" option with g++ and see the output as following:

```
aashish@aashish-ThinkPad-SL400:~$ g++ copy_elision.cpp -fno-elide-constructors
aashish@aashish-ThinkPad-SL400:~$ ./a.out
Constructor called
Copy constructor called
```

If "-fno-elide-constructors" option is used, first default constructor is called to create a temporary object, then copy constructor is called to copy the temporary object to ob.

Reference:

http://en.wikipedia.org/wiki/Copy_elision

This article is compiled by **Aashish Barnwal** 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

134. Playing with Destructors in C++

Predict the output of the below code snippet.

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

int i;

class A
{
public:
    ~A()
    {
        i=10;
    }
};

int foo()
{
    i=3;
    A ob;
    return i;
}

int main()
{
    cout << "i = " << foo() << endl;
    return 0;
}
```

Output of the above program is "i = 3".

Why the output is i= 3 and not 10?

While returning from a function, destructor is the last method to be executed. The destructor for the object "ob" is called after the value of i is copied to the return value of the function. So, before destructor could change the value of i to 10, the current value of i gets copied & hence the output is i = 3.

How to make the program to output "i = 10" ?

Following are two ways of returning updated value:

1) Return by Reference:

Since reference gives the l-value of the variable, by using return by reference the program will output "i = 10".

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

int i;

class A
{
public:
    ~A()
    {
        i = 10;
    }
};

int& foo()
{
    i = 3;
    A ob;
    return i;
}

int main()
{
    cout << "i = " << foo() << endl;
    return 0;
}
```

The function foo() returns the l-value of the variable i. So, the address of i will be copied in the return value. Since, the references are automatically dereferenced. It will output "i = 10".

2. Create the object ob in a block scope

```

#include <iostream>
using namespace std;

int i;

class A
{
public:
    ~A()
    {
        i = 10;
    }
};

int foo()
{
    i = 3;
    {
        A ob;
    }
    return i;
}

int main()
{
    cout << "i = " << foo() << endl;
    return 0;
}

```

Since the object ob is created in the block scope, the destructor of the object will be called after the block ends, thereby changing the value of i to 10. Finally 10 will be copied to the return value.

This article is compiled by **Aashish Barnwal** 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

135. References in C++

When a variable is declared as reference, it becomes an alternative name for an existing variable. A variable can be declared as reference by putting '&' in the declaration.

```

#include<iostream>
using namespace std;

int main()
{
    int x = 10;

    // ref is a reference to x.
    int& ref = x;

    // Value of x is now changed to 20
    ref = 20;
    cout << "x = " << x << endl ;

    // Value of x is now changed to 30
    x = 30;
    cout << "ref = " << ref << endl ;

    return 0;
}

```

Output:

```

x = 20
ref = 30

```

Following is one more example that uses references to swap two variables.

```

#include<iostream>
using namespace std;

void swap (int& first, int& second)
{
    int temp = first;
    first = second;
    second = temp;
}

int main()
{
    int a = 2, b = 3;
    swap( a, b );
    cout << a << " " << b;
    return 0;
}

```

Output:

```

3 2

```

References vs Pointers

Both references and pointers can be used to change local variables of one function inside another function. Both of them can also be used to save copying of big objects when passed as arguments to functions or returned from functions, to get efficiency gain.

Despite above similarities, there are following differences between references and

pointers.

References are less powerful than pointers

- 1) Once a reference is created, it cannot be later made to reference another object; it cannot be resealed. This is often done with pointers.
- 2) References cannot be NULL. Pointers are often made NULL to indicate that they are not pointing to any valid thing.
- 3) A reference must be initialized when declared. There is no such restriction with pointers

Due to the above limitations, references in C++ cannot be used for implementing data structures like Linked List, Tree, etc. In Java, references don't have above restrictions, and can be used to implement all data structures. References being more powerful in Java, is the main reason Java doesn't need pointers.

References are safer and easier to use:

- 1) *Safer*: Since references must be initialized, wild references like **wild pointers** are unlikely to exist. It is still possible to have references that don't refer to a valid location (See questions 5 and 6 in the below exercise)
- 2) *Easier to use*: References don't need dereferencing operator to access the value. They can be used like normal variables. '&' operator is needed only at the time of declaration. Also, members of an object reference can be accessed with dot operator ('.'), unlike pointers where arrow operator (->) is needed to access members.

Together with the above reasons, there are few places like copy constructor argument where pointer cannot be used. Reference must be used pass the argument in copy constructor. Similarly references must be used for overloading some operators like ++.

Exercise:

Predict the output of following programs. If there are compilation errors, then fix them.

Question 1

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

int &fun()
{
    static int x = 10;
    return x;
}
int main()
{
    fun() = 30;
    cout << fun();
    return 0;
}
```

Question 2

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

int fun(int &x)
{
    return x;
}

int main()
{
    cout << fun(10);
    return 0;
}
```

Question 3

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

void swap(char * &str1, char * &str2)
{
    char *temp = str1;
    str1 = str2;
    str2 = temp;
}

int main()
{
    char *str1 = "GEEKS";
    char *str2 = "FOR GEEKS";
    swap(str1, str2);
    cout<<"str1 is "<<str1<<endl;
    cout<<"str2 is "<<str2<<endl;
    return 0;
}
```

Question 4

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

int main()
{
    int x = 10;
    int *ptr = &x;
    int &*ptr1 = ptr;
}
```

Question 5

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

int main()
{
    int *ptr = NULL;
    int &ref = *ptr;
    cout << ref;
}
```

Question 6

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

int &fun()
{
    int x = 10;
    return x;
}
int main()
{
    fun() = 30;
    cout << fun();
    return 0;
}
```

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

136. 'this' pointer in C++

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

For a class X, the type of this pointer is 'X* const'. Also, if a member function of X is declared as const, then the type of this pointer is 'const X *const' (see [this GFact](#))

Following are the situations where 'this' pointer is used:

1) When local variable's name is same as member's name

```

#include<iostream>
using namespace std;

/* local variable is same as a member's name */
class Test
{
private:
    int x;
public:
    void setX (int x)
    {
        // The 'this' pointer is used to retrieve the object's x
        // hidden by the local variable 'x'
        this->x = x;
    }
    void print() { cout << "x = " << x << endl; }
};

int main()
{
    Test obj;
    int x = 20;
    obj.setX(x);
    obj.print();
    return 0;
}

```

Output:

```
x = 20
```

For constructors, **initializer list** can also be used when parameter name is same as member's name.

2) To return reference to the calling object

```

/* Reference to the calling object can be returned */
Test& Test::func ()
{
    // Some processing
    return *this;
}

```

When a reference to a local object is returned, the returned reference can be used to **chain function calls** on a single object.


```

#include<iostream>
using namespace std;

class Test
{
private:
    int x;
    int y;
public:
    Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
    Test &setX(int a) { x = a; return *this; }
    Test &setY(int b) { y = b; return *this; }
    void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
    Test obj1(5, 5);

    // Chained function calls. All calls modify the same object
    // as the same object is returned by reference
    obj1.setX(10).setY(20);

    obj1.print();
    return 0;
}

```

Output:

```
x = 10 y = 20
```

Exercise:

Predict the output of following programs. If there are compilation errors, then fix them.

Question 1

```

#include<iostream>
using namespace std;

class Test
{
private:
    int x;
public:
    Test(int x = 0) { this->x = x; }
    void change(Test *t) { this = t; }
    void print() { cout << "x = " << x << endl; }
};

int main()
{
    Test obj(5);
    Test *ptr = new Test (10);
    obj.change(ptr);
    obj.print();
    return 0;
}

```

Question 2

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

class Test
{
private:
    int x;
    int y;
public:
    Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
    static void fun1() { cout << "Inside fun1()"; }
    static void fun2() { cout << "Inside fun2()"; this->fun1(); }
};

int main()
{
    Test obj;
    obj.fun2();
    return 0;
}
```

Question 3

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

class Test
{
private:
    int x;
    int y;
public:
    Test (int x = 0, int y = 0) { this->x = x; this->y = y; }
    Test setX(int a) { x = a; return *this; }
    Test setY(int b) { y = b; return *this; }
    void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
    Test obj1;
    obj1.setX(10).setY(20);
    obj1.print();
    return 0;
}
```

Question 4

```

#include<iostream>
using namespace std;

class Test
{
private:
    int x;
    int y;
public:
    Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
    void setX(int a) { x = a; }
    void setY(int b) { y = b; }
    void destroy() { delete this; }
    void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
    Test obj;
    obj.destroy();
    obj.print();
    return 0;
}

```

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

137. Casting operators in C++ | Set 1 (const_cast)

C++ supports following 4 types of casting operators:

1. const_cast
2. static_cast
3. dynamic_cast
4. reinterpret_cast

1. const_cast

const_cast is used to cast away the constness of variables. Following are some interesting facts about const_cast.

1) const_cast can be used to change non-const class members inside a const member function. Consider the following code snippet. Inside const member function fun(), 'this' is treated by the compiler as 'const student* const this', i.e. 'this' is a constant pointer to a constant object, thus compiler doesn't allow to change the data members through 'this' pointer. const_cast changes the type of 'this' pointer to 'student* const this'.

```

#include <iostream>
using namespace std;

class student
{
private:
    int roll;
public:
    // constructor
    student(int r):roll(r) {}

    // A const function that changes roll with the help of const_cast
    void fun() const
    {
        ( const_cast <student*> (this) )->roll = 5;
    }

    int getRoll() { return roll; }
};

int main(void)
{
    student s(3);
    cout << "Old roll number: " << s.getRoll() << endl;

    s.fun();

    cout << "New roll number: " << s.getRoll() << endl;

    return 0;
}

```

Output:

```

Old roll number: 3
New roll number: 5

```

2) `const_cast` can be used to pass const data to a function that doesn't receive const. For example, in the following program `fun()` receives a normal pointer, but a pointer to a const can be passed with the help of `const_cast`.

```

#include <iostream>
using namespace std;

int fun(int* ptr)
{
    return (*ptr + 10);
}

int main(void)
{
    const int val = 10;
    const int *ptr = &val;
    int *ptr1 = const_cast <int *>(ptr);
    cout << fun(ptr1);
    return 0;
}

```

Output:

20

3) It is undefined behavior to modify a value which is initially declared as const. Consider the following program. The output of the program is undefined. The variable 'val' is a const variable and the call 'fun(ptr1)' tries to modify 'val' using const_cast.

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

int fun(int* ptr)
{
    *ptr = *ptr + 10;
    return (*ptr);
}

int main(void)
{
    const int val = 10;
    const int *ptr = &val;
    int *ptr1 = const_cast <int *>(ptr);
    fun(ptr1);
    cout << val;
    return 0;
}
```

Output:

Undefined Behavior

It is fine to modify a value which is not initially declared as const. For example, in the above program, if we remove const from declaration of val, the program will produce 20 as output.

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

int fun(int* ptr)
{
    *ptr = *ptr + 10;
    return (*ptr);
}

int main(void)
{
    int val = 10;
    const int *ptr = &val;
    int *ptr1 = const_cast <int *>(ptr);
    fun(ptr1);
    cout << val;
    return 0;
}
```

4) const_cast is considered safer than simple type casting. It's safer in the sense that the

casting won't happen if the type of cast is not same as original object. For example, the following program fails in compilation because 'int *' is being typecasted to 'char *'

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

int main(void)
{
    int a1 = 40;
    const int* b1 = &a1;
    char* c1 = const_cast <char *> (b1); // compiler error
    *c1 = 'A';
    return 0;
}
```

output:

```
prog.cpp: In function 'int main()':
prog.cpp:8: error: invalid const_cast from type 'const int*' to type 'char*'
```

5) const_cast can also be used to cast away volatile attribute. For example, in the following program, the typeid of b1 is PVKi (pointer to a volatile and constant integer) and typeid of c1 is Pi (Pointer to integer)

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

int main(void)
{
    int a1 = 40;
    const volatile int* b1 = &a1;
    cout << "typeid of b1 " << typeid(b1).name() << '\n';
    int* c1 = const_cast <int *> (b1);
    cout << "typeid of c1 " << typeid(c1).name() << '\n';
    return 0;
}
```

Output:

```
typeid of b1 PVKi
typeid of c1 Pi
```

Exercise

Predict the output of following programs. If there are compilation errors, then fix them.

