What is the difference between char *a and char a[]?

Discuss it!

There is a lot of difference!

```
char a[] = "string";
char *a = "string";
```

The declaration char a[] asks for space for 7 characters and see that its known by the name "a". In contrast, the declaration char *a, asks for a place that holds a pointer, to be known by the name "a". This pointer "a" can point anywhere. In this case its pointing to an anonymous array of 7 characters, which does have any name in particular. Its just present in memory with a pointer keeping track of its location.

```
char a[] = "string";

   +----+----+----+----+----+----+------+
a: | s  |  t |  r |  i |  n |  g | '\0' |
   +----+----+----+----+----+----+------+
   a[0] a[1] a[2] a[3] a[4] a[5] a[6]


char *a = "string";

+-----+          +---+---+---+---+---+---+------+
|  a: | *======> | s | t | r | i | n | g | '\0' |
+-----+          +---+---+---+---+---+---+------+
Pointer          Anonymous array
```

It is curcial to know that a[3] generates different code depending on whether a is an array or a pointer. When the compiler sees the expression a[3] and if a is an array, it starts at the location "a", goes three elements past it, and returns the character there. When it sees the expression a[3] and if a is a pointer, it starts at the location "a", gets the pointer value there, adds 3 to the pointer value, and gets the character pointed to by that value.

If a is an array, a[3] is three places past a. If a is a pointer, then a[3] is three places past the memory location pointed to by a. In the example above, both a[3] and a[3] return the same character, but the way they do it is different!

Doing something like this would be illegal.

```
char *p = "hello, world!";
p[0] = 'H';
```

How can I declare an array with only one element and still access elements beyond the first element (in a valid fashion)?

Discuss it!

There is a way to do this. Using structures.

```
struct mystruct {
  int  value;
  int length;
  char string[1];
};
```

Now, when allocating memory to the structure using malloc(), allocate more memory than what the structure would normally require!. This way, you can access beyond string[0] (till the extra amount of memory you have allocated, ofcourse).

But remember, compilers which check for array bounds carefully might throw warnings. Also, you need to have a length field in the structure to keep a count of how big your one element array really is :).

A cleaner way of doing this is to have a pointer instead of the one element array and allocate memory for it seperately after allocating memory for the structure.

```
struct mystruct {
  int  value;
  char *string;  // Need to allocate memory using malloc() after
allocating memory for the strucure.
};
```

Is char a[3] = "abc"; legal?

Discuss it!

It declares an array of size three, initialized with the three characters 'a', 'b', and 'c', without the usual terminating '\0' character. The array is therefore not a true C string and cannot be used with strcpy, printf %s, etc. But its legal.

What is the difference between enumeration variables and the preprocessor #defines?

[Discuss it!]

Functionality	Enumerations	#defines
Numeric values assigned automatically?	YES	NO
Can the debugger display the symbolic values?	YES	NO
Obey block scope?	YES	NO
Control over the size of the variables?	NO	NO

What is the difference between the declaration and the definition of a variable?

[Discuss it!]

The definition is the one that actually allocates space, and provides an initialization value, if any.

There can be many declarations, but there must be exactly one definition. A definition tells the compiler to set aside storage for the variable. A declaration makes the variable known to parts of the program that may wish to use it. A variable might be defined and declared in the same statement.

Do Global variables start out as zero?

[Discuss it!]

Uninitialized variables declared with the "static" keyword are initialized to zero. Such variables are implicitly initialized to the null pointer if they are pointers, and to 0.0F if they are floating point numbers.

Local variables start out containing garbage, unless they are explicitly initialized.

Memory obtained with malloc() and realloc() is likely to contain junk, and must be initialized. Memory obtained with calloc() is all-bits-0, but this is not necessarily useful for pointer or floating-point values (This is in contrast to Global pointers and Global floating point numbers, which start as zeroes of the right type).

Does C have boolean variable type?

Discuss it!

No, C does not have a boolean variable type. One can use ints, chars, #defines or enums to achieve the same in C.

```
#define TRUE 1
#define FALSE 0

enum bool {false, true};
```

An enum may be good if the debugger shows the names of enum constants when examining variables

To what does the term storage class refer? What are auto, static, extern, volatile, const classes?

Discuss it!

This is a part **of a variable declaration** that tells the compiler how to interpret the variable's symbol. It does not in itself allocate storage, but it usually tells the compiler how the variable should be stored. Storage class specifiers help you to specify the type of storage used for data objects. Only one storage class specifier is permitted in a declaration this makes sense, as there is only one way of storing things and if you omit the storage class specifier in a declaration, a default is chosen. **The default depends on whether the declaration is made outside a function (external declarations) or inside a function (internal declarations).** For external declarations the default storage class specifier will be **extern and for internal declarations it will be auto.** The only exception to this rule is the declaration of functions, whose default storage class specifier is always extern.

Here are C's storage classes and what they signify:

* **auto - local variables.**
* **static - variables are defined in a nonvolatile region of memory such that they retain their contents though out the program's execution.**
* **register - asks the compiler to devote a processor register to this variable in order to speed the program's execution.** The compiler may not comply and the variable looses it contents and identity when the function it which it is defined terminates.
* **extern - tells the compiler that the variable is defined in another module.**

In C, const and volatile are type qualifiers. The const and volatile type qualifiers are completely independent. A common misconception is to imagine that somehow const is the opposite of volatile and vice versa. This is wrong. The keywords const and volatile can be applied to any declaration, including those of structures, unions, enumerated types or typedef names. Applying them to a declaration is called qualifying the declaration?that's why const and volatile are called type qualifiers, rather than type specifiers.

* **const** means that something **is not modifiable**, so a data object that is declared with const as a part of its type specification must not be assigned to in any way during the run of a program. The main intention of introducing const objects was to allow them to be **put into read-only store**, and to permit compilers to do extra consistency checking in a program. Unless you defeat the intent by doing naughty things with pointers, a compiler is able to check that const objects are not modified explicitly by the user. It is very likely that the definition of the object will contain an initializer (otherwise, since you can't assign to it, how would it ever get a value?), but this is **not always the case. For example, if you were accessing a hardware port at a fixed memory address and promised only to read from it, then it would be declared to be const but not initialized.**
* **volatile tells the compiler that other programs will be modifying this variable in addition to the program being compiled.** For example, an I/O device might need write directly into a program or data space. Meanwhile, the program itself may never directly access the memory area in question. In such a case, we would not want the compiler to optimize-out this data area that never seems to be used by the program, yet must exist for the program to function correctly in a larger context. It tells the compiler that the object is subject to sudden change for reasons which cannot be predicted from a study of the

program itself, and forces every reference to such an object to be a genuine reference.
    * const volatile - Both constant and volatile.

The "volatile" modifier

The volatile modifier is a directive to the compiler?s optimizer that operations involving this variable should not be optimized in certain ways. There are two special cases in which use of the volatile modifier is desirable. The first case involves memory-mapped hardware (a device such as a graphics adaptor that appears to the computer?s hardware as if it were part of the computer?s memory), and the second involves shared memory (memory used by two or more programs running simultaneously). Most computers have a set of registers that can be accessed faster than the computer?s main memory. A good compiler will perform a kind of optimization called ?redundant load and store removal.? The compiler looks for places in the code where it can either remove an instruction to load data from memory because the value is already in a register, or remove an instruction to store data to memory because the value can stay in a register until it is changed again anyway.
If a variable is a pointer to something other than normal memory, such as memory-mapped ports on a
peripheral, redundant load and store optimizations might be detrimental. For instance, here?s a piece of code that might be used to time some operation:

```
time_t time_addition(volatile const struct timer *t, int a)
{
  int n;
  int x;
  time_t then;
  x = 0;
  then = t->value;
  for (n = 0; n < 1000; n++)
  {
    x = x + a;
  }
  return t->value - then;
}
```

In this code, the variable t->value is actually a hardware counter that is being incremented as time passes. The function adds the value of a to x 1000 times, and it returns the amount the timer was incremented by while the 1000 additions were being performed. Without the volatile modifier, a clever optimizer might assume that the value of t does not change during the execution of the function, because there is no statement that explicitly changes it. In that case, there?s no need to read it from memory a second time and subtract it, because the answer will always be 0. The compiler might therefore ?optimize? the function by making it always return 0. If a variable points to data in shared memory, you also don?t want the compiler to perform redundant load and store optimizations. Shared memory is normally used to enable two programs to communicate with each other by having one program store data in the shared portion of memory and the other program read the same portion of memory. If the compiler optimizes away a load or store of shared memory, communication between the two programs will be affected.

What does the typedef keyword do?

Discuss it!

This keyword provides a short-hand way to write variable declarations. It is not a true data typing mechanism, rather, it is syntactic "sugar coating".

For example

```
typedef struct node
{
  int value;
  struct node *next;
}mynode;
```

This can later be used to declare variables like this

```
mynode *ptr1;
```

and not by the lengthy expression

```
struct node *ptr1;
```

There are three main reasons for using typedefs:

    * It makes the writing of complicated declarations a lot easier. This helps in eliminating a lot of clutter in the code.
    * It helps in achieving portability in programs. That is, if we use typedefs for data types that are machine dependent, only the typedefs need to change when the program is ported to a new platform.
    * It helps in providing better documentation for a program. For example, a node of a doubly linked list is better understood as ptrToList than just a pointer to a complicated structure.

What is the difference between constants defined through #define and the constant keyword?

Discuss it!

A constant is similar to a variable in the sense that it represents a memory location (or simply, a value). It is different from a normal variable, in that it cannot change it's value in the proram - it must stay for ever stay constant. In general, constants are a useful because they can prevent program bugs and logical errors(errors are explained later). Unintended modifications are prevented from occurring. The compiler will catch attempts to reassign new values to constants.

Constants may be defined using the preprocessor directive #define. They may also be defined using the const keyword.

So whats the difference between these two?

```
#define ABC 5
```

and

```
const int abc = 5;
```

There are two main advantages of the second one over the first technique. First, the type of the constant is defined. "pi" is float. This allows for some type checking by the compiler. Second, these constants are variables with a definite scope. The scope of a variable relates to parts of your program in which it is defined.

There is also one good use of the important use of the const keyword. Suppose you want to make use of some structure data in some function. You will pass a pointer to that structure as argument to that function. But to make sure that your structure is readonly inside the function you can declare the structure argument as const in function prototype. This will prevent any accidental modification of the structure values inside the function.

What are Trigraph characters?

Discuss it!

These are used when you keyboard does not support some special characters

??=	#
??(	[
??)	]
??<	{
??>	}
??!	|
??/	\
??'	^
??-	~

How are floating point numbers stored? Whats the IEEE format?

Discuss it!

IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macintoshes, and most Unix platforms.

IEEE floating point numbers have three basic components: **the sign, the exponent, and the mantissa.** The mantissa is composed of the fraction and an implicit leading digit (explained below). The exponent base(2) is implicit and need not be stored.

The following figure shows the layout for single (32-bit) and double (64-bit) precision floating-point values. The number of bits for each field are shown (bit ranges are in square brackets):

	Sign	Exponent	Fraction	Bias
Single Precision	**1 [31]**	**8 [30-23]**	**23 [22-00]**	**127**
Double Precision	**1 [63]**	**11 [62-52]**	**52 [51-00]**	**1023**

The sign bit is as simple as it gets. **0 denotes a positive number; 1 denotes a negative number**. Flipping the value of this bit flips the sign of the number.

The exponent field needs to represent both positive and negative exponents. To do this, a bias is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200-127), or 73. For reasons discussed later, exponents of -127 (all 0s) and +128 (all 1s) are reserved for special numbers. For double precision, the exponent field is 11 bits, and has a bias of 1023.

The mantissa, also known as the significand, represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits. To find out the value of the implicit leading bit, consider that any number can be expressed in scientific notation in many different ways. For example, the number five can be represented as any of these:

```
        5.00 × 100
        0.05 × 10 ^ 2
        5000 × 10 ^ -3
```

In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in normalized form. **This basically puts the radix point after the first non-zero digit.** In normalized form, five is represented as 5.0 × 100. A nice little optimization is available to us in base two, since the only possible non-zero digit is 1. Thus, we can just assume a leading digit of 1, and don't need to represent it explicitly. As a result, the mantissa has effectively 24 bits of resolution, by way of 23 fraction bits.

So, to sum up:

1. The sign bit is 0 for positive, 1 for negative.
2. The exponent's base is two.
3. The exponent field contains 127 plus the true exponent for single-precision,
   or 1023 plus the true exponent for double precision.
4. **The first bit of the mantissa is typically assumed to be 1.f, where f is the field of fraction bits.**

When should the register modifier be used?

Discuss it!

The register modifier hints to the compiler that the variable will be heavily used and should be kept in the CPU?s registers, if possible, so that it can be accessed faster. There are several restrictions on the use of the register modifier.

First, the variable must be of a type that can be held in the CPU?s register. This usually means a single value of a size less than or equal to the size of an integer. Some machines have registers that can hold floating-point numbers as well. **Second, because the variable might not be stored in memory, its address cannot be taken with the unary & operator.** An attempt to do so is flagged as an error by the compiler. Some additional rules affect how useful the register modifier is. Because the number of registers is limited, and because some registers can hold only certain types of data (such as pointers or floating-point numbers), the number and types of register modifiers that will actually have any effect are dependent on what machine the program will run on. Any additional register modifiers are silently ignored by the compiler.

Also, in some cases, it might actually be slower to keep a variable in a register because that register then becomes unavailable for other purposes or because the variable isn?t used enough to justify the overhead of loading and storing it. So when should the register modifier be used? The answer is never, with most modern compilers. Early C compilers did not keep any variables in registers unless directed to do so, and the register modifier was a valuable addition to the language. C compiler design has advanced to the point, however, where the compiler will usually make better decisions than the programmer about which variables should be stored in registers. In fact, many compilers actually ignore the register modifier, which is perfectly legal, because it is only a hint and not a directive.

When should a type cast be used?

Discuss it!

There are two situations in which to use a type cast.

The first use is to change the type of an operand to an arithmetic operation so that the operation will be performed properly.

The second case is to cast pointer types to and from void * in order to interface with functions that expect or return void pointers. For example, the following line type casts the return value of the call to malloc() to be a pointer to a foo structure.

```
struct foo *p = (struct foo *) malloc(sizeof(struct foo));
```

A type cast should not be used to override a const or volatile declaration. Overriding these type modifiers can cause the program to fail to run correctly. A type cast should not be used to turn a pointer to one type of structure or data type into another. In the rare events in which this action is beneficial, using a union to hold the values makes the programmer?s intentions clearer

Whats short-circuiting in C expressions?

Discuss it!

What this means is that the right hand side of the expression is not evaluated if the left hand side determines the outcome. **That is if the left hand side is true for || or false for &&, the right hand side is not evaluated.**

Whats wrong with the expression a[i]=i++; ? Whats a sequence point?

Discuss it!

Although its surprising that an expression like i=i+1; is completely valid, something like a[i]=i++; is not. This is because all accesses to an element must be to change the value of that variable. In the statement a[i]=i++; , the access to i is not for itself, but for a[i] and so its invalid. On similar lines, i=i++; or i=++i; are invalid. If you want to increment the value of i, use i=i+1; or i+=1; or i++; or ++i; and not some combination.

A sequence point is a state in time (just after the evaluation of a full expression, or at the ||, &&, ?:, or comma operators, or just before a call to a function) at which there are no side effects.

The ANSI/ISO C Standard states that

Between the previous and next sequence point an object shall have its stored
value modified at most once by the evaluation of an expression. Furthermore,
the prior value shall be accessed only to determine the value to be stored.

At each sequence point, the side effects of all previous expressions will be completed. This is why you cannot rely on expressions such as a[i] = i++;, because there is no sequence point specified for the assignment, increment or index operators, you don't know when the effect of the increment on i occurs.

The sequence points laid down in the Standard are the following:

* The point of calling a function, after evaluating its arguments.
* The end of the first operand of the && operator.
* The end of the first operand of the || operator.
* The end of the first operand of the ?: conditional operator.
* The end of the each operand of the comma operator.
* Completing the evaluation of a full expression. They are the following:

    o Evaluating the initializer of an auto object.
    o The expression in an ?ordinary? statement?an expression followed by semicolon.
    o The controlling expressions in do, while, if, switch or for statements.
    o The other two expressions in a for statement.
    o The expression in a return statement.

Does the ?: (ternary operator) return a lvalue? How can I assign a value to the output of the ternary operator?

Discuss it!

No, it does not return an "lvalue"

Try doing something like this if you want to assign a value to the output of this operator

```
*((mycondition) ? &var1 : &var2) = myexpression;
```

What are #pragmas?

Discuss it!

The directive provides a single, well-defined "escape hatch" which can be used for all sorts of (nonportable) implementation-specific controls and extensions: source listing control, structure packing, warning suppression (like lint's old /* NOTREACHED */ comments), etc.

For example

```
#pragma once
```

inside a header file is an extension implemented by some preprocessors to help make header files idempotent (to prevent a header file from included twice).

Is ++i really faster than i = i + 1?

Discuss it!

Anyone asking this question does not really know what he is talking about.

Any good compiler will and should generate identical code for ++i, i += 1, and i = i + 1. Compilers are meant to optimize code. The programmer should not be bother about such things. Also, it depends on the processor and compiler you are using. One needs to check the compiler's assembly language output, to see which one of the different approcahes are better, if at all.

Note that speed comes Good, well written algorithms and not from such silly tricks.

Can we use variables inside a switch statement? Can we use floating point numbers? Can we use expressions?

Discuss it!

No

The only things that case be used inside a switch statement are constants or enums. Anything else will give you a

```
constant expression required
```

error. That is something like this is not valid

```
switch(i)
{
  case 1: // Something;
          break;
  case j: // Something;
          break;
}
```

So is this. You cannot switch() on strings

```
switch(i)
{
  case "string1" : // Something;
                   break;
  case "string2" : // Something;
                   break;
}
```

This is valid, however

```
switch(i)
```

```
{
  case 1:     // Something;
              break;
  case 1*2+4: // Something;
              break;
}
```

This is also valid, where t is an enum

```
switch(i)
{
  case 1: // Something;
          break;
  case t: // Something;
          break;
}
```

Also note that the default case does not require a break; if and only if its at the end of the switch() statement. Otherwise, even the default case requires a break;

Can goto be used to jump across functions?

[Discuss it!]

No!

This wont work

```
main()
{
  int i=1;
  while (i<=5)
  {
    printf("%d",i);
    if (i>2)
        goto here;
    i++;
  }
}

fun()
{
here:
  printf("PP");
}
```

What is the difference between a deep copy and a shallow copy?

Discuss it!

Deep copy involves using the contents of one object to create another instance of the same class. In a deep copy, the two objects may contain ht same information but the target object will have its own buffers and resources. the destruction of either object will not affect the remaining object. The overloaded assignment operator would create a deep copy of objects.

Shallow copy involves copying the contents of one object into another instance of the same class thus creating a mirror image. Owing to straight copying of references and pointers, the two objects will share the same externally contained contents of the other object to be unpredictable.
Using a copy constructor we simply copy the data values member by member. This method of copying is called shallow copy. If the object is a simple class, comprised of built in types and no pointers this would be acceptable. This function would use the values and the objects and its behavior would not be altered with a shallow copy, only the addresses of pointers that are members are copied and not the value the address is pointing to. The data values of the object would then be inadvertently altered by the function. When the function goes out of scope, the copy of the object with all its data is popped off the stack. If the object has any pointers a deep copy needs to be executed. With the deep copy of an object, memory is allocated for the object in free store and the elements pointed to are copied. A deep copy is used for objects that are returned from a function.

What do lvalue and rvalue mean?

Discuss it!

An lvalue is an expression that could appear on the left-hand sign of an assignment (An object that has a location). An rvalue is any expression that has a value (and that can appear on the right-hand sign of an assignment).

The lvalue refers to the left-hand side of an assignment expression. It must always evaluate to a memory location. The rvalue represents the right-hand side of an assignment expression; it may have any meaningful combination of variables and constants.

Is an array an expression to which we can assign a value?

An lvalue was defined as an expression to which a value can be

assigned. The answer to this question is no, because an array is composed of several separate array elements that cannot be treated as a whole for assignment purposes.

The following statement is therefore illegal:

```
int x[5], y[5];
x = y;
```

Additionally, you might want to copy the whole array all at once. You can do so using a library function such as the memcpy() function, which is shown here:

```
memcpy(x, y, sizeof(y));
```

It should be noted here that unlike arrays, structures can be treated as lvalues. Thus, you can assign one structure variable to another structure variable of the same type, such as this:

```
typedef struct t_name
{
  char last_name[25];
  char first_name[15];
  char middle_init[2];
} NAME;
...
  NAME my_name, your_name;
...
  your_name = my_name;
...
```

What purpose do #if, #else, #elif, #endif, #ifdef, #ifndef serve?