Question 1

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

int main(void)
{
    int a1 = 40;
    const int* b1 = &a1;
    char* c1 = (char *)(b1);
    *c1 = 'A';
    return 0;
}
```

Question 2

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

class student
{
private:
    const int roll;
public:
    // constructor
    student(int r):roll(r) {}

    // A const function that changes roll with the help of const_cast
    void fun() const
    {
        ( const_cast <student*> (this) )->roll = 5;
    }

    int getRoll() { return roll; }
};

int main(void)
{
    student s(3);
    cout << "Old roll number: " << s.getRoll() << endl;

    s.fun();

    cout << "New roll number: " << s.getRoll() << endl;

    return 0;
}
```

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

138. 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 "Struct 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 guarantees 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 contiguous). 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

139. Default arguments and virtual function

Predict the output of following C++ program.

```

#include <iostream>
using namespace std;

class Base
{
public:
    virtual void fun ( int x = 0 )
    {
        cout << "Base::fun(), x = " << x << endl;
    }
};

class Derived : public Base
{
public:
    virtual void fun ( int x )
    {
        cout << "Derived::fun(), x = " << x << endl;
    }
};

int main()
{
    Derived d1;
    Base *bp = &d1;
    bp->fun();
    return 0;
}

```

Output:

```
Derived::fun(), x = 0
```

If we take a closer look at the output, we observe that fun() of derived class is called and default value of base class fun() is used.

Default arguments do not participate in signature of functions. So signatures of fun() in base class and derived class are considered same, hence the fun() of base class is overridden. Also, the default value is used at compile time. When compiler sees that an argument is missing in a function call, it substitutes the default value given. Therefore, in the above program, value of x is substituted at compile time, and at run time derived class's fun() is called.

Now predict the output of following program.

```

#include <iostream>
using namespace std;

class Base
{
public:
    virtual void fun ( int x = 0)
    {
        cout << "Base::fun(), x = " << x << endl;
    }
};

class Derived : public Base
{
public:
    virtual void fun ( int x = 10 ) // NOTE THIS CHANGE
    {
        cout << "Derived::fun(), x = " << x << endl;
    }
};

int main()
{
    Derived d1;
    Base *bp = &d1;
    bp->fun();
    return 0;
}

```

The output of this program is same as the previous program. The reason is same, the default value is substituted at compile time. The fun() is called on bp which is a pointer of Base type. So compiler substitutes 0 (not 10).

In general, it is a best practice to avoid default values in virtual functions to avoid confusion (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

140. Static data members in C++

Predict the output of following C++ program:

```

#include <iostream>
using namespace std;

class A
{
public:
    A() { cout << "A's Constructor Called " << endl; }
};

class B
{
    static A a;
public:
    B() { cout << "B's Constructor Called " << endl; }
};

int main()
{
    B b;
    return 0;
}

```

Output:

```
B's Constructor Called
```

The above program calls only B's constructor, it doesn't call A's constructor. The reason for this is simple, *static members are only declared in class declaration, not defined. They must be explicitly defined outside the class using scope resolution operator.* If we try to access static member 'a' without explicit definition of it, we will get compilation error. For example, following program fails in compilation.

```

#include <iostream>
using namespace std;

class A
{
    int x;
public:
    A() { cout << "A's constructor called " << endl; }
};

class B
{
    static A a;
public:
    B() { cout << "B's constructor called " << endl; }
    static A getA() { return a; }
};

int main()
{
    B b;
    A a = b.getA();
    return 0;
}

```

Output:

```
Compiler Error: undefined reference to `B::a'
```

If we add definition of a, the program will work fine and will call A's constructor. See the following program.

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

class A
{
    int x;
public:
    A() { cout << "A's constructor called " << endl; }
};

class B
{
    static A a;
public:
    B() { cout << "B's constructor called " << endl; }
    static A getA() { return a; }
};

A B::a; // definition of a

int main()
{
    B b1, b2, b3;
    A a = b1.getA();

    return 0;
}
```

Output:

```
A's constructor called
B's constructor called
B's constructor called
B's constructor called
```

Note that the above program calls B's constructor 3 times for 3 objects (b1, b2 and b3), but calls A's constructor only once. The reason is, *static members are shared among all objects. That is why they are also known as class members or class fields*. Also, *static members can be accessed without any object*, see the below program where static member 'a' is accessed without any object.

```

#include <iostream>
using namespace std;

class A
{
    int x;
public:
    A() { cout << "A's constructor called " << endl; }
};

class B
{
    static A a;
public:
    B() { cout << "B's constructor called " << endl; }
    static A getA() { return a; }
};

A B::a; // definition of a

int main()
{
    // static member 'a' is accessed without any object of B
    A a = B::getA();

    return 0;
}

```

Output:

```
A's constructor called
```

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

141. 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

142. Private Destructor

Predict the output of following programs.

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

class Test
{
private:
    ~Test() {}
};
int main()
{ }
```

The above program compiles and runs fine. It is **not** compiler error to create private destructors. What do you say about below program.

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

class Test
{
private:
    ~Test() {}
};
int main()
{
    Test t;
}
```

The above program fails in compilation. The compiler notices that the local variable 't' cannot be destructed because the destructor is private. What about the below program?

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

class Test
{
private:
    ~Test() {}
};
int main()
{
    Test *t;
}
```

The above program works fine. There is no object being constructed, the program just creates a pointer of type "Test *", so nothing is destructed. What about the below program?

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

class Test
{
private:
    ~Test() {}
};
int main()
{
    Test *t = new Test;
}
```

The above program also works fine. When something is created using dynamic memory allocation, it is programmer's responsibility to delete it. So compiler doesn't bother.

The below program fails in compilation. When we call delete, destructor is called.

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

class Test
{
private:
    ~Test() {}
};
int main()
{
    Test *t = new Test;
    delete t;
}
```

We noticed in the above programs, when a class has private destructor, only dynamic objects of that class can be created. Following is a way to create classes with private destructors and have a function as friend of the class. The function can only delete the objects.


```

#include <iostream>

// A class with private destructor
class Test
{
private:
    ~Test() {}
    friend void destructTest(Test* );
};

// Only this function can destruct objects of Test
void destructTest(Test* ptr)
{
    delete ptr;
}

int main()
{
    // create an object
    Test *ptr = new Test;

    // destruct the object
    destructTest (ptr);

    return 0;
}

```

What is the use of private destructor?

Whenever we want to control destruction of objects of a class, we make the destructor private. For dynamically created objects, it may happen that you pass a pointer to the object to a function and the function deletes the object. If the object is referred after the function call, the reference will become dangling. 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

143. 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 runs fine because the 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 are also `int`. Had the compiler assumed input parameters are `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

144. What happens when more restrictive access is given to a derived class method in C++?

We have discussed a similar topic in Java [here](#). Unlike Java, C++ allows to give more restrictive access to derived class methods. For example the following program compiles fine.

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

class Base {
public:
    virtual int fun(int i) { }
};

class Derived: public Base {
private:
    int fun(int x) { }
};

int main()
{ }
```

In the above program, if we change main() to following, will get compiler error because fun() is private in derived class.

```
int main()
{
    Derived d;
    d.fun(1);
    return 0;
}
```

What about the below program?

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

class Base {
public:
    virtual int fun(int i) { cout << "Base::fun(int i) called"; }
};

class Derived: public Base {
private:
    int fun(int x) { cout << "Derived::fun(int x) called"; }
};

int main()
{
    Base *ptr = new Derived;
    ptr->fun(10);
    return 0;
}
```

Output:

```
Derived::fun(int x) called
```

In the above program, private function "Derived::fun(int)" is being called through a base class pointer, the program works fine because fun() is public in base class. Access specifiers are checked at compile time and fun() is public in base class. At run time, only the function corresponding to the pointed object is called and access specifier is not checked. So a private function of derived class is being called through a pointer of base class.

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

145. Multiple Inheritance in C++

Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes.

The constructors of inherited classes are called in the same order in which they are inherited. For example, in the following program, B's constructor is called before A's constructor.

```

#include<iostream>
using namespace std;

class A
{
public:
    A() { cout << "A's constructor called" << endl; }
};

class B
{
public:
    B() { cout << "B's constructor called" << endl; }
};

class C: public B, public A // Note the order
{
public:
    C() { cout << "C's constructor called" << endl; }
};

int main()
{
    C c;
    return 0;
}

```

Output:

```

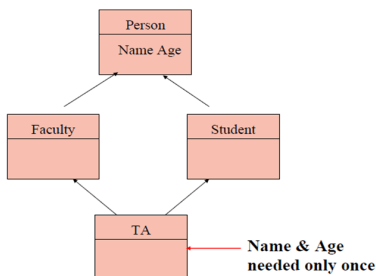
B's constructor called
A's constructor called
C's constructor called

```

The destructors are called in reverse order of constructors.

The diamond problem

The diamond problem occurs when two superclasses of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities.



For example, consider the following program.

```

#include<iostream>
using namespace std;
class Person {
    // Data members of person
public:
    Person(int x) { cout << "Person::Person(int ) called" << endl;
};

class Faculty : public Person {
    // data members of Faculty
public:
    Faculty(int x):Person(x) {
        cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};

class Student : public Person {
    // data members of Student
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int ) called"<< endl;
    }
};

class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x) {
        cout<<"TA::TA(int ) called"<< endl;
    }
};

int main() {
    TA ta1(30);
}

```

```

Person::Person(int ) called
Faculty::Faculty(int ) called
Person::Person(int ) called
Student::Student(int ) called
TA::TA(int ) called

```

In the above program, constructor of 'Person' is called two times. Destructor of 'Person' will also be called two times when object 'ta1' is destructed. So object 'ta1' has two copies of all members of 'Person', this causes ambiguities. *The solution to this problem is 'virtual' keyword.* We make the classes 'Faculty' and 'Student' as virtual base classes to avoid two copies of 'Person' in 'TA' class. For example, consider the following program.

```

#include<iostream>
using namespace std;
class Person {
public:
    Person(int x) { cout << "Person::Person(int ) called" << endl;
    Person()      { cout << "Person::Person() called" << endl;    }
};

class Faculty : virtual public Person {
public:
    Faculty(int x):Person(x) {
        cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};

class Student : virtual public Person {
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int ) called"<< endl;
    }
};

class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x) {
        cout<<"TA::TA(int ) called"<< endl;
    }
};

int main() {
    TA ta1(30);
}

```

Output:

```

Person::Person() called
Faculty::Faculty(int ) called
Student::Student(int ) called
TA::TA(int ) called

```

In the above program, constructor of 'Person' is called once. One important thing to note in the above output is, *the default constructor of 'Person' is called*. When we use 'virtual' keyword, the default constructor of grandparent class is called by default even if the parent classes explicitly call parameterized constructor.

How to call the parameterized constructor of the 'Person' class? The constructor has to be called in 'TA' class. For example, see the following program.

```

#include<iostream>
using namespace std;
class Person {
public:
    Person(int x) { cout << "Person::Person(int ) called" << endl;
    Person()      { cout << "Person::Person() called" << endl;    }
};

class Faculty : virtual public Person {
public:
    Faculty(int x):Person(x) {
        cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};

class Student : virtual public Person {
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int ) called"<< endl;
    }
};

class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x), Person(x) {
        cout<<"TA::TA(int ) called"<< endl;
    }
};

int main() {
    TA ta1(30);
}

```

Output:

```

Person::Person(int ) called
Faculty::Faculty(int ) called
Student::Student(int ) called
TA::TA(int ) called

```

In general, it is not allowed to call the grandparent's constructor directly, it has to be called through parent class. It is allowed only when 'virtual' keyword is used.

As an exercise, predict the output of following programs.

Question 1


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

class A
{
    int x;
public:
    void setX(int i) {x = i;}
    void print() { cout << x; }
};

class B: public A
{
public:
    B() { setX(10); }
};

class C: public A
{
public:
    C() { setX(20); }
};

class D: public B, public C {
};

int main()
{
    D d;
    d.print();
    return 0;
}
```

Question 2

```

#include<iostream>
using namespace std;

class A
{
    int x;
public:
    A(int i) { x = i; }
    void print() { cout << x; }
};

class B: virtual public A
{
public:
    B():A(10) { }
};

class C: virtual public A
{
public:
    C():A(10) { }
};

class D: public B, public C {

int main()
{
    D d;
    d.print();
    return 0;
}

```

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

146. Function overloading and const keyword

Predict the output of following C++ program.

```

#include<iostream>
using namespace std;

class Test
{
protected:
    int x;
public:
    Test (int i):x(i) { }
    void fun() const
    {
        cout << "fun() const called " << endl;
    }
    void fun()
    {
        cout << "fun() called " << endl;
    }
};

int main()
{
    Test t1 (10);
    const Test t2 (20);
    t1.fun();
    t2.fun();
    return 0;
}

```

Output: The above program compiles and runs fine, and produces following output.

```

fun() called
fun() const called

```

The two methods 'void fun() const' and 'void fun()' have same signature except that one is const and other is not. Also, if we take a closer look at the output, we observe that, 'const void fun()' is called on const object and 'void fun()' is called on non-const object. C++ allows member methods to be overloaded on the basis of const type. Overloading on the basis of const type can be useful when a function return reference or pointer. We can make one function const, that returns a const reference or const pointer, other non-const function, that returns non-const reference or pointer. See [this](#) for more details.

What about parameters?

Rules related to const parameters are interesting. Let us first take a look at following two examples. The program 1 fails in compilation, but program 2 compiles and runs fine.

```
// PROGRAM 1 (Fails in compilation)
#include<iostream>
using namespace std;

void fun(const int i)
{
    cout << "fun(const int) called ";
}
void fun(int i)
{
    cout << "fun(int ) called " ;
}
int main()
{
    const int i = 10;
    fun(i);
    return 0;
}
```

Output:

```
Compiler Error: redefinition of 'void fun(int)'
```

```
// PROGRAM 2 (Compiles and runs fine)
#include<iostream>
using namespace std;

void fun(char *a)
{
    cout << "non-const fun() " << a;
}

void fun(const char *a)
{
    cout << "const fun() " << a;
}

int main()
{
    const char *ptr = "GeeksforGeeks";
    fun(ptr);
    return 0;
}
```

Output:

```
const fun() GeeksforGeeks
```

C++ allows functions to be overloaded on the basis of const-ness of parameters only if the const parameter is a reference or a pointer. That is why the program 1 failed in compilation, but the program 2 worked fine. This rule actually makes sense. In program 1, the parameter 'i' is passed by value, so 'i' in fun() is a copy of 'i' in main(). Hence fun() cannot modify 'i' of main(). Therefore, it doesn't matter whether 'i' is received as a const parameter or normal parameter. When we pass by reference or pointer, we can modify the value referred or pointed, so we can have two versions of a function, one which can modify the referred or pointed value, other which can not.

As an exercise, predict the output of following program.

```

#include<iostream>
using namespace std;

void fun(const int &i)
{
    cout << "fun(const int &) called ";
}
void fun(int &i)
{
    cout << "fun(int &) called " ;
}
int main()
{
    const int i = 10;
    fun(i);
    return 0;
}

```

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

147. When should we write our own assignment operator in C++?

The answer is same as Copy Constructor. If a class doesn't contain pointers, then there is no need to write assignment operator and copy constructor. The compiler creates a default copy constructor and assignment operators for every class. The compiler created copy constructor and assignment operator may not be sufficient when we have pointers or any run time allocation of resource like file handle, a network connection..etc. For example, consider the following program.

```

#include<iostream>
using namespace std;

// A class without user defined assignment operator
class Test
{
    int *ptr;
public:
    Test (int i = 0)    { ptr = new int(i); }
    void setValue (int i) { *ptr = i; }
    void print()        { cout << *ptr << endl; }
};

int main()
{
    Test t1(5);
    Test t2;
    t2 = t1;
    t1.setValue(10);
    t2.print();
    return 0;
}

```

Output of above program is “10”. If we take a look at main(), we modified ‘t1’ using setValue() function, but the changes are also reflected in object ‘t2’. This type of unexpected changes cause problems.

Since there is no user defined assignment operator in the above program, compiler creates a default assignment operator, which copies ‘ptr’ of right hand side to left hand side. So both ‘ptr’s start pointing to the same location.

We can handle the above problem in two ways.

1) Do not allow assignment of one object to other object. We can create our own dummy assignment operator and make it private.

2) Write your own assignment operator that does deep copy.

Same is true for Copy Constructor.

Following is an example of overloading assignment operator for the above class.

```

#include<iostream>
using namespace std;

class Test
{
    int *ptr;
public:
    Test (int i = 0)    { ptr = new int(i); }
    void setValue (int i) { *ptr = i; }
    void print()        { cout << *ptr << endl; }
    Test & operator = (const Test &t);
};

Test & Test::operator = (const Test &t)
{
    // Check for self assignment
    if(this != &t)
        *ptr = *(t.ptr);

    return *this;
}

int main()
{
    Test t1(5);
    Test t2;
    t2 = t1;
    t1.setValue(10);
    t2.print();
    return 0;
}

```

Output

5

We should also add a copy constructor to the above class, so that the statements like "Test t3 = t4;" also don't cause any problem.

Note the if condition in assignment operator. While overloading assignment operator, we must check for self assignment. Otherwise assigning an object to itself may lead to unexpected results (See [this](#)). Self assignment check is not necessary for the above 'Test' class, because 'ptr' always points to one integer and we may reuse the same memory. But in general, it is a recommended practice to do self-assignment check.

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

148. Default Assignment Operator and References

We have discussed assignment operator overloading for dynamically allocated

resources [here](#) . This is an extension of the previous post. In [the previous post](#), we discussed that when we don't write our own assignment operator, compiler created assignment operator does shallow copy and that cause problems. What happens when we have references in our class and there is no user defined assignment operator. For example, predict the output of following program.

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

class Test
{
    int x;
    int &ref;
public:
    Test (int i):x(i), ref(x) {}
    void print() { cout << ref; }
    void setX(int i) { x = i; }
};

int main()
{
    Test t1(10);
    Test t2(20);
    t2 = t1;
    t1.setX(40);
    t2.print();
    return 0;
}
```

Output:

```
Compiler Error: non-static reference member 'int& Test::ref',
               can't use default assignment operator
```

Compiler doesn't creates default assignment operator in following cases

1. Class has a nonstatic data member of a const type or a reference type
2. Class has a nonstatic data member of a type which has an inaccessible copy assignment operator
3. Class is derived from a base class with an inaccessible copy assignment operator

When any of the above conditions is true, user must define assignment operator. For example, if we add an assignment operator to the above code, the code works fine without any error.


```

#include<iostream>
using namespace std;

class Test
{
    int x;
    int &ref;
public:
    Test (int i):x(i), ref(x) {}
    void print() { cout << ref; }
    void setX(int i) { x = i; }
    Test &operator = (const Test &t) { x = t.x; return *this; }
};

int main()
{
    Test t1(10);
    Test t2(20);
    t2 = t1;
    t1.setX(40);
    t2.print();
    return 0;
}

```

Output:

```
10
```

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

149. Print 1 to 100 in C++, without loop and recursion

Following is a C++ program that prints 1 to 100 without loop and without recursion.

```

#include <iostream>
using namespace std;

template<int N>
class PrintOneToN
{
public:
    static void print()
    {
        PrintOneToN<N-1>::print(); // Note that this is not recursion
        cout << N << endl;
    }
};

template<>
class PrintOneToN<1>
{
public:
    static void print()
    {
        cout << 1 << endl;
    }
};

int main()
{
    const int N = 100;
    PrintOneToN<N>::print();
    return 0;
}

```

Output:

```

1
2
3
..
..
98
99
100

```

The program prints all numbers from 1 to n without using a loop and recursion. The concept used in this program is [Template Metaprogramming](#). Let us see how this works. [Templates in C++](#) allow non-datatypes also as parameter. Non-datatype means a value, not a datatype. For example, in the above program, N is passed as a value which is not a datatype. A new instance of a generic class is created for every parameter and these classes are created at compile time. In the above program, when compiler sees the statement "PrintOneToN<N>::print()" with N = 100, it creates an instance PrintOneToN<100>. In function PrintOneToN<100>::print(), another function PrintOneToN<99>::print() is called, therefore an instance PrintOneToN<99> is created. Similarly, all instances from PrintOneToN<100> to PrintOneToN<2> are created. PrintOneToN<1>::print() is already there and prints 1. The function PrintOneToN<2> prints 2 and so on. Therefore we get all numbers from 1 to N printed

on the screen.

Following is **another approach** to print 1 to 100 without loop and recursion.

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

class A
{
public:
    static int a;
    A()
    { cout<<a++<<endl; }
};

int A::a = 1;

int main()
{
    int N = 100;
    A obj[N];
    return 0;
}
```

The output of this program is same as above program. In the above program, class A has a static variable 'a', which is incremented with every instance of A. The default constructor of class A prints the value of 'a'. When we create an array of objects of type A, the default constructor is called for all objects one by one. Value of 'a' is printed and incremented with every call. Therefore, we get all values from 1 to 100 printed on the screen.

Thanks to *Lakshmanan* for suggesting this approach.

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

150. 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.

151. Program to print last 10 lines

Given some text lines in one string, each line is separated by ‘\n’ character. Print the last ten lines. If number of lines is less than 10, then print all lines.

Source: [Microsoft Interview | Set 10](#)

Following are the steps

- 1) Find the last occurrence of DELIM or ‘\n’
- 2) Initialize target position as last occurrence of ‘\n’ and count as 0 , and do following while count < 10
 -2.a) Find the next instance of ‘\n’ and update target position
 -2.b) Skip ‘\n’ and increment count of ‘\n’ and update target position

```
/* Program to print the last 10 lines. If number of lines is less
   than 10, then print all lines. */
```

```

/* Function to print last n lines of a given string */
void print_last_lines(char *str, int n)
{
    /* Base case */
    if (n <= 0)
        return;

    size_t cnt = 0; // To store count of '\n' or DELIM
    char *target_pos = NULL; // To store the output position in str

    /* Step 1: Find the last occurrence of DELIM or '\n' */
    target_pos = strchr(str, DELIM);

    /* Error if '\n' is not present at all */
    if (target_pos == NULL)
    {
        fprintf(stderr, "ERROR: string doesn't contain '\\n' character\n");
        return;
    }

    /* Step 2: Find the target position from where we need to print the string */
    while (cnt < n)
    {
        /* Step 2.a: Find the next instance of '\n' */
        while (str < target_pos && *target_pos != DELIM)
            --target_pos;

        /* Step 2.b: skip '\n' and increment count of '\n' */
        if (*target_pos == DELIM)
            --target_pos, ++cnt;

        /* str < target_pos means str has less than 10 '\n' characters
           so break from loop */
        else
            break;
    }

    /* In while loop, target_pos is decremented 2 times, that's why target_pos += 2; */
    if (str < target_pos)
        target_pos += 2;

    /* Step 3: Print the string from target_pos */
    printf("%s\n", target_pos);
}

```

```
int main(void)
```

```
char *str1 = "str1\\nstr2\\nstr3\\nstr4\\nstr5\\nstr6\\nstr7\\nstr8\\nstr9\\nstr10\\nstr11\\nstr12\\nstr13\\nstr14\\nstr15\\nstr16\\nstr17\\nstr18\\nstr19\\nstr20\\nstr21\\nstr22\\nstr23\\nstr24\\nstr25";
char *str2 = "str1\\nstr2\\nstr3\\nstr4\\nstr5\\nstr6\\nstr7";
char *str3 = "\\n";
```

```

char *str4 = "";

print_last_lines(str1, 10);
printf("-----\n");

print_last_lines(str2, 10);
printf("-----\n");

print_last_lines(str3, 10);
printf("-----\n");

print_last_lines(str4, 10);
printf("-----\n");

return 0;
}

```

Output:

```

str16
str17
str18
str19
str20
str21
str22
str23
str24
str25
-----
str1
str2
str3
str4
str5
str6
str7
-----
-----
ERROR: string doesn't contain '\n' character
-----

```

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.

152. Program to validate an IP address

Write a program to Validate an IPv4 Address.

Source: [Microsoft Interview | Set 7](#)

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
    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.

153. 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
```

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.