Discuss it!

The following preprocessor directives are used for conditional compilation. Conditional compilation allows statements to be included or omitted based on conditions at compile time.

```
#if
#else
```

```
#elif
#endif
#ifdef
#ifndef
```

In the following example, the printf statements are compiled when the symbol DEBUG is defined, but not compiled otherwise

```
/* remove to suppress debug printf's*/
#define DEBUG
...
x = ....

  #ifdef DEBUG
    printf( "x=%d\n" );
  #endif...

y = ....;

  #ifdef DEBUG
    printf( "y=%d\n" );
  #endif...
```

#if, #else, #elif statements

#if directive

* #if is followed by a intger constant expression.
* If the expression is not zero, the statement(s) following the #if are compiled, otherwise they are ignored.
* #if statements are bounded by a matching #endif, #else or #elif
* Macros, if any, are expanded, and any undefined tokens are replaced with 0 before the constant expression is evaluated
* Relational operators and integer operators may be used

Expression examples

```
#if 1
#if 0
#if ABE == 3
#if ZOO < 12
#if ZIP == 'g'
#if (ABE + 2 - 3 * ZIP) > (ZIP - 2)
```

In most uses, expression is simple relational, often equality test

```
#if SPARKY == '7'
```

#else directive

    * #else marks the beginning of statement(s) to be compiled if the
preceding #if or #elif expression is zero (false)
    * Statements following #else are bounded by matching #endif

Examples

```
#if OS = 'A'
  system( "clear" );
#else
  system( "cls" );
#endif
```

#elif directive

    * #elif adds an else-if branch to a previous #if
    * A series of #elif's provides a case-select type of structure
    * Statement(s) following the #elif are compiled if the expression
is not zero, ignored otherwise
    * Expression is evaluated just like for #if

Examples

```
#if TST == 1
  z = fn1( y );
#elif TST == 2
  z = fn2( y, x );
#elif TST == 3
  z = fn3( y, z, w );
#endif

...
#if ZIP == 'g'
  rc = gzip( fn );
#elif ZIP == 'q'
  rc = qzip( fn );
#else
  rc = zip( fn );
#endif
```

#ifdef and #ifndef directives

Testing for defined macros with #ifdef, #ifndef, and defined()

* #ifdef is used to include or omit statements from compilation depending of whether a macro name is defined or not.
* Often used to allow the same source module to be compiled in different environments (UNIX/ DOS/MVS), or with different options (development/production).
* #ifndef similar, but includes code when macro name is not defined.

Examples

```
#ifdef TESTENV
  printf( "%d ", i );
#endif
#ifndef DOS
  #define LOGFL "/tmp/loga.b";
#else
  #define LOGFL "c:\\tmp\\log.b";
#endif
```

defined() operator

* defined(mac), operator is used with #if and #elif and gives 1 (true) if macro name mac is defined, 0 (false) otherwise.
* Equivalent to using #ifdef and #ifndef, but many shops prefer #if with defined(mac) or !defined(mac)

Examples

```
#if defined(TESTENV)
  printf( "%d ", i );
#endif
#if !defined(DOS)
  #define LOGFL "/tmp/loga.b";
#else
  #define LOGFL "c:\\tmp\\log.b";
#endif
```

```
Nesting conditional statements

Conditional compilation structures may be nested:

#if defined(UNIX)
  #if LOGGING == 'y'
     #define LOGFL "/tmp/err.log"
  #else
     #define LOGFL "/dev/null"
  #endif
#elif defined( MVS )
  #if LOGGING == 'y'
     #define LOGFL "TAP.AVS.LOG"
  #else
     #define LOGFL "NULLFILE"
  #endif
#elif defined( DOS )
  #if LOGGING == 'y'
     #define LOGFL "C:\\tmp\\err.log"
  #else
     #define LOGFL "nul"
  #endif
#endif
```

How can we find out the length of an array dynamically in C?

Discuss it!

```
Here is a C program to do the same...


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

int main()
{
  int arr[] = {3,4,65,78,1,2,4};
  int arr_length = sizeof(arr)/sizeof(arr[0]);

  printf("\nLength of the array is :[%d]\n\n", arr_length);

  getch();
  return(0);
}
```

What does *p++ do? Does it increment p or the value pointed by p?

Discuss it!

The postfix "++" operator has higher precedence than prefix "*" operator. Thus, *p++ is same as *(p++); it increments the pointer p, and returns the value which p pointed to before p was incremented. If you want to increment the value pointed to by p, try (*p)++.

How to write functions which accept two-dimensional arrays when the width is not known before hand?

Discuss it!

Try something like

```
myfunc(&myarray[0][0], NO_OF_ROWS, NO_OF_COLUMNS);

void myfunc(int *array_pointer, int no_of_rows, int no_of_columns)
{
  // myarray[i][j] is accessed as array_pointer[i * no_of_columns + j]
}
```

What do Segmentation fault, access violation, core dump and Bus error mean?

Discuss it!

The segmentation fault, core dump, bus error kind of errors usually mean that the program tried to access memory it shouldn't have.

Probable causes are overflow of local arrays; improper use of null pointers; corruption of the malloc() data structures; mismatched function arguments (specially variable argument functions like sprintf(), fprintf(), scanf(), printf()).

For example, the following code is a sure shot way of inviting a segmentation fault in your program:

```
sprintf(buffer,
        "%s %d",
        "Hello");
```

So whats the difference between a bus error and a segmentation fault?

A bus error is a fatal failure in the execution of a machine language instruction resulting from the processor
detecting an anomalous condition on its bus.

Such conditions include:

- Invalid address alignment (accessing a mullti-byte number at an odd address).
- Accessing a memory location outside its adddress space.
- Accessing a physical address that does nott correspond to any device.
- Out-of-bounds array references.
- References through uninitialized or mangleed pointers.

A bus error triggers a processor-level exception, which Unix translates into a "SIGBUS" signal,which if not caught, will terminate the current process. It looks like a SIGSEGV, but the difference between the two is that SIGSEGV indicates an invalid access to valid memory, while SIGBUS indicates an access to an invalid address.

Bus errors mean different thing on different machines. On systems such as Sparcs a bus error occurs when you access memory that is not positioned correctly.

Maybe an example will help to understand how a bus error occurs

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

int main(void)
{
  char *c;
  long int *i;
  c = (char *) malloc(sizeof(char));
  c++;
  i = (long int *)c;
  printf("%ld", *i);
  return 0;
}
```

On Sparc machines long ints have to be at addresses that are multiples of four (because they are four bytes long), while chars do not (they are only one byte long so they can be put anywhere). The example code uses the char to create an invalid address, and assigns the long int to the invalid address. This causes a bus error when the long int is dereferenced.

A segfault occurs when a process tries to access memory that it is not allowed to, such as the memory at address 0 (where NULL usually points). It is easy to get a segfault, such as the following example, which dereferences NULL.

```
#include < stdio.h>

int main(void)
{
  char *p;
  p = NULL;
  putchar(*p);
  return 0;
}
```

What is a NULL pointer? How is it different from an unitialized pointer? How is a NULL pointer defined?

Discuss it!

A null pointer simply means "I am not allocated yet!" and "I am not pointing to anything yet!".

The C language definition states that for every available pointer type, there is a special value which is called the null pointer. It is guaranteed to compare unequal to a pointer to any object or function.

A null pointer is very different from an uninitialized pointer. A null pointer does not point to any object or function; but an uninitialized pointer can point anywhere.

There is usually a null pointer for each type of a pointer, and the internal values of these null pointers for different pointer types may be different, its up to the compiler. The & operator will never yield a null pointer, nor will a successful call to malloc() (malloc() does return a null pointer when it fails).

```
execl("/bin/ls", "ls", "-l", (char *)0);
```

In this call to execl(), the last argument has been explicitly casted to force the 0 to be treated as a pointer.

Also, if ptr is a pointer then

```
if(ptr){}
```

and

```
if(!ptr){}
```

are perfectly valid.

How is NULL defined?, you may ask.

ANSI C allows the following definition

```
#define NULL ((void *)0)
```

NULL and 0 are interchangeable in pointer contexts.

Make sure you are able to distinguish between the following : the null pointer, the internal representation of a null pointer, the null pointer constant (i.e, 0), the NULL macro, the ASCII null character (NUL), the null string ("").

What is a null pointer assignment error?

Discuss it!

This error means that the program has written, through a null (probably because its an uninitialized) pointer, to a location thats invalid

Does an array always get converted to a pointer? What is the difference between arr and &arr? How does one declare a pointer to an entire array?

Discuss it!

Well, not always.

In C, the array and pointer arithmetic is such that a pointer can be used to access an array or to simulate an array. What this means is whenever an array appears in an expression, the compiler automatically generates a pointer to the array's first element (i.e, &a[0]).

There are three exceptions to this rule

1. When the array is the operand of the sizeof() operator.
2. When using the & operator.
3. When the array is a string literal initializer for a character array.

Also, on a side note, the rule by which arrays decay into pointers is not applied recursively!. An array of arrays (i.e. a two-dimensional array in C) decays into a pointer to an array, not a pointer to a pointer.

If you are passing a two-dimensional array to a function:

```
int myarray[NO_OF_ROWS][NO_OF_COLUMNS];
myfunc(myarray);
```

then, the function's declaration must match:

```
void myfunc(int myarray[][NO_OF_COLUMNS])
```

or

```
void myfunc(int (*myarray)[NO_OF_COLUMNS])
```

Since the called function does not allocate space for the array, it does not need to know the overall size, so the number of rows, NO_OF_ROWS, can be omitted. The width of the array is still important, so the column dimension
NO_OF_COLUMNS must be present.

An array is never passed to a function, but a pointer to the first element of the array is passed to the function. Arrays are automatically allocated memory. They can't be relocated or resized later. Pointers must be assigned to allocated memory (by using (say) malloc), but pointers can be reassigned and made to point to other memory chunks.

So, whats the difference between func(arr) and func(&arr)?

In C, &arr yields a pointer to the entire array. On the other hand, a simple reference to arr returns a pointer to the first element of the array arr. Pointers to arrays (as in &arr) when subscripted or incremented, step over entire arrays, and are useful only when operating on arrays of arrays. Declaring a pointer to an entire array can be done like int (*arr)[N];, where N is the size of the array.

Also, note that sizeof() will not report the size of an array when the array is a parameter to a function, simply because the compiler pretends that the array parameter was declared as a pointer and sizeof reports the size of the pointer.

Is the cast to malloc() required at all?

Discuss it!

Before ANSI C introduced the void * generic pointer, these casts were required because older compilers used to return a char pointer.