154. Name Mangling and extern “C” in C++

C++ supports function overloading, i.e., there can be more than one functions with same name and differences in parameters. How does C++ compiler distinguishes between different functions when it generates object code – it changes names by adding information about arguments. This technique of adding additional information to function names is called **Name Mangling**. C++ standard doesn't specify any particular technique for name mangling, so different compilers may append different information to function names.

Consider following declarations of function *f()*

```
int f (void) { return 1; }
int f (int) { return 0; }
void g (void) { int i = f(), j = f(0); }
```

A C++ compiler may mangle above names to following (Source: [Wiki](#))

```
int __f_v (void) { return 1; }
int __f_i (int) { return 0; }
void __g_v (void) { int i = __f_v(), j = __f_i(0); }
```

How to handle C symbols when linking from C++?

In C, names may not be mangled as C doesn't support function overloading. So how to make sure that name of a symbol is not changed when we link a C code in C++. For example, see the following C++ program that uses `printf()` function of C.

```
// Save file as .cpp and use C++ compiler to compile it
int printf(const char *format,...);

int main()
{
    printf("GeeksforGeeks");
    return 0;
}
```

Output:

```
undefined reference to `printf(char const*, ...)'
ld returned 1 exit status
```

The reason for compiler error is simple, name of *printf* is changed by C++ compiler and it doesn't find definition of the function with new name.

The solution of problem is extern "C" in C++. When some code is put in extern "C" block, the C++ compiler ensures that the function names are unmangled – that the compiler emits a binary file with their names unchanged, as a C compiler would do.

If we change the above program to following, the program works fine and prints "GeeksforGeeks" on console.

```
// Save file as .cpp and use C++ compiler to compile it
extern "C"
{
    int printf(const char *format,...);
}

int main()
{
    printf("GeeksforGeeks");
    return 0;
}
```

Output:

```
GeeksforGeeks
```

Therefore, all C style header files (stdio.h, string.h, .. etc) have their declarations in extern "C" block.

```
#ifndef __cplusplus
extern "C" {
#endif
    /* Declarations of this file */
#ifdef __cplusplus
}
#endif
```

Following are main points discussed above

1. Since C++ supports function overloading, additional information has to be added to function names (called name mangling) to avoid conflicts in binary code.
2. Function names may not be changed in C as C doesn't support function overloading.

To avoid linking problems, C++ supports extern "C" block. C++ compiler makes sure that names inside extern "C" block are not changed.

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

155. 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.

156. 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 before p2  
0 is equivalent to p2  
>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.

157. 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://en.wikipedia.org/wiki/C11_(C_standard_revision))

<http://www.cplusplus.com/reference/cstdio/freopen/>

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

158. When does compiler create default and copy constructors in C++?

In C++, compiler creates a **default constructor** if we don't define our own constructor (See [this](#)). Compiler created default constructor has empty body, i.e., it doesn't assign default values to data members (In java, [default constructors assign default values](#)).

Compiler also creates a copy constructor if we don't write our own copy constructor. Unlike default constructor, body of compiler created copy constructor is not empty, it copies all data members of passed object to the object which is being created.

What happens when we write only a copy constructor – does compiler create default constructor?

Compiler doesn't create a default constructor if we write any constructor even if it is copy constructor. For example, the following program doesn't compile.

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

class Point
{
    int x, y;
public:
    Point(const Point &p) { x = p.x; y = p.y; }
};

int main()
{
    Point p1; // COMPILER ERROR
    Point p2 = p1;
    return 0;
}
```

Output:

```
COMPILER ERROR: no matching function for call to 'Point::Point()'
```

What about reverse – what happens when we write a normal constructor and don't write a copy constructor?

Reverse is not true. Compiler creates a copy constructor if we don't write our own. Compiler creates it even if we have written other constructors in class. For example, the below program works fine.

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

class Point
{
    int x, y;
public:
    Point(int i, int j) { x = 10; y = 20; }
    int getX() { return x; }
    int getY() { return y; }
};

int main()
{
    Point p1(10, 20);
    Point p2 = p1; // This compiles fine
    cout << "x = " << p2.getX() << " y = " << p2.getY();
    return 0;
}
```

Output:

```
x = 10 y = 20
```

So we need to write copy constructor only when we have pointers or run time allocation of resource like file handle, a network connection, etc (See [this](#))

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

159. How to print “GeeksforGeeks” with empty main() in C++?

Write a C++ program that prints “GeeksforGeeks” with empty main() function. You are not allowed to write anything in main().

One way of doing this is to apply constructor attribute to a function so that it executes before main() (See [this](#) for details).

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

/* Apply the constructor attribute to myStartupFun() so that it
   is executed before main() */
void myStartupFun (void) __attribute__ ((constructor));

/* implementation of myStartupFun */
void myStartupFun (void)
{
    cout << "GeeksforGeeks";
}

int main ()
{
}

}
```

Output:

```
GeeksforGeeks
```

The above approach would work only with GCC family of compilers. Following is an interesting way of doing it in C++.

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

class MyClass
{
public:
    MyClass()
    {
        cout << "GeeksforGeeks";
    }
}m;

int main()
{
}

}
```

Output:

The idea is to create a class, have a cout statement in constructor and create a global object of the class. When the object is created, constructor is called and "GeeksforGeeks" is printed.

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

160. 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( "\r\nLength 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 entered 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.

161. Why copy constructor argument should be const in C++?

When we create our own copy constructor, we pass an object by reference and we generally pass it as a const reference.

One reason for passing const reference is, we should use const in C++ wherever possible so that objects are not accidentally modified. This is one good reason for passing reference as const, but there is more to it. For example, predict the output of following C++ program. Assume that **copy elision** is not done by compiler.

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

class Test
{
    /* Class data members */
public:
    Test(Test &t) { /* Copy data members from t*/}
    Test()       { /* Initialize data members */ }
};

Test fun()
{
    cout << "fun() Called\n";
    Test t;
    return t;
}

int main()
{
    Test t1;
    Test t2 = fun();
    return 0;
}
```

Output:

```
Compiler Error in line "Test t2 = fun();"

```

The program looks fine at first look, but it has compiler error. If we add const in copy constructor, the program works fine, i.e., we change copy constructor to following.

```
Test(const Test &t) { cout << "Copy Constructor Called\n"; }
```

Or if we change the line "Test t2 = fun();" to following two lines, then also the program works fine.

```
Test t2;  
t2 = fun();
```

The function fun() returns by value. So the compiler creates a temporary object which is copied to t2 using copy constructor in the original program (The temporary object is passed as an argument to copy constructor). The reason for compiler error is, compiler created temporary objects cannot be bound to non-const references and the original program tries to do that. It doesn't make sense to modify compiler created temporary objects as they can die any moment.

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

162. How to make a C++ class whose objects can only be dynamically allocated?

The problem is to create a class such that the non-dynamic allocation of object causes compiler error. For example, create a class 'Test' with following rules.

```
Test t1; // Should generate compiler error  
Test *t3 = new Test; // Should work fine
```

The idea is to create a **private destructor** in the class. When we make a private destructor, the compiler would generate a compiler error for non-dynamically allocated objects because compiler need to remove them from stack segment once they are not in use.

Since compiler is not responsible for deallocation of dynamically allocated objects (programmer should explicitly deallocate them), compiler won't have any problem with them. To avoid memory leak, we create a friend function *destructTest()* which can be called by users of class to destroy objects.

```

#include <iostream>
using namespace std;

// A class whose object can only be dynamically created
class Test
{
private:
    ~Test() { cout << "Destroying Object\n"; }
public:
    Test() { cout << "Object Created\n"; }
    friend void destructTest(Test* );
};

// Only this function can destruct objects of Test
void destructTest(Test* ptr)
{
    delete ptr;
    cout << "Object Destroyed\n";
}

int main()
{
    /* Uncommenting following following line would cause compiler error
    // Test t1;

    // create an object
    Test *ptr = new Test;

    // destruct the object to avoid memory leak
    destructTest(ptr);

    return 0;
}

```

Output:

```

Object Created
Destroying Object
Object Destroyed

```

If we don't want to create a friend function, we can also overload delete and delete[] operators in Test, this way we don't have to call a specific function to delete dynamically allocated objects.

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

163. 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

164. 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 explicitly, 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.

165. 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 contributed by **Vineet Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

166. Write a C program that won't compile in C++

Although C++ is designed to have backward compatibility with C there can be many C programs that would produce compiler error when compiled with a C++ compiler. Following are some of them.

1) In C++, it is a compiler error to call a function before it is declared. But in C, it may compile (See <http://www.geeksforgeeks.org/g-fact-95/>)

```
#include<stdio.h>
int main()
{
    foo(); // foo() is called before its declaration/definition
}

int foo()
{
    printf("Hello");
    return 0;
}
```

2) In C++, it is compiler error to make a normal pointer to point a const variable, but it is allowed in C. (See [Const Qualifier in C](#))

```
#include <stdio.h>

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

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

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

    return 0;
}
```

3) In C, a void pointer can directly be assigned to some other pointer like int *, char *. But in C++, a void pointer must be explicitly typecasted.

```
#include <stdio.h>
int main()
{
    void *vptr;
    int *iptr = vptr; //In C++, it must be replaced with int *iptr=(int
    return 0;
}
```

This is something we notice when we use malloc(). Return type of malloc() is void *. In C++, we must explicitly typecast return value of malloc() to appropriate type, e.g., "int *p = (void *)malloc(sizeof(int))". In C, typecasting is not necessary.

4) Following program compiles & runs fine in C, but fails in compilation in C++. const variable in C++ must be initialized but in c it isn't necessary. Thanks to Pravasi Meet for suggesting this point.

```
#include <stdio.h>
int main()
{
    const int a;    // LINE 4
    return 0;
}
```

Line 4 [Error] uninitialized const 'a' [-fpermissive]

5) This is the worst answer among all, but still a valid answer. We can use one of the C++ specific keywords as variable names. The program won't compile in C++, but would compile in C.

```
#include <stdio.h>
int main(void)
{
    int new = 5;    // new is a keyword in C++, but not in C
    printf("%d", new);
}
```

Similarly, we can use other keywords like delete, explicit, class, .. etc.

6) C++ does more strict type checking than C. For example the following program compiles in C, but not in C++. In C++, we get compiler error "invalid conversion from 'int' to 'char*'", Thanks to Pravasi Meet for adding this point.

```
#include <stdio.h>
int main()
{
    char *c = 333;
    printf("c = %u", c);
    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

167. Write a program that produces different results in C and C++

Write a program that compiles and runs both in C and C++, but produces different results when compiled by C and C++ compilers.

There can be many such programs, following are some of them.

1) Character literals are treated differently in C and C++. In C character literals like 'a', 'b', ...etc are treated as integers, while as characters in C++. (See [this](#) for details)

For example, the following program produces sizeof(int) as output in C, but sizeof(char) in C++.

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

2) In C, we need to use struct tag whenever we declare a struct variable. In C++, the struct tag is not necessary. For example, let there be a structure for *Student*. In C, we must use '*struct Student*' for *Student* variables. In C++, we can omit struct and use '*Student*' only.

Following is a program that is based on the fact and produces different outputs in C and C++. It prints sizeof(int) in C and sizeof(struct T) in C++.

```
#include <stdio.h>
int T;

int main()
{
    struct T { double x; }; // In C++, this T hides the global variable
                           // but not in C
    printf("%d", sizeof(T));
    return 0;
}
```

3) Types of boolean results are different in C and C++. Thanks to Gaurav Jain for suggesting this point.

```
// output = 4 in C (which is size of int)
printf("%d", sizeof(1==1));

// output = 1 in c++ (which is the size of boolean datatype)
cout << sizeof(1==1);
```

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

168. Can we access private data members of a class without using a member or a friend function?

The idea of **Encapsulation** is to bundle data and methods (that work on the data) together and restrict access of private data members outside the class. In C++, a friend

function or friend class can also access private data members.