```
int *myarray;
myarray = (int *)malloc(no_of_elements * sizeof(int));
```

But, under ANSI Standard C, these casts are no longer necessary as a void pointer can be assigned to any pointer. These casts are still required with C++, however.

What does malloc() , calloc(), realloc(), free() do? What are the common problems with malloc()? Is there a way to find out how much memory a pointer was allocated?

Discuss it!

malloc() is used to allocate memory. Its a memory manager.

calloc(m, n) is also used to allocate memory, just like malloc(). But in addition, it also zero fills the allocated memory area. The zero fill is all-bits-zero. calloc(m.n) is essentially equivalent to

```
p = malloc(m * n);
memset(p, 0, m * n);
```

The malloc() function allocates raw memory given a size in bytes. On the other hand, calloc() clears the requested memory to zeros before return a pointer to it. (It can also compute the request size given the size of the base data structure and the number of them desired.)

The most common source of problems with malloc() are

1. Writing more data to a malloc'ed region than it was allocated to hold.
2. malloc(strlen(string)) instead of (strlen(string) + 1).
3. Using pointers to memory that has been freed.
4. Freeing pointers twice.
5. Freeing pointers not obtained from malloc.
6. Trying to realloc a null pointer.

How does free() work?

Any memory allocated using malloc() realloc() must be freed using free(). In general, for every call to malloc(), there should be a corresponding call to free(). When you call free(), the memory pointed to by the passed pointer is freed. However, the value of the pointer in the caller remains unchanged. Its a good practice to set the pointer to NULL after freeing it to prevent accidental usage. The malloc()/free() implementation keeps track of the size of each block as it is allocated, so it is not required to remind it of the size when freeing it using free(). You can't use dynamically-allocated memory after you free it.

Is there a way to know how big an allocated block is?

Unfortunately there is no standard or portable way to know how big an allocated block is using the pointer to the block!. God knows why this was left out in C.

Is this a valid expression?

```
pointer = realloc(0, sizeof(int));
```

Yes, it is!

What's the difference between const char *p, char * const p and const char * const p?

Discuss it!

```
const char *p    -   This is a pointer to a constant char. One cannot
change the value
                     pointed at by p, but can change the pointer p
itself.
```

```
                      *p = 'A' is illegal.
                      p  = "Hello" is legal.

                      Note that even char const *p is the same!

const * char p   -    This is a constant pointer to (non-const) char.
One cannot change
                      the pointer p, but can change the value pointed at
by p.

                      *p = 'A' is legal.
                      p  = "Hello" is illegal.



const char * const p  -    This is a constant pointer to constant char!
One cannot
                           change the value pointed to by p nor the
pointer.

                           *p = 'A' is illegal.
                           p  = "Hello" is also illegal.
```

To interpret these declarations, let us first consider the general form of declaration:

```
  [qualifier] [storage-class] type [*[*]..] [qualifier] ident ;

                              or

  [storage-class] [qualifier] type [*[*]..] [qualifier] ident ;
```

where,

```
qualifier:
            volatile
            const

storage-class:

            auto            extern
            static          register

type:
            void            char            short
            int             long            float
            double          signed          unsigned
            enum-specifier
            typedef-name
            struct-or-union-specifier
```

Both the forms are equivalent. Keywords in the brackets are optional. The simplest tip here is to notice the relative position of the `const' keyword with respect to the asterisk (*).

Note the following points:

* If the `const' keyword is to the left of the asterisk, and is the only such keyword in the declaration, then object pointed by the pointer is constant, however, the pointer itself is variable. For example:

```
      const char * pcc;
      char const * pcc;
```

* If the `const' keyword is to the right of the asterisk, and is the only such keyword in the declaration, then the object pointed by the pointer is variable, but the pointer is constant; i.e., the pointer, once initialized, will always point to the same object through out it's scope. For example:

```
      char * const cpc;
```

* If the `const' keyword is on both sides of the asterisk, the both the pointer and the pointed object are constant. For example:

```
      const char * const cpcc;
      char const * const cpcc2;
```

One can also follow the "nearness" principle; i.e.,

* If the `const' keyword is nearest to the `type', then the object is constant. For example:

```
      char const * pcc;
```

```
    * If the `const' keyword is nearest to the identifier, then the
pointer is constant. For example:
```

```
      char * const cpc;
```

```
    * If the `const' keyword is nearest, both to the identifier and the
type, then both the pointer and the object are constant. For example:
```

```
      const char * const cpcc;
      char const * const cpcc2;
```

However, the first method seems more reliable...

What is a void pointer? Why can't we perform arithmetic on a void * pointer?

Discuss it!

The void data type is used when no other data type is appropriate. A void pointer is a pointer that may point to any kind of object at all. It is used when a pointer must be specified but its type is unknown.

The compiler doesn't know the size of the pointed-to objects incase of a void * pointer. Before performing arithmetic, convert the pointer either to char * or to the pointer type you're trying to manipulate

What is the difference between an array of pointers and a pointer to an array?

Discuss it!

This is an array of pointers

```
int *p[10];
```

This is a pointer to a 10 element array

```
int (*p)[10];
```

What is a dangling pointer? What are reference counters with respect to pointers?

Discuss it!

A pointer which points to an object that no longer exists. Its a pointer referring to an area of memory that has been deallocated. Dereferencing such a pointer usually produces garbage.

Using reference counters which keep track of how many pointers are pointing to this memory location can prevent such issues. The reference counts are incremented when a new pointer starts to point to the memory location and decremented when they no longer need to point to that memory. When the reference count reaches zero, the memory can be safely freed. Also, once freed, the corresponding pointer must be set to NULL.

What operations are valid on pointers? When does one get the Illegal use of pointer in function error?

Discuss it!

This is what is Valid

```
    px<py
    px>=py
    px==py
    px!=py
    px==NULL
    px=px+n
    px=px-n
    px-py
```

Everything else is invalid (multiplication, division, addition of two pointers)!

Something like

```
j = j * 2;
k = k / 2;
```

where j and k are pointers will give this error

```
Illegal use of pointer in function main
```

What are the common causes of pointer bugs?

Discuss it!

* Uninitialized pointers : One of the easiest ways to create a pointer bug is to try to reference the value of a pointer even though the pointer is uninitialized and does not yet point to a valid address. For example:

```
int *p;
*p = 12;
```

The pointer p is uninitialized and points to a random location in memory when you declare it. It could be pointing into the system stack, or the global variables, or into the program's code space, or into the operating system. When you say *p=12;, the program will simply try to write a 12 to whatever random location p points to. The program may explode immediately, or may wait half an hour and then explode, or it may subtly corrupt data in another part of your program and you may never realize it. This can make this error very hard to track down. Make sure you initialize all pointers to a valid address before dereferencing them.

* Invalid Pointer References : An invalid pointer reference occurs when a pointer's value is referenced even though the pointer doesn't point to a valid block. One way to create this error is to say p=q;, when q is uninitialized. The pointer p will then become uninitialized as well, and any reference to *p is an invalid pointer reference. The only way to avoid this bug is to draw pictures of each step of the program and make sure that all pointers point somewhere. Invalid pointer references cause a program to crash inexplicably for the same reasons given in cause 1.

* Zero Pointer Reference : A zero pointer reference occurs whenever a pointer pointing to zero is used in a statement that attempts to reference a block. For example, if p is a pointer to an integer, the following code is invalid:

```
      p = 0;
      *p = 12;
```

There is no block pointed to by p. Therefore, trying to read or write anything from or to that block is an invalid zero pointer reference. There are good, valid reasons to point a pointer to zero. Dereferencing such a pointer, however, is invalid.

Why is sizeof() an operator and not a function?

Discuss it!

sizeof() is a compile time operator. To calculate the size of an object, we need the type information. This type information is available only at compile time. At the end of the compilation phase, the resulting object code doesn't have (or not required to have) the type information. Of course, type information can be stored to access it at run-time, but this results in bigger object code and less performance. And most of the time, we don't need it. All the runtime environments that support run time type identification (RTTI) will retain type information even after compilation phase. But, if something can be done in compilation time itself, why do it at run time?

On a side note, something like this is illegal...

```
printf("%u\n", sizeof(main));
```

This asks for the size of the main function, which is actually illegal:

6.5.3.4 The sizeof operator
The sizeof operator shall not be applied to an expression that has function type....

What is the difference between malloc() and calloc()?

Discuss it!

First lets look at the prototypes of these two popular functions..

```
  #include <stdlib.h>
  void *calloc(size_t n, size_t size);
  void *malloc(size_t size);
```

The two functions malloc() and calloc() are functionally same in that they both allocate memory from a storage pool (generally called heap). Actually, the right thing to say is that these two functions are memory managers and not memory allocators. Memory allocation is done by OS specific routines (like brk() and sbrk()). But lets not get into that for now...

Here are some differences between these two functions..

* malloc() takes one argument, whereas calloc() takes two.
* calloc() initializes all the bits in the allocated space to zero (this is all-bits-zero!, where as malloc() does not do this.
* A call to calloc() is equivalent to a call to malloc() followed by one to memset().

```
      calloc(m, n)

      is essentially equivalent to

      p = malloc(m * n);
      memset(p, 0, m * n);
```

Using calloc(), we can carry out the functionality in a faster way than a combination of malloc() and memset() probably would. You will agree that one libray call is faster than two calls. Additionally, if provided by the native CPU, calloc() could be implementated by the CPU's "allocate-and-initialize-to-zero" instruction.
* The reason for providing the "n" argument is that sometimes it is required to allocate a number ("n") of uniform objects of a particular size ("size"). Database application, for instance, will have such requirements. Proper planning for the values of "n" and "size" can lead to good memory utilization.

What are brk() and sbrk() used for? How are they different from malloc()?

Discuss it!

brk() and sbrk() are the only calls of memory management in UNIX. For one value of the address, beyond the last logical data page of the process, the MMU generates a segmentation violation interrupt and UNIX kills the process. This address is known as the break address of a process. Addition of a logical page to the data space implies raising of the break address (by a multiple of a page size). Removal of an entry from the page translation table automatically lowers the break address.

brk()and sbrk() are systems calls for this process

```
char *brk(char *new_break);
char *sbrk(displacement)
```

Both calls return the old break address to the process. In brk(), the new break address desired needs to be specified as the parameter. In sbrk(), the displacement (+ve or -ve) is the difference between the new and the old break address. sbrk() is very similar to malloc() when it allocates memory (+ve displacement).

malloc() is really a memory manager and not a memory allocator since, brk/sbrk only can do memory allocations under UNIX. malloc() keeps track of occupied and free peices of memory. Each malloc request is expected to give consecutive bytes and hence malloc selects the smallest free pieces that satisfy a request. When free is called, any consecutive free pieces are coalesced into a large free piece. These is done to avoid fragmentation.

realloc() can be used only with a preallocated/malloced/realloced memory. realloc() will automatically allocate new memory and transfer maximum possible contents if the new space is not available. Hence the returned value of realloc must always be stored back into the old pointer itself.

Implement the memmove() function. What is the difference between the memmove() and memcpy() function?

Discuss it!

One more most frequently asked interview question!.

memmove() offers guaranteed behavior if the source and destination arguments overlap. memcpy() makes no such guarantee, and may therefore be more efficient to implement. It's always safer to use memmove().

Note that the prototype of memmove() is ...