Is it possible to access private members outside a class without friend?

Yes, it is possible using pointers. See the following program as an example.

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

class Test
{
private:
    int data;
public:
    Test() { data = 0; }
    int getData() { return data; }
};

int main()
{
    Test t;
    int* ptr = (int*)&t;
    *ptr = 10;
    cout << t.getData();
    return 0;
}
```

Output:

```
10
```

Note that the above way of accessing private data members is not at all a recommended way of accessing members and should never be used. Also, it doesn't mean that the encapsulation doesn't work in C++. The idea of making private members is to avoid accidental changes. The above change to data is not accidental. It's an intentionally written code to fool the compiler.

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

169. Template Specialization in C++

Template is a great feature in C++. We write code once and use it for any data type including user defined data types. For example, `sort()` can be written and used to sort any data type items. A class stack can be created that can be used as a stack of any data type.

What if we want a different code for a particular data type? Consider a big project that needs a function `sort()` for arrays of many different data types. Let Quick Sort be used for all datatypes except char. In case of char, total possible values are 256 and counting

sort may be a better option. Is it possible to use different code only when sort() is called for char data type?

It is possible in C++ to get a special behavior for a particular data type. This is called template specialization.

```
// A generic sort function
template <class T>
void sort(T arr[], int size)
{
    // code to implement Quick Sort
}

// Template Specialization: A function specialized for char data type
template <>
void sort<char>(char arr[], int size)
{
    // code to implement counting sort
}
```

Another example could be a class *Set* that represents a set of elements and supports operations like union, intersection, etc. When the type of elements is char, we may want to use a simple boolean array of size 256 to make a set. For other data types, we have to use some other complex technique.

An Example Program for function template specialization

For example, consider the following simple code where we have general template fun() for all data types except int. For int, there is a specialized version of fun().

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

template <class T>
void fun(T a)
{
    cout << "The main template fun(): " << a << endl;
}

template<>
void fun(int a)
{
    cout << "Specialized Template for int type: " << a << endl;
}

int main()
{
    fun<char>('a');
    fun<int>(10);
    fun<float>(10.14);
}
```

Output:

```
The main template fun(): a
Specialized Template for int type: 10
The main template fun(): 10.14
```


An Example Program for class template specialization

In the following program, a specialized version of class Test is written for int data type.

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

template <class T>
class Test
{
    // Data memnbers of test
public:
    Test()
    {
        // Initializstion of data memnbers
        cout << "General template object \n";
    }
    // Other methods of Test
};

template <>
class Test <int>
{
public:
    Test()
    {
        // Initializstion of data memnbers
        cout << "Specialized template object\n";
    }
};

int main()
{
    Test<int> a;
    Test<char> b;
    Test<float> c;
    return 0;
}
```

Output:

```
Specialized template object
General template object
General template object
```

How does template specialization work?

When we write any template based function or class, compiler creates a copy of that function/class whenever compiler sees that being used for a new data type or new set of data types(in case of multiple template arguments).

If a specialized version is present, compiler first checks with the specialized version and then the main template. Compiler first checks with the most specialized version by matching the passed parameter with the data type(s) specified in a specialized version.

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

170. Can virtual functions be private in C++?

In C++, virtual functions can be private and can be overridden by the derived class. For example, the following program compiles and runs fine.

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

class Derived;

class Base {
private:
    virtual void fun() { cout << "Base Fun"; }
friend int main();
};

class Derived: public Base {
public:
    void fun() { cout << "Derived Fun"; }
};

int main()
{
    Base *ptr = new Derived;
    ptr->fun();
    return 0;
}
```

Output:

```
Derived fun()
```

There are few things to note in the above program.

- 1) `ptr` is a pointer of `Base` type and points to a `Derived` class object. When `ptr->fun()` is called, `fun()` of `Derived` is executed.
- 2) `int main()` is a friend of `Base`. If we remove this friendship, the program won't compile (See [this](#)). We get `compiler error`.

Note that this behavior is totally different in Java. In Java, private methods are final by default and cannot be overridden (See [this](#))

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

171. Exception Handling in C++

One of the advantages of C++ over C is Exception Handling. C++ provides following specialized keywords for this purpose.

try: represents a block of code that can throw an exception.

catch: represents a block of code that is executed when a particular exception is thrown.

throw: Used to throw an exception. Also used to list the exceptions that a function throws, but doesn't handle itself.

Why Exception Handling?

Following are main advantages of exception handling over traditional error handling.

1) Separation of Error Handling code from Normal Code: In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.

2) Functions/Methods can handle any exceptions they choose: A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.

In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it)

3) Grouping of Error Types: In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

Exception Handling in C++

1) Following is a simple example to show exception handling in C++. The output of program explains flow of execution of try/catch blocks.

```

#include <iostream>
using namespace std;

int main()
{
    int x = -1;

    // Some code
    cout << "Before try \n";
    try {
        cout << "Inside try \n";
        if (x < 0)
        {
            throw x;
            cout << "After throw (Never executed) \n";
        }
    }
    catch (int x ) {
        cout << "Exception Caught \n";
    }

    cout << "After catch (Will be executed) \n";
    return 0;
}

```

Output:

```

Before try
Inside try
Exception Caught
After catch (Will be executed)

```

2) There is a special catch block called 'catch all' catch(...) that can be used to catch all types of exceptions. For example, in the following program, an int is thrown as an exception, but there is no catch block for int, so catch(...) block will be executed.

```

#include <iostream>
using namespace std;

int main()
{
    try {
        throw 10;
    }
    catch (char *excp) {
        cout << "Caught " << excp;
    }
    catch (...) {
        cout << "Default Exception\n";
    }
    return 0;
}

```

Output:

```

Default Exception

```

3) Implicit type conversion doesn't happen for primitive types. For example, in the

following program 'a' is not implicitly converted to int

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

int main()
{
    try {
        throw 'a';
    }
    catch (int x) {
        cout << "Caught " << x;
    }
    catch (...) {
        cout << "Default Exception\n";
    }
    return 0;
}
```

Output:

```
Default Exception
```

4) If an exception is thrown and not caught anywhere, the program terminates abnormally. For example, in the following program, a char is thrown, but there is no catch block to catch a char.

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

int main()
{
    try {
        throw 'a';
    }
    catch (int x) {
        cout << "Caught ";
    }
    return 0;
}
```

Output:

```
terminate called after throwing an instance of 'char'
```

```
This application has requested the Runtime to terminate it in an
unusual way. Please contact the application's support team for
more information.
```

We can change this abnormal termination behavior by **writing our own unexpected function**.

5) A derived class exception should be caught before a base class exception. See [this](#) for more details.

6) Like Java, C++ library has a **standard exception class** which is base class for all

standard exceptions. All objects thrown by components of the standard library are derived from this class. Therefore, all standard exceptions can be caught by catching this type

7) Unlike Java, in C++, all exceptions are unchecked. Compiler doesn't check whether an exception is caught or not (See [this](#) for details). For example, in C++, it is not necessary to specify all uncaught exceptions in a function declaration. Although it's a recommended practice to do so. For example, the following program compiles fine, but ideally signature of fun() should list unchecked exceptions.

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

// This function signature is fine by the compiler, but not recommended
// Ideally, the function should specify all uncaught exceptions and fun
// signature should be "void fun(int *ptr, int x) throw (int *, int)"
void fun(int *ptr, int x)
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
    /* Some functionality */
}

int main()
{
    try {
        fun(NULL, 0);
    }
    catch(...) {
        cout << "Caught exception from fun()";
    }
    return 0;
}
```

Output:

```
Caught exception from fun()
```

8) In C++, try-catch blocks can be nested. Also, an exception can be re-thrown using "throw;"

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

int main()
{
    try {
        try {
            throw 20;
        }
        catch (int n) {
            cout << "Handle Partially ";
            throw; //Re-throwing an exception
        }
    }
    catch (int n) {
        cout << "Handle remaining ";
    }
    return 0;
}
```

Output:

```
Handle Partially Handle remaining
```

A function can also re-throw a function using same “throw; “. A function can handle a part and can ask the caller to handle remaining.

9) When an exception is thrown, all objects created inside the enclosing try block are destructed before the control is transferred to catch block.

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

class Test {
public:
    Test() { cout << "Constructor of Test " << endl; }
    ~Test() { cout << "Destructor of Test " << endl; }
};

int main() {
    try {
        Test t1;
        throw 10;
    } catch(int i) {
        cout << "Caught " << i << endl;
    }
}
```

Output:

```
Constructor of Test
Destructor of Test
Caught 10
```

10) You may like to try [Quiz on Exception Handling in C++](#).

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

172. Object Slicing in C++

In C++, a derived class object can be assigned to a base class object, but the other way is not possible.

```
class Base { int x, y; };  
  
class Derived : public Base { int z, w; };  
  
int main()  
{  
    Derived d;  
    Base b = d; // Object Slicing, z and w of d are sliced off  
}
```

Object slicing happens when a derived class object is assigned to a base class object, additional attributes of a derived class object are sliced off to form the base class object.


```

#include <iostream>
using namespace std;

class Base
{
protected:
    int i;
public:
    Base(int a)    { i = a; }
    virtual void display()
    { cout << "I am Base class object, i = " << i << endl; }
};

class Derived : public Base
{
    int j;
public:
    Derived(int a, int b) : Base(a) { j = b; }
    virtual void display()
    { cout << "I am Derived class object, i = "
        << i << ", j = " << j << endl; }
};

// Global method, Base class object is passed by value
void somefunc (Base obj)
{
    obj.display();
}

int main()
{
    Base b(33);
    Derived d(45, 54);
    somefunc(b);
    somefunc(d); // Object Slicing, the member j of d is sliced off
    return 0;
}

```

Output:

```

I am Base class object, i = 33
I am Base class object, i = 45

```

We can avoid above unexpected behavior with the use of pointers or references. Object slicing doesn't occur when pointers or references to objects are passed as function arguments since a pointer or reference of any type takes same amount of memory. For example, if we change the global method myfunc() in the above program to following, object slicing doesn't happen.

```

// rest of code is similar to above
void somefunc (Base &obj)
{
    obj.display();
}
// rest of code is similar to above

```

Output:

```

I am Base class object, i = 33

```

```
I am Derived class object, i = 45, j = 54
```

We get the same output if we use pointers and change the program to following.

```
// rest of code is similar to above
void somefunc (Base *objp)
{
    objp->display();
}

int main()
{
    Base *bp = new Base(33) ;
    Derived *dp = new Derived(45, 54);
    somefunc(bp);
    somefunc(dp); // No Object Slicing
    return 0;
}
```

Output:

```
I am Base class object, i = 33
I am Derived class object, i = 45, j = 54
```

Object slicing can be prevented by making the base class function pure virtual there by disallowing object creation. It is not possible to create the object of a class which contains a pure virtual method.

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

173. 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

174. Does overloading work with Inheritance?

If we have a function in base class and a function with same name in derived class, can the base class function be called from derived class object? This is an interesting question and as an experiment predict the output of the following **C++** program.