```
void *memmove(void *dest, const void *src, size_t count);
```

Here is an implementation..

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

void *mymemmove(void *dest, const void *src, size_t count);

int main(int argc, char* argv[])
{
  char *p1, *p2;
  char *p3, *p4;
  int  size;

  printf("\n--------------------------------\n");

  /* ----------------------------------------
   *
   * CASE 1 : From (SRC) < To (DEST)
   *
   *     +--+---------------------+--+
   *     |  |                     |  |
  *     +--+---------------------+--+
   *     ^  ^
   *     |  |
   *   From To
   *
   * --------------------------------------- */

  p1 = (char *) malloc(12);
  memset(p1,12,'\0');
  size=10;

  strcpy(p1,"ABCDEFGHI");

  p2 = p1 + 2;

  printf("\n--------------------------------\n");
  printf("\nFrom (before) = [%s]",p1);
  printf("\nTo (before)   = [%s]",p2);

  mymemmove(p2,p1,size);

  printf("\n\nFrom (after) = [%s]",p1);
  printf("\nTo (after)   = [%s]",p2);

  printf("\n--------------------------------\n");



  /* ----------------------------------------
   *
   * CASE 2 : From (SRC) > To (DEST)
   *
   *     +--+---------------------+--+
   *     |  |                     |  |
   *     +--+---------------------+--+
   *     ^  ^
   *     |  |
   *    To From
   *
   * --------------------------------------- */


  p3 = (char *) malloc(12);
  memset(p3,12,'\0');
  p4 = p3 + 2;

  strcpy(p4, "ABCDEFGHI");

  printf("\nFrom (before) = [%s]",p4);
  printf("\nTo (before)   = [%s]",p3);

  mymemmove(p3, p4, size);

  printf("\n\nFrom (after) = [%s]",p4);
  printf("\nTo (after)   = [%s]",p3);

  printf("\n--------------------------------\n");


  /* ----------------------------------------
   *
   * CASE 3 : No overlap
   *
   * --------------------------------------- */

  p1 = (char *) malloc(30);
  memset(p1,30,'\0');
  size=10;

  strcpy(p1,"ABCDEFGHI");

  p2 = p1 + 15;

  printf("\n--------------------------------\n");
  printf("\nFrom (before) = [%s]",p1);
  printf("\nTo (before)   = [%s]",p2);

  mymemmove(p2,p1,size);

  printf("\n\nFrom (after) = [%s]",p1);
  printf("\nTo (after)   = [%s]",p2);

  printf("\n--------------------------------\n");

  printf("\n\n");

  return 0;
}
```

```
void *mymemmove(void *to, const void *from, size_t size)
{
    unsigned char *p1;
    const unsigned char *p2;

    p1 = (unsigned char *) to;
    p2 = (const unsigned char *) from;

    p2 = p2 + size;

    // Check if there is an overlap or not.
    while (p2 != from && --p2 != to);


    if (p2 != from)
    {
        // Overlap detected!

        p2  = (const unsigned char *) from;
        p2  = p2 + size;
        p1  = p1 + size;

        while (size-- != 0)
        {
            *--p1 = *--p2;
        }
    }
    else
    {
        // No overlap OR they overlap as CASE 2 above.
        // memcopy() would have done this directly.

        while (size-- != 0)
        {
            *p1++ = *p2++;
        }
    }

    return(to);
}
```

And here is the output

```
--------------------------------

From (before) = [ABCDEFGHI]
To   (before) = [CDEFGHI]
```

```
From (after) = [ABABCDEFGHI]
To   (after) = [ABCDEFGHI]

--------------------------------

From (before) = [ABCDEFGHI]
To   (before) = [a+ABCDEFGHI]


From (after) = [CDEFGHI]
To   (after) = [ABCDEFGHI]

--------------------------------

From (before) = [ABCDEFGHI]
To   (before) = [FEd?&:F]


From (after) = [ABCDEFGHI]
To (after)   = [ABCDEFGHI]

--------------------------------
```

So then, whats the difference between the implementation of memmove() and memcpy(). Its just that memcpy() will not care if the memories overlap and will either copy from left to right or right to left without checking which method to used depending on the type of the overlap. Also note that the C code proves that the results are the same irrespective of the Endian-ness of the machine.

Write C code to implement the strstr() (search for a substring) function.

Discuss it!

This is also one of the most frequently asked interview questions. Its asked almost 99% of the times. Here are a few C programs to implement your own strstr() function.

There are a number of ways to find a string inside another string. Its important to be aware of these algorithms than to memorize them. Some of the fastest algorithms are quite tough to understand!.

Method1

The first method is the classic Brute force method. The Brute Force algorithm checks, at all positions in the text between 0 and (n-m), if an occurrence of the pattern starts at that position or not. Then, after each successfull or unsuccessful attempt, it shifts the pattern exactly one position to the right. The time complexity of this searching phase is O(mn). The expected number of text character comparisons is 2n.

Here 'n' is the size of the string in which the substring of size 'm' is being searched for.

Here is some code (which works!)

```
#include<stdio.h>

void BruteForce(char *x /* pattern */,
                int m   /* length of the pattern */,
                char *y /* actual string being searched */,
                int n   /* length of this string */)
{
   int i, j;
   printf("\nstring    : [%s]"
          "\nlength    : [%d]"
          "\npattern   : [%s]"
          "\nlength    : [%d]\n\n", y,n,x,m);


   /* Searching */
   for (j = 0; j <= (n - m); ++j)
   {
      for (i = 0; i < m && x[i] == y[i + j]; ++i);
        if (i >= m) {printf("\nMatch found at\n\n->[%d]\n-
>[%s]\n",j,y+j);}
   }
}


int main()
{
  char *string  = "hereroheroero";
  char *pattern = "hero";

  BF(pattern,strlen(pattern),string,strlen(string));
  printf("\n\n");
  return(0);
}
```

This is how the comparison happens visually

```
hereroheroero
   !
hero


hereroheroero
!
```

```
hero


hereroheroero
  !
  hero


hereroheroero
   !
   hero


hereroheroero
    !
    hero


hereroheroero
     !
     hero


hereroheroero
      |||| ----> Match!
      hero


hereroheroero
       !
       hero


hereroheroero
        !
        hero


hereroheroero
         !
         hero
```

Method2

The second method is called the Rabin-Karp method.

Instead of checking at each position of the text if the pattern occurs or not, it is better to check first if the contents of the current string "window" looks like the pattern or not. In order to check the resemblance between these two patterns, a hashing function is used.

Hashing a string involves computing a numerical value from the value of its characters using a hash function.

The Rabin-Karp method uses the rule that if two strings are equal, their hash values must also be equal. Note that the converse of this statement is not always true, but a good hash function tries to reduce the number of such hash collisions. Rabin-Karp computes hash value of the pattern, and then goes through the string computing hash values of all of its substrings and checking if the pattern's hash value is equal to the substring hash value, and advancing by 1 character every time. If the two hash values are the same, then the algorithm verifies if the two string really are equal, rather than this being a fluke of the hashing scheme. It uses regular string comparison for this final check. Rabin-Karp is an algorithm of choice for multiple pattern search. If we want to find any of a large number, say k, fixed length patterns in a text, a variant Rabin-Karp that uses a hash table to check whether the hash of a given string belongs to a set of hash values of patterns we are looking for. Other algorithms can search for a single pattern in time order O(n), hence they will search for k patterns in time order O(n*k). The variant Rabin-Karp will still work in time order O(n) in the best and average case because a hash table allows to check whether or not substring hash equals any of the pattern hashes in time order of O(1).

Here is some code (not working though!)

```
#include<stdio.h>

hashing_function()
{
  // A hashing function to compute the hash values of the strings.
  ....
}

void KarpRabinR(char *x, int m, char *y, int n)
{
   int hx, hy, i, j;

   printf("\nstring    : [%s]"
          "\nlength    : [%d]"
          "\npattern   : [%s]"
          "\nlength    : [%d]\n\n", y,n,x,m);


   /* Preprocessing  phase */
   Do preprocessing here..

   /* Searching */
   j = 0;
   while (j <= n-m)
   {
      if (hx == hy && memcmp(x, y + j, m) == 0)
      {
         // Hashes match and so do the actual strings!
         printf("\nMatch found at : [%d]\n",j);
      }

      hy = hashing_function(y[j], y[j + m], hy);
      ++j;
   }
}


int main()
{

  char *string="hereroheroero";
  char *pattern="hero";

  KarpRabin(pattern,strlen(pattern),string,strlen(string));

  printf("\n\n");
  return(0);

}
```

This is how the comparison happens visually

```
hereroheroero
   !
hero


hereroheroero
!
hero


hereroheroero
  !
  hero


hereroheroero
   !
   hero


hereroheroero
    !
    hero


hereroheroero
     !
     hero
```

```
hereroheroero
      |||| ----> Hash values match, so do the strings!
      hero


hereroheroero
       !
       hero


hereroheroero
        !
        hero


hereroheroero
         !
         hero
```

Method3

The Knuth-Morris-Pratt or the Morris-Pratt algorithms are extensions of the basic Brute Force algorithm. They use precomputed data to skip forward not by 1 character, but by as many as possible for the search to succeed.