```

#include <iostream>
using namespace std;
class Base
{
public:
    int f(int i)
    {
        cout << "f(int): ";
        return i+3;
    }
};
class Derived : public Base
{
public:
    double f(double d)
    {
        cout << "f(double): ";
        return d+3.3;
    }
};
int main()
{
    Derived* dp = new Derived;
    cout << dp->f(3) << '\n';
    cout << dp->f(3.3) << '\n';
    delete dp;
    return 0;
}

```

The output of this program is:

```

f(double): 6.3
f(double): 6.6

```

Instead of the supposed output:

```

f(int): 6
f(double): 6.6

```

Overloading doesn't work for derived class in C++ programming language. There is no overload resolution between Base and Derived. The compiler looks into the scope of Derived, finds the single function “double f(double)” and calls it. It never disturbs with the (enclosing) scope of Base. In C++, there is no overloading across scopes – derived class scopes are not an exception to this general rule. (See [this](#) for more examples)

Reference: [technical FAQs on www.stroustrup.com](http://www.stroustrup.com)

Now consider **Java** version of this program:

```

class Base
{
    public int f(int i)
    {
        System.out.print("f (int): ");
        return i+3;
    }
}

```

```

    }
}
class Derived extends Base
{
    public double f(double i)
    {
        System.out.print("f (double) : ");
        return i + 3.3;
    }
}
class myprogram3
{
    public static void main(String args[])
    {
        Derived obj = new Derived();
        System.out.println(obj.f(3));
        System.out.println(obj.f(3.3));
    }
}

```

The output of the above program is:

```

f (int): 6
f (double): 6.6

```

So in Java overloading works across scopes contrary to C++. Java compiler determines correct version of the overloaded method to be executed at compile time based upon the type of argument used to call the method and parameters of the overloaded methods of both these classes receive the values of arguments used in call and executes the overloaded method.

Finally, let us try the output of following **C#** program:

```

using System;
class Base
{
    public int f(int i)
    {
        Console.WriteLine("f (int): ");
        return i + 3;
    }
}
class Derived : Base
{
    public double f(double i)
    {
        Console.WriteLine("f (double) : ");
        return i+3.3;
    }
}
class MyProgram
{
    static void Main(string[] args)
    {
        Derived obj = new Derived();
        Console.WriteLine(obj.f(3));
        Console.WriteLine(obj.f(3.3));
        Console.ReadKey(); // write this line if you use visual studio
    }
}

```

Note: Console.ReadKey() is used to halt the console. It is similar to getch as in C/C++. The output of the above program is:

```

f(double) : 6.3
f(double): 6.6

```

instead of the assumed output

```

f(int) : 6
f(double) : 6.6

```

The reason is same as explained in C++ program. Like C++, there is no overload resolution between class Base and class Derived. In C#, there is no overloading across scopes derived class scopes are not an exception to this general rule. This is same as C++ because C# is designed to be much closer to C++, according to the [Anders hejlsberg](#) the creator of C# language.

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

175. Can main() be overloaded in C++?

Predict the output of following C++ program.

```
#include <iostream>
using namespace std;
int main(int a)
{
    cout << a << "\n";
    return 0;
}
int main(char *a)
{
    cout << a << endl;
    return 0;
}
int main(int a, int b)
{
    cout << a << " " << b;
    return 0;
}
int main()
{
    main(3);
    main("C++");
    main(9, 6);
    return 0;
}
```

The above program fails in compilation and produces warnings and errors (See [this](#) for produced warnings and errors). You may get different errors on different compilers.

To overload main() function in C++, it is necessary to use class and declare the main as member function. Note that main is not reserved word in programming languages like C, C++, Java and C#. For example, we can declare a variable whose name is main, try below example:

```
#include <iostream>
int main()
{
    int main = 10;
    std::cout << main;
    return 0;
}
```

Output:

```
10
```

The following program shows overloading of main() function in a class.

```

#include <iostream>
using namespace std;
class Test
{
public:
    int main(int s)
    {
        cout << s << "\n";
        return 0;
    }
    int main(char *s)
    {
        cout << s << endl;
        return 0;
    }
    int main(int s ,int m)
    {
        cout << s << " " << m;
        return 0;
    }
};
int main()
{
    Test obj;
    obj.main(3);
    obj.main("I love C++");
    obj.main(9, 6);
    return 0;
}

```

The outcome of program is:

```

3
I love C++
9 6

```

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

176. 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

177. Virtual Functions and Runtime Polymorphism in C++ | Set 1 (Introduction)

Consider the following simple program which is an example of runtime polymorphism.

```

#include<iostream>
using namespace std;

class Base
{
public:
    virtual void show() { cout<<" In Base \n"; }
};

class Derived: public Base
{
public:
    void show() { cout<<"In Derived \n"; }
};

int main(void)
{
    Base *bp = new Derived;
    bp->show(); // RUN-TIME POLYMORPHISM
    return 0;
}

```

Output:

```
In Derived
```

The main thing to note about the above program is, derived class function is called using a base class pointer. The idea is, **virtual functions** are called according to the type of object pointed or referred, not according to the type of pointer or reference. In other words, virtual functions are resolved late, at runtime.

What is the use?

Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object. For example, consider a employee management software for an organization, let the code has a simple base class *Employee* , the class contains virtual functions like *raiseSalary()*, *transfer()*, *promote()*,... etc. Different types of employees like *Manager*, *Engineer*, ..etc may have their own implementations of the virtual functions present in base class *Employee*. In our complete software, we just need to pass a list of employees everywhere and call appropriate functions without even knowing the type of employee. For example, we can easily raise salary of all employees by iterating through list of employees. Every type of employee may have its own logic in its class, we don't need to worry because if *raiseSalary()* is present for a specific employee type, only that function would be called.

```

class Employee
{
public:
    virtual void raiseSalary()
    { /* common raise salary code */ }

    virtual void promote()
    { /* common promote code */ }
};

class Manager: public Employee {
    virtual void raiseSalary()
    { /* Manager specific raise salary code, may contain
        increment of manager specific incentives*/ }

    virtual void promote()
    { /* Manager specific promote */ }
};

// Similarly, there may be other types of employees

// We need a very simple function to increment salary of all employees
// Note that emp[] is an array of pointers and actual pointed objects may
// be any type of employees. This function should ideally be in a class
// like Organization, we have made it global to keep things simple
void globalRaiseSalary(Employee *emp[], int n)
{
    for (int i = 0; i < n; i++)
        emp[i]->raiseSalary(); // Polymorphic Call: Calls raiseSalary()
                                // according to the actual object, not
                                // according to the type of pointer
}

```

like *globalRaiseSalary()*, there can be many other operations that can be appropriately done on a list of employees without even knowing the type of actual object.

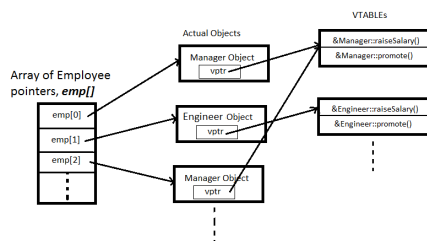
Virtual functions are so useful that later languages like **Java** keep all methods as virtual by default.

How does compiler do this magic of late resolution?

Compiler maintains two things to this magic:

vtable: A table of function pointers. It is maintained per class.

vpitr: A pointer to vtable. It is maintained per object (See [this](#) for an example).



Compiler adds additional code at two places to maintain and use *vpitr*.

- 1) Code in every constructor. This code sets *vpitr* of the object being created. This code sets *vpitr* to point to *vtable* of the class.
- 2) Code with polymorphic function call (e.g. *bp->show()* in above code). Wherever a polymorphic call is made, compiler inserts code to first look for *vpitr* using base class pointer or reference (In the above example, since pointed or referred object is of derived

type, *vptr* of derived class is accessed). Once *vptr* is fetched, *vtable* of derived class can be accessed. Using *vtable*, address of derived class function *show()* is accessed and called.

Is this a standard way for implementation of run-time polymorphism in C++?

The C++ standards do not mandate exactly how runtime polymorphism must be implemented, but compilers generally use minor variations on the same basic model.

Quiz on Virtual Functions.

References:

http://en.wikipedia.org/wiki/Virtual_method_table

http://en.wikipedia.org/wiki/Virtual_function

<http://www.drbio.cornell.edu/pl47/programming/TICPP-2nd-ed-Vol-one-html/Frames.html>

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

178. 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

179. 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.

180. 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

181. 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 associativity. 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 associativity. 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.

182. How to restrict dynamic allocation of objects in C++?

C++ programming language allows both auto(or stack allocated) and dynamically allocated objects. In java & C#, all objects must be dynamically allocated using new. C++ supports stack allocated objects for the reason of runtime efficiency. Stack based objects are implicitly managed by C++ compiler. They are destroyed when they go out of scope and dynamically allocated objects must be manually released, using delete operator otherwise memory leak occurs. C++ doesn't support automatic garbage collection approach used by languages such as Java & C#.

How do we achieve following behavior from a class 'Test' in C++?

```
Test* t = new Test; // should produce compile time error
Test t;           // OK
```

The idea of is to keep new operator function private so that new cannot be called. See the following program. Objects of 'Test' class cannot be created using new as new operator function is private in 'Test'. If we uncomment the 2nd line of main(), the program would produce compile time error.

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

// Objects of Test can not be dynamically allocated
class Test
{
    // Notice this, new operator function is private
    void* operator new(size_t size);
    int x;
public:
    Test()          { x = 9; cout << "Constructor is called\n"; }
    void display()  { cout << "x = " << x << "\n"; }
    ~Test()         { cout << "Destructor is executed\n"; }
};

int main()
{
    // Uncommenting following line would cause a compile time error.
    // Test* obj=new Test();
    Test t;          // OK, object is allocated at compile time
    t.display();
    return 0;
} // object goes out of scope, destructor will be called
```

Output:

```
Constructor is called
x = 9
Destructor is executed
```

Reference:

[Design and evolution of C++, Bjarne Stroustrup](#)

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.

183. 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

184. 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 **Left to Right** or **Right to Left**. For example '*' and '/' have same precedence and their associativity is **Left to Right**, 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) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2)	left-to-right
++ -- + - ! ~ (type) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of type) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left
* / %	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 Relational greater than/greater than or equal to	left-to-right
== !=	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 Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	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

185. 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;

    // sizeof(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("*(ptr + 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
*(ptr + 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

186. 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 string 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); // Expanded 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

187. 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.

<< (left shift) 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 = 18
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 << and >> is undefined behaviour if any of the operands is a negative number. For example results of both -1 << 1 and 1 << -1 is undefined. Also, if the number is shifted more than the size of integer, the behaviour is undefined. For example, 1 << 33 is undefined if integers are stored using 32 bits. See [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 occurring 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

188. 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 above 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

189. 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/AQTLTB

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

190. 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\"", 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

191. 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
```

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

192. 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 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.

1) 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.

2) 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;
    print((int *)arr, m, n);
    return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9
```

3) Using an array of pointers or double pointer

In this method also, we must typecast the 2D array when passing to function.

```
#include <stdio.h>
// Same as "void print(int **arr, int m, int n)"
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;
    int n = 3;
    print((int **)arr, m, n);
    return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9
```

Remember that in C, array parameters are treated as pointers, so an array of pointers or a double pointer are same when they are parameters. For dynamic allocation, we had 2 different methods for array of pointers and double pointer. In [dynamic allocation post](#), we had 2 different methods for array of pointers and double pointer.

Methods 2 and 3 can also be useful on a C99 compatible compiler, in a situation when

2D array is dynamically allocated using malloc.

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

193. Can virtual functions be inlined?

Virtual functions are used to achieve runtime polymorphism or say late binding or dynamic binding. **Inline functions** are used for efficiency. The whole idea behind the inline functions is that whenever inline function is called code of inline function gets inserted or substituted at the point of inline function call at compile time. Inline functions are very useful when small functions are frequently used and called in a program many times.

By default all the functions defined inside the class are implicitly or automatically considered as inline except virtual functions (Note that inline is a request to the compiler and its compilers choice to do inlining or not).

Whenever virtual function is called using base class reference or pointer it cannot be inlined (because call is resolved at runtime), but whenever called using the object (without reference or pointer) of that class, can be inlined because compiler knows the exact class of the object at compile time.

```

#include <iostream>
using namespace std;
class Base
{
public:
    virtual void who()
    {
        cout << "I am Base\n";
    }
};
class Derived: public Base
{
public:
    void who()
    {
        cout << "I am Derived\n";
    }
};

int main()
{
    // note here virtual function who() is called through
    // object of the class (it will be resolved at compile
    // time) so it can be inlined.
    Base b;
    b.who();

    // Here virtual function is called through pointer,
    // so it cannot be inlined
    Base *ptr = new Derived();
    ptr->who();

    return 0;
}

```

References:

<http://www.parashift.com/c++-faq/inline-virtuals.html>

Effective C++, by Scott Meyers

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

194. 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 can 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

195. When are static objects destroyed?

*Remain Careful from these two persons
new friends and old enemies — Kabir*

What is static keyword in C++?

static keyword can be applied to local variables, functions, class' data members and objects in C++. static local variable retain their values between function call and initialized only once. static function can be directly called using the scope resolution operator preceded by class name (See [this](#), [this](#) and [this](#) for more details). C++ also supports static objects.

What are static objects in C++?

An object become static when static keyword is used in its declaration. See the following

two statements for example in C++.

```
Test t;           // Stack based object
static Test t1;   // Static object
```

First statement when executes creates object on stack means storage is allocated on stack. Stack based objects are also called automatic objects or local objects. static object are initialized only once and live until the program terminates. Local object is created each time its declaration is encountered in the execution of program.

static objects are allocated storage in static storage area. static object is destroyed at the termination of program. C++ supports both local static object and global static object. Following is example that shows use of local static object.

```
#include <iostream>
class Test
{
public:
    Test()
    {
        std::cout << "Constructor is executed\n";
    }
    ~Test()
    {
        std::cout << "Destructor is executed\n";
    }
};
void myfunc()
{
    static Test obj;
} // Object obj is still not destroyed because it is static

int main()
{
    std::cout << "main() starts\n";
    myfunc();    // Destructor will not be called here
    std::cout << "main() terminates\n";
    return 0;
}
```

Output:

```
main() starts
Constructor is executed
main() terminates
Destructor is executed
```

If we observe the output of this program closely, we can see that the destructor for the local object named obj is not called after it's constructor is executed because the local object is static so it has scope till the lifetime of program so it's destructor will be called when main() terminates.

What happens when we remove static in above program?

As an experiment if we remove the static keyword from the global function myfunc(), we

get the output as below:

```
main() starts
Constructor is called
Destructor is called
main() terminates
```

This is because the object is now stack based object and it is destroyed when it is goes out of scope and its destructor will be called.

How about global static objects?

The following program demonstrates use of global static object.

```
#include <iostream>
class Test
{
public:
    int a;
    Test()
    {
        a = 10;
        std::cout << "Constructor is executed\n";
    }
    ~Test()
    {
        std::cout << "Destructor is executed\n";
    }
};
static Test obj;
int main()
{
    std::cout << "main() starts\n";
    std::cout << obj.a;
    std::cout << "\nmain() terminates\n";
    return 0;
}
```

Output:

```
Constructor is executed
main() starts
10
main() terminates
Destructor is executed
```

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

Consider the following simple C++ code with normal pointers.

```
MyClass *ptr = new MyClass();
ptr->doSomething();
// We must do delete(ptr) to avoid memory leak
```

Using **smart pointers**, we can make pointers to work in way that we don't need to explicitly call delete. **Smart pointer** is a wrapper class over a pointer with operator like * and -> overloaded. The objects of smart pointer class look like pointer, but can do many things that a normal pointer can't like automatic destruction (yes, we don't have to explicitly use delete), reference counting and more.

The idea is to make a class with a pointer, destructor and **overloaded operators** like * and ->. Since destructor is automatically called when an object goes out of scope, the dynamically allocated memory would automatically be deleted (or reference count can be decremented). Consider the following simple smartPtr class.

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

class SmartPtr
{
    int *ptr; // Actual pointer
public:
    // Constructor: Refer http://www.geeksforgeeks.org/g-fact-93/
    // for use of explicit keyword
    explicit SmartPtr(int *p = NULL) { ptr = p; }

    // Destructor
    ~SmartPtr() { delete(ptr); }

    // Overloading dereferencing operator
    int &operator *() { return *ptr; }
};

int main()
{
    SmartPtr ptr(new int());
    *ptr = 20;
    cout << *ptr;

    // We don't need to call delete ptr: when the object
    // ptr goes out of scope, destructor for it is automatically
    // called and destructor does delete ptr.

    return 0;
}
```

Output:

```
20
```

Can we write one smart pointer class that works for all types?

Yes, we can use **templates** to write a generic smart pointer class. Following C++ code demonstrates the same.


```

#include<iostream>
using namespace std;

// A generic smart pointer class
template <class T>
class SmartPtr
{
    T *ptr; // Actual pointer
public:
    // Constructor
    explicit SmartPtr(T *p = NULL) { ptr = p; }

    // Destructor
    ~SmartPtr() { delete(ptr); }

    // Overloading dereferencing operator
    T & operator * () { return *ptr; }

    // Overloading arrow operator so that members of T can be accessed
    // like a pointer (useful if T represents a class or struct or
    // union type)
    T * operator -> () { return ptr; }
};

int main()
{
    SmartPtr<int> ptr(new int());
    *ptr = 20;
    cout << *ptr;
    return 0;
}

```

Output:

20

Smart pointers are also useful in management of resources, such as file handles or network sockets.

C++ libraries provide implementations of smart pointers in the form of `auto_ptr`, `unique_ptr`, `shared_ptr` and `weak_ptr`

References:

http://en.wikipedia.org/wiki/Smart_pointer

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

197. Can we use % operator on floating point numbers?

Predict the output of following program:

Can % be used with floating point numbers in C++?

```
#include <iostream>
int main()
{
    float f = 9.9f, m = 3.3f;
    float c = f % m; // LINE 5
    std::cout << c;
    return 0;
}
```

The above program fails in compilation and compiler report the following error in line 5:
Output:

```
invalid operands of types 'float' and 'float'
to binary 'operator%'
```

% operator cannot be used with floating point numbers in C & C++.

What about Java and C#?

This behavior is different in Java & C#. % operator can be used on floating point numbers in these languages.

Consider following example of **Java** program:

```
class Test
{
    public static void main(String args[])
    {
        float f = 9.9f, m = 3.3f;
        float c = f % m;
        System.out.println(c);
    }
}
```

Output:

```
3.2999997
```

Same way try this **C#** program. It works fine:

```
using System;
class Test
{
    public static void Main()
    {
        float f = 9.9f, m = 3.3f;
        float c = f % m;
        Console.WriteLine(c);
    }
}
```

Output:

3.3

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

198. Print substring of a given string without using any string function and loop in C

Write a function `mysubstr()` in C that doesn't use any string function, doesn't use any loop and prints substring of a string. *The function should not modify contents of string and should not use a temporary char array or string.*

For example `mysubstr("geeksforgeeks", 1, 3)` should print "eek" i.e., the substring between indexes 1 and 3.

We strongly recommend to minimize the browser and try this yourself first.

One solution is to use recursion. Thanks to Gopi and oggy for suggesting this solution.

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

```
// This function prints substring of str[] between low and  
// high indexes (both inclusive).
```

```
void mysubstr(char str[], int low, int high)  
{  
    if (low<=high)  
    {  
        printf("%c", str[low]);  
        mysubstr(str, low+1, high);  
    }  
}
```

```
int main ()  
{  
    char str[] = "geeksforgeeks";  
    mysubstr(str, 1, 3);  
    return 0;  
}
```

Output:

eek

How to do it if recursions is also not allowed?

We can always use pointer arithmetic to change the beginning part. For example (`str + i`)

gives us address of i'th character. To limit the ending, we can use width specifier in printf which can be passed as an argument when * is used in format string.

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

```
// This function prints substring of str[] between low and  
// high indexes (both inclusive).  
void mysubstr(char str[], int low, int high)  
{  
    printf("%.s", high-low+1, (str+low));  
}
```

```
int main ()  
{  
    char str[] = "geeksforgeeks";  
    mysubstr(str, 1, 3);  
    return 0;  
}
```

Output:

```
ee k
```

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

199. C++ final specifier

```
Cast all your worries on him,  
because he cares for you.  
1 Peter 5: 7 (Bible)
```

In Java, we can use **final** for a function to make sure that it cannot be overridden. We can also use final in Java to make sure that a class cannot be inherited. Similarly, the latest C++ standard **C++ 11** added final.

Use of final specifier in C++ 11:

Sometimes you don't want to allow derived class to override the base class' virtual function. **C++ 11** allows built-in facility to prevent overriding of virtual function using final specifier.

Consider the following example which shows use of final specifier. This program fails in compilation.

```

#include <iostream>
using namespace std;

class Base
{
public:
    virtual void myfun() final
    {
        cout << "myfun() in Base";
    }
};

class Derived : public Base
{
    void myfun()
    {
        cout << "myfun() in Derived\n";
    }
};

int main()
{
    Derived d;
    Base &b = d;
    b.myfun();
    return 0;
}

```

Output:

```

prog.cpp:14:10: error: virtual function 'virtual void Derived::myfun()'
        void myfun()
            ^
prog.cpp:7:18: error: overriding final function 'virtual void Base::myfun()'
        virtual void myfun() final

```

2nd use of final specifier:

final specifier in C++ 11 can also be used to prevent inheritance of class / struct. If a class or struct is marked as final then it becomes non inheritable and it cannot be used as base class/struct.

The following program shows use of final specifier to make class non inheritable:

```

#include <iostream>
class Base final
{
};

class Derived : public Base
{
};

int main()
{
    Derived d;
    return 0;
}

```

Output:

```
error: cannot derive from 'final' base 'Base' in derived type 'Derived'
class Derived : public Base
```

final in C++ 11 vs in Java

Note that use of final specifier in C++ 11 is same as in Java but Java uses final before the class name while final specifier is used after the class name in C++ 11. Same way Java uses final keyword in the beginning of method definition (Before the return type of method) but C++ 11 uses final specifier after the function name.

```
class Test
{
    final void fun()// use of final in Java
    { }
}
class Test
{
public:
    virtual void fun() final //use of final in C++ 11
    {}
};
```

Unlike Java, final is not a keyword in C++ 11. final has meaning only when used in above contexts, otherwise it's just an identifier.

One possible reason to not make final a keyword is to ensure backward compatibility. There may exist production codes which use final for other purposes. For example the following program compiles and runs without error.

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

int main()
{
    int final = 20;
    cout << final;
    return 0;
}
```

Output:

```
20
```

In java, final can also be used with variables to make sure that a value can only be assigned once. this use of final is not there in C++ 11.

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

200. How to print range of basic data types without any library function and constant in C?

How to write C code to print range of basic data types like int, char, short int, unsigned int, unsigned char etc?

It is assumed that signed numbers are stored in 2's complement form.

We strongly recommend to minimize the browser and try this yourself first.

Following are the steps to be followed for unsigned data types.

- 1) Find number of bytes for a given data type using sizeof operator.
- 2) Find number of bits by multiplying result of sizeof with 8.
- 3) The minimum value for an unsigned type is always 0 irrespective of data type.
- 4) The maximum value of an unsigned type is $(1 \ll n) - 1$ where n is number of bits needed in data type. For example for char which typically requires 8 bits, the maximum value is 255.

Following are the steps to be followed for signed data type.

- 1) Find number of bytes for a given data type using sizeof operator.
- 2) Find number of bits by multiplying result of sizeof with 8.
- 3) The minimum value for a signed type is $-(1 \ll (n-1))$. For example for char which typically requires 8 bits, the minimum value is -128.
- 4) The maximum value of a data type is $(1 \ll (n-1)) - 1$ where n is number of bits needed in data type. For example for char which typically requires 8 bits, the maximum value is 127.

Following is C code to demonstrate above idea.

```
// C program to print range of basic data types
#include <stdio.h>
```

```
// Prints min and max value for a signed type
void printUnsignedRange(size_t bytes)
{
    int bits = 8*bytes;

    // Note that the value of 'to' is "(1 << bits) - 1"
    // Writing it in following way doesn't cause overflow
    unsigned int to = ((1 << (bits-1)) - 1) + (1 << (bits-1)) ;

    printf(" range is from %u to %u \n", 0, to);
}

// Prints min and max value for an unsigned type
void printSignedRange(size_t bytes)
{
    int bits = 8*bytes;
    int from = -(1 << (bits-1));
    int to = (1 << (bits-1)) - 1;
    printf(" range is from %d to %d\n", from, to);
}
```

```
int main()
{
    printf("signed char: ");
    printSignedRange(sizeof(char));

    printf("unsigned char: ");
    printUnsignedRange(sizeof(unsigned char));

    printf("signed int: ");
    printSignedRange(sizeof(int));

    printf("unsigned int: ");
    printUnsignedRange(sizeof(unsigned int));

    printf("signed short int: ");
    printSignedRange(sizeof(short int));

    printf("unsigned short int: ");
    printUnsignedRange(sizeof(unsigned short int));

    return 0;
}
```

Output:

```
signed char:  range is from -128 to 127
unsigned char:  range is from 0 to 255
signed int:  range is from -2147483648 to 2147483647
unsigned int:  range is from 0 to 4294967295
signed short int:  range is from -32768 to 32767
unsigned short int:  range is from 0 to 65535
```

Note that the above functions cannot be used for float. Also, the above program may not work for data types bigger than int, like 'long long int'. We can make it work for bigger

types by changing data type of 'to' and 'from' to long long int.