Here is some code

```
void preComputeData(char *x, int m, int Next[])
{
   int i, j;
   i = 0;
   j = Next[0] = -1;

   while (i < m)
   {
      while (j > -1 && x[i] != x[j])
         j = Next[j];
      Next[++i] = ++j;

   }
}


void MorrisPrat(char *x, int m, char *y, int n)
{
   int i, j, Next[1000];

   /* Preprocessing */
   preComputeData(x, m, Next);
   /* Searching */
   i = j = 0;
   while (j < n)
   {
      while (i > -1 && x[i] != y[j])
         i = Next[i];
      i++;
      j++;
      if (i >= m)
      {
         printf("\nMatch found at : [%d]\n",j - i);
         i = Next[i];
      }
   }
}


int main()
{
  char *string="hereroheroero";
  char *pattern="hero";

  MorrisPrat(pattern,strlen(pattern),string,strlen(string));

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

This is how the comparison happens visually

```
hereroheroero
   !
hero


hereroheroero
   !
   hero


hereroheroero
    !
    hero


hereroheroero
     !
     hero



hereroheroero
      |||| ----> Match found!
      hero


hereroheroero
          !
          hero
```

Method4

The Boyer Moore algorithm is the fastest string searching algorithm. Most editors use this algorithm.

It compares the pattern with the actual string from right to left. Most other algorithms compare from left to right. If the character that is compared with the rightmost pattern symbol does not occur in the pattern at all, then the pattern can be shifted by m positions behind this text symbol.

The following example illustrates this situation.

Example:

```
0 1 2 3 4 5 6 7 8 9 ...
a b b a d a b a c b a
        | |
b a b a c |
  <------ |
          |
          b a b a c
```

The comparison of "d" with "c" at position 4 does not match. "d" does not occur in the pattern. Therefore, the pattern cannot match at any of the positions 0,1,2,3,4, since all corresponding windows contain a "d". The pattern can be shifted to position 5. The best case for the Boyer-Moore algorithm happens if, at each search attempt the first compared character does not occur in the pattern. Then the algorithm requires only O(n/m) comparisons .

Bad character heuristics

This method is called bad character heuristics. It can also be applied if the bad character (the character that causes a mismatch), occurs somewhere else in the pattern. Then the pattern can be shifted so that it is aligned to this text symbol. The next example illustrates this situation.

Example:

```
0 1 2 3 4 5 6 7 8 9 ...
a b b a b a b a c b a
        |
b a b a c
    <----
    |
    b a b a c
```

Comparison between "b" and "c" causes a mismatch. The character "b" occurs in the pattern at positions 0 and 2. The pattern can be shifted so that the rightmost "b" in the pattern is aligned to "b".

Good suffix heuristics

Sometimes the bad character heuristics fails. In the following situation the comparison between "a" and "b" causes a mismatch. An alignment of the rightmost occurence of the pattern symbol a with the text symbol a would produce a negative shift. Instead, a shift by 1 would be possible. However, in this case it is better to derive the maximum possible shift distance from the structure of the pattern. This method is called good suffix heuristics.

Example:

```
0 1 2 3 4 5 6 7 8 9 ...
a b a a b a b a c b a
    | | |
c a b a b
    <----
    | | |
    c a b a b
```

The suffix "ab" has matched. The pattern can be shifted until the next occurence of ab in the pattern is aligned to the text symbols ab, i.e. to position 2.

In the following situation the suffix "ab" has matched. There is no other occurence of "ab" in the pattern.Therefore, the pattern can be shifted behind "ab", i.e. to position 5.

Example:

```
0 1 2 3 4 5 6 7 8 9 ...
a b c a b a b a c b a
    | | |
c b a a b
          c b a a b
```

In the following situation the suffix "bab" has matched. There is no other occurence of "bab" in the pattern. But in this case the pattern cannot be shifted to position 5 as before, but only to position 3, since a prefix of the pattern "ab" matches the end of "bab". We refer to this situation as case 2 of the good suffix heuristics.

Example:

```
0 1 2 3 4 5 6 7 8 9 ...
a a b a b a b a c b a
  | | | |
a b b a b
      a b b a b
```

The pattern is shifted by the longest of the two distances that are given by the bad character and the good suffix heuristics.

The Boyer-Moore algorithm uses two different heuristics for determining the maximum possible shift distance in case of a mismatch: the "bad character" and the "good suffix" heuristics. Both heuristics can lead to a shift distance of m. For the bad character heuristics this is the case, if the first comparison causes a mismatch and the corresponding text symbol does not occur in the pattern at all. For the good suffix heuristics this is the case, if only the first comparison was a match, but that symbol does not occur elsewhere in the pattern.

A lot of these algorithms have been explained here with good visualizations. Remember, again that its sufficient to know the basic Brute force algorithm and be aware of the other methods. No one expects you to know every possible algorithm on earth.

Write your own printf() function in C

Discuss it!

This is again one of the most frequently asked interview questions. Here is a C program which implements a basic version of printf(). This is a really, really simplified version of printf(). Note carefully how floating point and other compilcated support has been left out. Also, note how we use low level puts() and putchar(). Dont make a fool of yourself by using printf() within the implementation of printf()!

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

main()
{
    void myprintf(char *,...);
    char * convert(unsigned int, int);
    int i=65;
    char str[]="This is my string";
    myprintf("\nMessage = %s%d%x",str,i,i);
}

void myprintf(char * frmt,...)
{

    char *p;
    int i;
    unsigned u;
    char *s;
    va_list argp;


    va_start(argp, fmt);

    p=fmt;
    for(p=fmt; *p!='\0';p++)
    {
        if(*p=='%')
        {
            putchar(*p);continue;
        }

        p++;

        switch(*p)
        {
            case 'c' : i=va_arg(argp,int);putchar(i);break;
            case 'd' : i=va_arg(argp,int);
                                if(i<0){i=-i;putchar('-
');}puts(convert(i,10));break;
            case 'o': i=va_arg(argp,unsigned int);
puts(convert(i,8));break;
            case 's': s=va_arg(argp,char *); puts(s); break;
            case 'u': u=va_arg(argp,argp, unsigned int);
puts(convert(u,10));break;
            case 'x': u=va_arg(argp,argp, unsigned int);
puts(convert(u,16));break;
            case '%': putchar('%');break;
        }
    }

    va_end(argp);
}

char *convert(unsigned int, int)
{
    static char buf[33];
    char *ptr;

    ptr=&buf[sizeof(buff)-1];
    *ptr='\0';
    do
    {
        *--ptr="0123456789abcdef"[num%base];
        num/=base;
    }while(num!=0);
    return(ptr);
}
```

Implement the strcpy() function.

[Discuss it!]

Here are some C programs which implement the strcpy() function. This is one of the most frequently asked C interview questions.

Method1

```
char *mystrcpy(char *dst, const char *src)
{
  char *ptr;
  ptr = dst;
  while(*dst++=*src++);
  return(ptr);
}
```

The strcpy function copies src, including the terminating null character, to the location specified by dst. No overflow checking is performed when strings are copied or appended. The behavior of strcpy is undefined if the source and destination strings overlap. It returns the destination string. No return value is reserved to indicate an error.

Note that the prototype of strcpy as per the C standards is

```
char *strcpy(char *dst, const char *src);
```

Notice the const for the source, which signifies that the function must not change the source string in anyway!.

Method2

```
char *my_strcpy(char dest[], const char source[])
{
  int i = 0;
  while (source[i] != '\0')
  {
    dest[i] = source[i];
    i++;
  }
  dest[i] = '\0';
  return(dest);
}
```

Implement the strcmp(str1, str2) function.

Discuss it!

There are many ways one can implement the strcmp() function. Note that strcmp(str1,str2) returns a -ve number if str1 is alphabetically above str2, 0 if both are equal and +ve if str2 is alphabetically above str1.

Here are some C programs which implement the strcmp() function. This is also one of the most frequently asked interview questions. The prototype of strcmp() is

```
int strcmp( const char *string1, const char *string2 );
```

Here is some C code..

```
#include <stdio.h>

int mystrcmp(const char *s1, const char *s2);

int main()
{
  printf("\nstrcmp() = [%d]\n", mystrcmp("A","A"));
  printf("\nstrcmp() = [%d]\n", mystrcmp("A","B"));
  printf("\nstrcmp() = [%d]\n", mystrcmp("B","A"));
  return(0);
}

int mystrcmp(const char *s1, const char *s2)
{
    while (*s1==*s2)
    {
        if(*s1=='\0')
           return(0);
        s1++;
        s2++;
    }
    return(*s1-*s2);
}
```

And here is the output...

```
strcmp() = [0]
strcmp() = [-1]
strcmp() = [1]
```

Implement the substr() function in C.

Discuss it!

Here is a C program which implements the substr() function in C.

```
int main()
{
  char str1[] = "India";
  char str2[25];

  substr(str2, str1, 1, 3);
  printf("\nstr2 : [%s]", str2);
  return(0);
}

substr(char *dest, char *src, int position, int length)
{
    dest[0]='\0';
    strncat(dest, (src + position), length);
}
```

Here is another C program to do the same...

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

void mySubstr(char *dest, char *src, int position, int length);

int main()
{
char subStr[100];
char str[]="My Name Is Sweet";

mySubstr(subStr, str, 1, 5);
printf("\nstr    = [%s]"
        "\nsubStr = [%s]\n\n",
        str, subStr);
getch();
return(0);
}
```

```
void mySubstr(char *dest, char *src, int position, int length)
{
  while(length > 0)
  {
    *dest = *(src+position);
    dest++;
    src++;
    length--;
  }
}
```

Write your own copy() function

Discuss it!

Here is some C code that simulates a file copy action.

```
#include <stdio.h>              /* standard I/O routines. */
#define MAX_LINE_LEN 1000 /* maximum line length supported. */


void main(int argc, char* argv[])
{
    char* file_path_from;
    char* file_path_to;
    FILE* f_from;
    FILE* f_to;
    char buf[MAX_LINE_LEN+1];

    file_path_from = "<something>";
    file_path_to   = "<something_else>";

    f_from = fopen(file_path_from, "r");
    if (!f_from) {exit(1);}

    f_to = fopen(file_path_to, "w+");
    if (!f_to) {exit(1);}

    /* Copy source to target, line by line. */
    while (fgets(buf, MAX_LINE_LEN+1, f_from))
    {
        if (fputs(buf, f_to) == EOF){exit(1);}
    }

    if (!feof(f_from)){exit(1);}

    if (fclose(f_from) == EOF) {exit(1);}
    if (fclose(f_to) == EOF)   {exit(1);}

    return(0);
}
```

```
    if(ch>='A' && ch <='Z')
       return(1); //Yes, its upper!
    else
       return(0); // No, its lower!
}
```

Its important to know that the upper and lower case alphabets have corresponding integer values.

A-Z - 65-90
a-z - 97-122

Another way to do this conversion is to maintain a correspondance between the upper and lower case alphabets. The program below does that. This frees us from the fact that these alphabets have a corresponding integer values. I dont know what one should do for non-english alphabets. Do other languages have upper and lower case letters in the first place :) !