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

201. 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?

A thread has its own program counter (PC), a register set, and a stack space. 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.

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:~/ $

```

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

202. Is it possible to call constructor and destructor explicitly?

```
All the days of the afflicted are evil but he
that is of a merry heart hath a continual feast.
Proverbs 15:15 (Bible)
```

Constructor is a special member function that is automatically called by compiler when object is created and **destructor** is also special member function that is also implicitly called by compiler when object goes out of scope. They are also called when dynamically allocated object is allocated and destroyed, new operator allocates storage and calls constructor, delete operator calls destructor and free the memory allocated by new.

Is it possible to call constructor and destructor explicitly?

Yes, it is possible to call special member functions explicitly by programmer. Following program calls constructor and destructor explicitly.

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

class Test
{
public:
    Test() { cout << "Constructor is executed\n"; }
    ~Test() { cout << "Destructor is executed\n"; }
};

int main()
{
    Test(); // Explicit call to constructor
    Test t; // local object
    t.~Test(); // Explicit call to destructor
    return 0;
}
```

Output:

```
Constructor is executed
Destructor is executed
Constructor is executed
Destructor is executed
Destructor is executed
```

When the constructor is called explicitly the compiler creates a nameless temporary object and it is immediately destroyed. That's why 2nd line in the output is call to destructor.

Here is a conversation between me and Dr. Bjarne Stroustrup via mail about this topic:

Me: Why C++ allows to call constructor explicitly? Don't you think that this shouldn't be?

Dr. Bjarne: No. temporary objects of the class types are useful.

If the destructor is called explicitly on local object it doesn't mean that the object is destroyed. Local objects are automatically destroyed by compiler when they go out of scope and this is the guarantee of C++ language. In general, special member functions shouldn't be called explicitly.

Constructor and destructor can also be called from the member function of class. See following program:

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

class Test
{
public:
    Test() { cout << "Constructor is executed\n"; }
    ~Test() { cout << "Destructor is executed\n"; }
    void show() { Test(); this->Test::~~Test(); }
};

int main()
{
    Test t;
    t.show();
    return 0;
}
```

Output:

```
Constructor is executed
Constructor is executed
Destructor is executed
Destructor is executed
Destructor is executed
```

Explicit call to destructor is only necessary when object is placed at particular location in memory by using placement new. Destructor should not be called explicitly when the object is dynamically allocated because delete operator automatically calls destructor.

As an exercise predict the output of following program:

```

#include <iostream>
using namespace std;

class Test
{
public:
    Test() { cout << "Constructor is executed\n"; }
    ~Test() { cout << "Destructor is executed\n"; }
    friend void fun(Test t);
};

void fun(Test t)
{
    Test();
    t.~Test();
}

int main()
{
    Test();
    Test t;
    fun(t);
    return 0;
}

```

Sources:

<http://www.parashift.com/c++-faq/dont-call-dtor-on-local.html>

<http://www.parashift.com/c++-faq/placement-new.html>

<http://msdn.microsoft.com/en-us/library/35xa3368.aspx>

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

203. C++ mutable keyword

Be self controlled and alert. Your enemy the devil prowls around like a roaring lion looking for someone to devour.
1 Peter 5:8 (Bible)

The mutable storage class specifier in C++ (or use of mutable keyword in C++)

auto, register, static and extern are the storage class specifiers in C. typedef is also considered as a storage class specifier in C. C++ also supports all these storage class specifiers. In addition to this C++, adds one important storage class specifier whose name is mutable.

What is the need of mutable?

Sometimes there is requirement to modify one or more data members of class / struct through const function even though you don't want the function to update other members

of class / struct. This task can be easily performed by using mutable keyword. Consider this example where use of mutable can be useful. Suppose you go to hotel and you give the order to waiter to bring some food dish. After giving order, you suddenly decide to change the order of food. Assume that hotel provides facility to change the ordered food and again take the order of new food within 10 minutes after giving the 1st order. After 10 minutes order can't be cancelled and old order can't be replaced by new order. See the following code for details.

```
#include <iostream>
#include <string.h>
using std::cout;
using std::endl;

class Customer
{
    char name[25];
    mutable char placedorder[50];
    int tableno;
    mutable int bill;
public:
    Customer(char* s, char* m, int a, int p)
    {
        strcpy(name, s);
        strcpy(placedorder, m);
        tableno = a;
        bill = p;
    }
    void changePlacedOrder(char* p) const
    {
        strcpy(placedorder, p);
    }
    void changeBill(int s) const
    {
        bill = s;
    }
    void display() const
    {
        cout << "Customer name is: " << name << endl;
        cout << "Food ordered by customer is: " << placedorder << endl;
        cout << "table no is: " << tableno << endl;
        cout << "Total payable amount: " << bill << endl;
    }
};

int main()
{
    const Customer c1("Pravasi Meet", "Ice Cream", 3, 100);
    c1.display();
    c1.changePlacedOrder("GulabJammuns");
    c1.changeBill(150);
    c1.display();
    return 0;
}
```

Output:

```
Customer name is: Pravasi Meet
Food ordered by customer is: Ice Cream
```

```
table no is: 3
Total payable amount: 100
Customer name is: Pravasi Meet
Food ordered by customer is: GulabJammuns
table no is: 3
Total payable amount: 150
```

Closely observe the output of above program. The values of *placedorder* and *bill* data members are changed from const function because they are declared as mutable.

The keyword mutable is mainly used to allow a particular data member of const object to be modified. When we declare a function as const, the this pointer passed to function becomes const. Adding mutable to a variable allows a const pointer to change members.

mutable is particularly useful if most of the members should be constant but a few need to be updateable. Data members declared as mutable can be modified even though they are the part of object declared as const. You cannot use the mutable specifier with names declared as static or const, or reference.

As an **exercise** predict the output of following two programs.

```
// PROGRAM 1
#include <iostream>
using std::cout;

class Test {
public:
    int x;
    mutable int y;
    Test() { x = 4; y = 10; }
};
int main()
{
    const Test t1;
    t1.y = 20;
    cout << t1.y;
    return 0;
}
```

```
// PROGRAM 2
#include <iostream>
using std::cout;

class Test {
public:
    int x;
    mutable int y;
    Test() { x = 4; y = 10; }
};
int main()
{
    const Test t1;
    t1.x = 8;
    cout << t1.x;
    return 0;
}
```


Source:

The mutable storage class specifier (C++ only)

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

204. Pure virtual destructor in C++

```
Do not be anxious about anything, but in everything,  
by prayer and petition, with thanksgiving, present  
your requests to god.  
Philippians 4:6 (Bible)
```

Can a destructor be pure virtual in C++?

Yes, it is possible to have pure virtual destructor. Pure virtual destructor are legal in standard C++ and one of the most important thing is that if class contains pure virtual destructor it is must to provide a function body for the pure virtual destructor. This seems strange that how a virtual function is pure if it requires a function body? But destructors are always called in the reverse order of the class derivation. That means derived class destructor will be invoked first & then base class destructor will be called. If definition for the pure virtual destructor is not provided then what function body will be called during object destruction? Therefore compiler & linker enforce existence of function body for pure virtual destructor.

Consider following program:

```
#include <iostream>
class Base
{
public:
    virtual ~Base()=0; // Pure virtual destructor
};

class Derived : public Base
{
public:
    ~Derived()
    {
        std::cout << "~Derived() is executed";
    }
};

int main()
{
    Base *b=new Derived();
    delete b;
    return 0;
}
```

The linker will produce following error in the above program.

```
test.cpp: (.text$_ZN7DerivedD1Ev[__ZN7DerivedD1Ev]+0x4c):  
undefined reference to `Base::~Base()'
```

Now if the definition for the pure virtual destructor is provided then the program compiles & runs fine.

```
#include <iostream>  
class Base  
{  
public:  
    virtual ~Base()=0; // Pure virtual destructor  
};  
Base::~Base()  
{  
    std::cout << "Pure virtual destructor is called";  
}  
  
class Derived : public Base  
{  
public:  
    ~Derived()  
    {  
        std::cout << "~Derived() is executed\n";  
    }  
};  
  
int main()  
{  
    Base *b = new Derived();  
    delete b;  
    return 0;  
}
```

Output:

```
~Derived() is executed  
Pure virtual destructor is called
```

It is important to note that class becomes abstract class when it contains pure virtual destructor. For example try to compile the below program.

```
#include <iostream>  
class Test  
{  
public:  
    virtual ~Test()=0; // Test now becomes abstract class  
};  
Test::~Test() { }  
  
int main()  
{  
    Test p;  
    Test* t1 = new Test;  
    return 0;  
}
```

The above program fails in compilation & shows following error messages.

[Error] cannot declare variable 'p' to be of abstract type 'Test'

[Note] because the following virtual functions are pure within 'Test':

[Note] virtual Test::~Test()

[Error] cannot allocate an object of abstract type 'Test'

[Note] since type 'Test' has pure virtual functions

Sources:

<http://www.bogotobogo.com/cplusplus/virtualfunctions.php>

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1905.pdf>

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

205. 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 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

206. Some Interesting facts about default arguments in C++

Predict the output of following C++ programs.

1)

```
#include <iostream>
void init(int a=1, int b=2, int c=3);

int main()
{
    init();
    return 0;
}
void init(int a=1, int b=2, int c=3)
{
    std::cout << a << ' ' << b << ' ' << c;
}
```

The above program looks correct at first glance but will fail in compilation. If function uses **default arguments** then default arguments can't be written in both function

declaration & definition. It should only be in declaration, not in definition.

The following program is now correct.

```
#include <iostream>
void init(int a=1, int b=2, int c=3);
int main()
{
    init(); // It is fine
    return 0;
}
void init(int a,int b,int c)
{
    std::cout << a << ' ' << b << ' ' << c;
}
```

2)

```
#include <iostream>

// something looks missing
void init(int =1, int =2, int =3);

int main()
{
    init();
    return 0;
}

void init(int a, int b, int c)
{
    std::cout << a << ' ' << b << ' ' << c;
}
```

If you closely observe function prototype then it looks like an error but it isn't actually. Variable names can be omitted in default arguments.

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.

207. Write your own memcpy()

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);
```

How to implement your own memcpy()?

A simple solution is to simply typecast given addresses to `char*` (char takes 1 byte). Then one by one copy data from source to destination. Below is solution based on the simple idea. One important trick here is to use a temp array instead of directly copying

from src to dest (The use of temp array is discussed below after the program)

```
//C++ program to demonstrate implementation of memcpy()
#include<stdio.h>
#include<string.h>
```

```
// A function to copy block of 'n' bytes from source
// address 'src' to destination address 'dest'.
void myMemCpy(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];
}
```

```
// 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
```

Why do we need a temp array?

The use of temp array is important to handle cases when source and destination addresses are overlapping.

```

// An incorrect 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;

    // INCORRECT: Copy without temp[]
    for (int i=0; i<n; i++)
        cdest[i] = csrc[i];
}

// Driver program
int main()
{
    char csrc[100] = "Geeksfor";
    myMemCpy(csrc+5, csrc, strlen(csrc)+1);
    printf("%s", csrc);
    return 0;
}

```

Output:

```
GeeksGeeksGeek
```

The above program produces incorrect result, the correct result is “GeeksGeeksfor”. Since the input addresses are overlapping, the above program overwrites the original string and causes data loss.

The correct version and the library memcpy function both produce the output as “GeeksGeeksfor”.

The correct version can further be optimized by first checking if the addresses overlap or not and create a temp array only if addresses overlap and the size of temp array can be restricted to only overlapping part.

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.

208. 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 within 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 12 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.

209. 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 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. fptr 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], int_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.