```
#include <string.h>

#define UPPER   "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
#define LOWER   "abcdefghijklmnopqrstuvwxyz"

int toUpper(int c)
{
    const char *upper;
    const char *const lower = LOWER;

    // Get the position of the lower case alphabet in the LOWER string
using the strchr() function ..
    upper = ( ((CHAR_MAX >= c)&&(c > '\0')) ? strchr(lower, c) : NULL);


    // Now return the corresponding alphabet at that position in the
UPPER string ..
    return((upper != NULL)?UPPER[upper - lower] : c);
}
```

Note that these routines dont have much error handling incorporated in them. Its really easy to add error handling to these routines or just leave it out (as I like it). This site consciously leaves out error handling for most of the programs to prevent unwanted clutter and present the core logic first.

Write a C program to implement your own strdup() function.

Discuss it!

Here is a C program to implement the strdup() function.

```
char *mystrdup(char *s)
{
    char *result = (char*)malloc(strlen(s) + 1);
    if (result == (char*)0){return (char*)0;}
    strcpy(result, s);
    return result;
}
```

Write your own C program to implement the atoi() function

Discuss it!

The prototype of the atoi() function is ...

```
int atoi(const char *string);
```

Here is a C program which explains a different way of coding the atoi() function in the C language.

```
#include<stdio.h>

int myatoi(const char *string);

int main(int argc, char* argv[])
{
  printf("\n%d\n", myatoi("1998"));
  getch();
  return(0);
}
```

```
int myatoi(const char *string)
{
    int i;
    i=0;
    while(*string)
    {
        i=(i<<3) + (i<<1) + (*string - '0');
        string++;

        // Dont increment i!

    }
    return(i);
}
```

Try working it out with a small string like "1998", you will find out it does work!.

Ofcourse, there is also the trivial method ....

"1998" == 8 + (10 * 9) + (100 * 9) + (1 * 1000) = 1998

This can be done either by going from right to left or left to right in the string

One solution is given below

```
int myatoi(const char* string)
{
  int value = 0;

  if (string)
  {
    while (*string && (*string <= '9' && *string >= '0'))
    {
      value = (value * 10) + (*string - '0');
      string++;
    }
  }
  return value;
}
```

Note that these functions have no error handling incorporated in them (what happens if someone passes non-numeric data (say "1A998"), or negative numeric strings (say "-1998")). I leave it up to you to add these cases. The essense is to understand the core logic first.

Write a C program to implement your own strdup() function.

Discuss it!

Here is a C program to implement the strdup() function.

```
char *mystrdup(char *s)
{
    char *result = (char*)malloc(strlen(s) + 1);
    if (result == (char*)0){return (char*)0;}
    strcpy(result, s);
    return result;
}
```

Write a C program to implement the strlen() function

Discuss it!

The prototype of the strlen() function is...

```
size_t strlen(const char *string);
```

Here is some C code which implements the strlen() function....

```
int my_strlen(char *string)
{
  int length;
  for (length = 0; *string != '\0', string++)
  {
    length++;
  }
  return(length);
}
```

Also, see another example

```
int my_strlen(char *s)
{
  char *p=s;

  while(*p!='\0')
    p++;

  return(p-s);
}
```

Write your own strcat() function

Discuss it!

```
Here is a C function which implements the strcat() function...


/* Function to concatenate string t to end of s; return s */
char *myStrcat(char *s, const char *t)
{
    char *p = s;

    if (s == NULL || t == NULL)
        return s;   /* we need not have to do anything */

    while (*s)
        s++;

    while (*s++ = *t++)
        ;

    return p;
}
```

Write a C program to swap two variables without using a temporary variable

Discuss it!

```
This questions is asked almost always in every interview.

The best way to swap two variables is to use a temporary variable.


int a,b,t;
```

```
t = a;
a = b;
b = t;
```

There is no way better than this as you will find out soon. There are a few slick expressions that do swap variables without using temporary storage. But they come with their own set of problems.

Method1 (The XOR trick)

```
a ^= b ^= a ^= b;
```

Although the code above works fine for most of the cases, it tries to modify variable 'a' two times between sequence points, so the behavior is undefined. What this means is it wont work in all the cases. This will also not work for floating-point values. Also, think of a scenario where you have written your code like this

```
swap(int *a, int *b)
{
  *a ^= *b ^= *a ^= *b;
}
```

Now, if suppose, by mistake, your code passes the pointer to the same variable to this function. Guess what happens? Since Xor'ing an element with itself sets the variable to zero, this routine will end up setting the variable to zero (ideally it should have swapped the variable with itself). This scenario is quite possible in sorting algorithms which sometimes try to swap a variable with itself (maybe due to some small, but not so fatal coding error). One solution to this problem is to check if the numbers to be swapped are already equal to each other.

```
swap(int *a, int *b)
{
  if(*a!=*b)
  {
    *a ^= *b ^= *a ^= *b;
  }
}
```

Method2

This method is also quite popular

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

But, note that here also, if a and b are big and their addition is bigger than the size of an int, even this might end up giving you wrong results.

Method3

One can also swap two variables using a macro. However, it would be required to pass the type of the variable to the macro. Also, there is an interesting problem using macros. Suppose you have a swap macro which looks something like this

```
#define swap(type,a,b) type temp;temp=a;a=b;b=temp;
```

Now, think what happens if you pass in something like this

```
swap(int,temp,a) //You have a variable called "temp" (which is quite possible).
```

This is how it gets replaced by the macro

```
int temp;
temp=temp;
temp=b;
b=temp;
```

Which means it sets the value of "b" to both the variables!. It never swapped them! Scary, isn't it?

So the moral of the story is, dont try to be smart when writing code to swap variables. Use a temporary variable. Its not only fool proof, but also easier to understand and maintain.

What is the 8 queens problem? Write a C program to solve it.

Discuss it!

The 8 queens problem is a classic problem using the chess board. This problem is to place 8 queens on the chess board so that they do not attack each other horizontally, vertically or diagonally. It turns out that there are 12 essentially distinct solutions to this problem.

Suppose we have an array t[8] which keeps track of which column is occupied in which row of the chess board. That is, if t[0]==5, then it means that the queen has been placed in the fifth column of the first row. We need to couple the backtracking algorithm with a procedure that checks whether the tuple is completable or not, i.e. to check that the next placed queen 'i' is not menaced by any of the already placed 'j' (j < i):

```
Two queens are in the same column         if t[i]=t[j]
Two queens are in the same major diagonal if (t[i]-t[j])=(i-j)
two queens are in the same minor diagonal if (t[j]-t[i])=(i-j)
```

Here is some working C code to solve this problem using backtracking

```
#include<stdio.h>
static int t[10]={-1};
void queens(int i);
int empty(int i);

void print_solution();

int main()
{
  queens(1);
  print_solution();
  return(0);
}

void queens(int i)
{
  for(t[i]=1;t[i]<=8;t[i]++)
  {
    if(empty(i))
    {
       if(i==8)
       {
          print_solution();
          /* If this exit is commented, it will show ALL possible
combinations */
          exit(0);
       }
       else
       {
          // Recurse!
          queens(i+1);
       }

    }// if

  }// for
}



int empty(int i)
{
  int j;
  j=1;

  while(t[i]!=t[j] && abs(t[i]-t[j])!=(i-j) &&j<8)j++;

  return((i==j)?1:0);
}



void print_solution()
{
  int i;
  for(i=1;i<=8;i++)printf("\nt[%d] = [%d]",i,t[i]);
}
```

And here is one of the possible solutions

```
t[1] = [1] // This means the first square of the first row.
t[2] = [5] // This means the fifth square of the second row.
t[3] = [8] ..
t[4] = [6] ..
t[5] = [3] ..
t[6] = [7] ..
t[7] = [2] ..
t[8] = [4] // This means the fourth square of the last row.
```

Write a C program to print a square matrix helically.

Discuss it!

Here is a C program to print a matrix helically. Printing a matrix helically means printing it in this spiral fashion

```
>-----------+
             |
+---->--+   |
|       |   |
|       |   |
|   <---+   |
|           |
+-----------+
```

This is a simple program to print a matrix helically.

```
#include<stdio.h>

/* HELICAL MATRIX */

int main()
{
       int arr[][4] = { {1,2,3,4},
                        {5,6,7,8},
                        {9,10,11,12},
                        {13, 14, 15, 16}
                       };

       int i, j, k,middle,size;
       printf("\n\n");
       size = 4;

       for(i=size-1, j=0; i>0; i--, j++)
       {
              for(k=j; k<i; k++) printf("%d ", arr[j][k]);
              for(k=j; k<i; k++) printf("%d ", arr[k][i]);
              for(k=i; k>j; k--) printf("%d ", arr[i][k]);
              for(k=i; k>j; k--) printf("%d ", arr[k][j]);
       }

       middle = (size-1)/2;
       if (size % 2 == 1) printf("%d", arr[middle][middle]);
       printf("\n\n");
       return 1;
}
```

Write a C program to reverse a string

Discuss it!

There are a number of ways one can reverse strings. Here are a few of them. These should be enough to impress the interviewer! The methods span from recursive to non-recursive (iterative).

Also note that there is a similar question about reversing the words in a sentence, but still keeping the words in place. That is

I am a good boy

would become

boy good a am I

This is dealt with in another question. Here I only concentrate on reversing strings. That is

I am a good boy

would become

yob doog a ma I

Here are some sample C programs to do the same

Method1 (Recursive)

```
#include <stdio.h>

static char str[]="STRING TO REVERSE";

int main(int argc, char *argv)
{
   printf("\nOriginal string : [%s]", str);

   // Call the recursion function
   reverse(0);

   printf("\nReversed string : [%s]", str);
   return(0);
}

int reverse(int pos)
{
   // Here I am calculating strlen(str) everytime.
   // This can be avoided by doing this computation
   // earlier and storing it somewhere for later use.
   if(pos<(strlen(str)/2))
   {
      char ch;

      // Swap str[pos] and str[strlen(str)-pos-1]
      ch = str[pos];
      str[pos]=str[strlen(str)-pos-1];
      str[strlen(str)-pos-1]=ch;

      // Now recurse!
      reverse(pos+1);
   }
}
```

Method2

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

void ReverseStr ( char *buff, int start, int end )
{
   char tmp ;

   if ( start >= end )
   {
      printf ( "\n%s\n", buff );
      return;
   }

   tmp = *(buff + start);
   *(buff + start) = *(buff + end);
   *(buff + end) = tmp ;

   ReverseStr (buff, ++start, --end );
}


int main()
{
   char buffer[]="This is Test";
   ReverseStr(buffer,0,strlen(buffer)-1);
   return 0;
}
```

Method3

```
public static String reverse(String s)
{
    int N = s.length();
    if (N <= 1) return s;
    String left = s.substring(0, N/2);
    String right = s.substring(N/2, N);
    return reverse(right) + reverse(left);
}
```

Method4

```
for(int i = 0, j = reversed.Length - 1; i < j; i++, j--)
{
    char temp = reversed[i];
    reversed[i] = reversed[j];
    reversed[j] = temp;
}
return new String(reversed);
```

Method5

```
public static String reverse(String s)
{
   int N = s.length();
   String reverse = "";
   for (int i = 0; i < N; i++)
        reverse = s.charAt(i) + reverse;
   return reverse;
}
```

Method6

```
public static String reverse(String s)
{
   int N = s.length();
   char[] a = new char[N];
   for (int i = 0; i < N; i++)
       a[i] = s.charAt(N-i-1);
   String reverse = new String(a);
   return reverse;
}
```

Write a C program to reverse the words in a sentence in place.

Discuss it!

That is, given a sentence like this

I am a good boy

The in place reverse would be

boy good a am I

Method1

First reverse the whole string and then individually reverse the words

```
I am a good boy
<------------->

yob doog  a   ma   I
<-> <---> <->  <-> <->

boy good a am I
```

Here is some C code to do the same ....

```
/*
  Algorithm..

  1. Reverse whole sentence first.
  2. Reverse each word individually.

  All the reversing happens in-place.
*/

#include <stdio.h>

void rev(char *l, char *r);


int main(int argc, char *argv[])
{
   char buf[] = "the world will go on forever";
   char *end, *x, *y;
   // Reverse the whole sentence first..
   for(end=buf; *end; end++);
   rev(buf,end-1);


   // Now swap each word within sentence...
   x = buf-1;
   y = buf;

   while(x++ < end)
   {
      if(*x == '\0' || *x == ' ')
      {
        rev(y,x-1);
        y = x+1;
      }
   }

   // Now print the final string....
   printf("%s\n",buf);

   return(0);
}


// Function to reverse a string in place...
void rev(char *l,char *r)
{
   char t;
   while(l<r)
   {
      t    = *l;
      *l++ = *r;
      *r-- = t;
   }
}
```

Method2
Another way to do it is, allocate as much memory as the input for the final output. Start from the right of the string and copy the words one by one to the output.

```
Input  : I am a good boy
                     <--
                <-------
              <---------
           <------------
         <--------------


Output : boy
```

```
       : boy good
       : boy good a
       : boy good a am
       : boy good a am I
```

The only problem to this solution is the extra space required for the output and one has to write this code really well as we are traversing the string in the reverse direction and there is no null at the start of the string to know we have reached the start of the string!. One can use the strtok() function to breakup the string into multiple words and rearrange them in the reverse order later.

Method3

Create a linked list like

```
+---+    +----------+    +----+    +----------+    +---+    +----------
+
| I | -> | <spaces> | -> | am | -> | <spaces> | -> | a | -> | <spaces>
| --+
+---+    +----------+    +----+    +----------+    +---+    +----------
+   |

    |

    |
+-----------------------------------------------------------------------
---+
|
|     +------+    +----------+    +-----+    +------+
+---> | good | -> | <spaces> | -> | boy | -> | NULL |
       +------+    +----------+    +-----+    +------+
```

Now its a simple question of reversing the linked list!. There are plenty of algorithms to reverse a linked list easily. This also keeps track of the number of spaces between the words. Note that the linked list algorithm, though inefficient, handles multiple spaces between the words really well.

I really dont know what is the use of reversing a string or a sentence like this!, but its still asked. Can someone tell me a really practical application of this? Please!

Write a C program generate permutations.

Discuss it!

Iterative C program

```
#include <stdio.h>
#define SIZE 3
int main(char *argv[],int argc)
{
  char list[3]={'a','b','c'};
  int i,j,k;

  for(i=0;i<SIZE;i++)
    for(j=0;j<SIZE;j++)
      for(k=0;k<SIZE;k++)
        if(i!=j && j!=k && i!=k)
          printf("%c%c%c\n",list[i],list[j],list[k]);

  return(0);
}
```

Recursive C program

```
#include <stdio.h>
#define N  5


int main(char *argv[],int argc)
{
  char list[5]={'a','b','c','d','e'};
  permute(list,0,N);
  return(0);
}


void permute(char list[],int k, int m)
{
  int i;
  char temp;

  if(k==m)
  {
    /* PRINT A FROM k to m! */
    for(i=0;i<N;i++){printf("%c",list[i]);}
    printf("\n");
  }
  else
  {
     for(i=k;i<m;i++)
     {
        /* swap(a[i],a[m-1]); */
        temp=list[i];
        list[i]=list[m-1];
        list[m-1]=temp;

        permute(list,k,m-1);
        /* swap(a[m-1],a[i]); */

        temp=list[m-1];
        list[m-1]=list[i];
        list[i]=temp;
       }
  }
}
```

Write a C program for calculating the factorial of a number

Discuss it!

Here is a recursive C program

```
fact(int n)
{
    int fact;
    if(n==1)
        return(1);
    else
        fact = n * fact(n-1);
    return(fact);
}
```

Please note that there is no error handling added to this function (to check if n is negative or 0. Or if n is too large for the system to handle). This is true for most of the answers in this website. Too much error handling and standard compliance results in a lot of clutter making it difficult to concentrate on the crux of the solution. You must ofcourse add as much error handling and comply to the standards of your compiler when you actually write the code to implement these algorithms.

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

Discuss it!

There are again different methods to do this in C

Brute force C program

```
int pow(int x, int y)
{
  if(y == 1) return x ;
  return x * pow(x, y-1) ;
}
```

Divide and Conquer C program

```
#include <stdio.h>
int main(int argc, char*argv[])
{
  printf("\n[%d]\n",pow(5,4));
}

int pow(int x, int n)
{
  if(n==0)return(1);
  else if(n%2==0)
  {
    return(pow(x,n/2)*pow(x,(n/2)));
  }
  else
  {
    return(x*pow(x,n/2)*pow(x,(n/2)));
  }
}
```

Also, the code above can be optimized still by calculating pow(z, (n/2)) only one time (instead of twice) and using its value in the two return() expressions above.

Write a C program which does wildcard pattern matching algorithm

Discuss it!

Here is an example C program...

```
#include<stdio.h>
#define TRUE 1
#define FALSE 0

int wildcard(char *string, char *pattern);

int main()
{
  char *string = "hereheroherr";
  char *pattern = "*hero*";

  if(wildcard(string, pattern)==TRUE)
  {
    printf("\nMatch Found!\n");
  }
  else
  {
    printf("\nMatch not found!\n");
  }
  return(0);
}

int wildcard(char *string, char *pattern)
{
  while(*string)
  {
     switch(*pattern)
     {
        case '*': do {++pattern;}while(*pattern == '*');
                  if(!*pattern) return(TRUE);
                  while(*string){if(wildcard(pattern,string++)==TRUE)re
turn(TRUE);}
                  return(FALSE);
        default : if(*string!=*pattern)return(FALSE); break;
     }
     ++pattern;
     ++string;
   }

   while (*pattern == '*') ++pattern;
   return !*pattern;
}
```

How do you calculate the maximum subarray of a list of numbers?

Discuss it!

```
This is a very popular question

You are given a large array X of integers (both positive and negative)
and you need to find the
maximum sum found in any contiguous subarray of X.


Example X = [11, -12, 15, -3, 8, -9, 1, 8, 10, -2]
Answer is 30.



There are various methods to solve this problem, some are listed below
```

Brute force

```
maxSum = 0
for L = 1 to N
{
  for R = L to N
  {
     sum = 0
     for i = L to R
     {
        sum = sum + X[i]
     }
     maxSum = max(maxSum, sum)
   }
}
```

O(N^3)

Quadratic

Note that sum of [L..R] can be calculated from sum of [L..R-1] very easily.

```
maxSum = 0
for L = 1 to N
{
  sum = 0
  for R = L to N
  {
    sum = sum + X[R]
    maxSum = max(maxSum, sum)
  }
}
```

Using divide-and-conquer

O(N log(N))

```
maxSum(L, R)
{
  if L > R then
    return 0

  if L = R then
    return max(0, X[L])

  M = (L + R)/2
  sum = 0; maxToLeft = 0
  for i = M downto L do
  {
     sum = sum + X[i]
     maxToLeft = max(maxToLeft, sum)
  }

  sum = 0; maxToRight = 0
  for i = M to R do
  {
     sum = sum + X[i]
     maxToRight = max(maxToRight, sum)
  }


  maxCrossing = maxLeft + maxRight
  maxInA = maxSum(L,M)
  maxInB = maxSum(M+1,R)
  return max(maxCrossing, maxInA, maxInB)
}
```

Here is working C code for all the above cases

```
#include<stdio.h>
#define N 10
int maxSubSum(int left, int right);
int list[N] = {11, -12, 15, -3, 8, -9, 1, 8, 10, -2};

int main()
{
  int i,j,k;
  int maxSum, sum;

  /*---------------------------------------
  * CUBIC - O(n*n*n)
  *---------------------------------------*/

  maxSum = 0;
  for(i=0; i<N; i++)
  {
    for(j=i; j<N; j++)
    {
      sum = 0;
      for(k=i ; k<j; k++)
      {
        sum = sum + list[k];
      }
      maxSum = (maxSum>sum)?maxSum:sum;
    }
   }
```

```
    printf("\nmaxSum = [%d]\n", maxSum);


    /*-----------------------------------
     * Quadratic - O(n*n)
     * ---------------------------------- */

    maxSum = 0;
    for(i=0; i<N; i++)
    {
       sum=0;
       for(j=i; j<N ;j++)
       {
          sum = sum + list[j];
          maxSum = (maxSum>sum)?maxSum:sum;
       }
    }

    printf("\nmaxSum = [%d]\n", maxSum);


    /*--------------------------------------
     * Divide and Conquer - O(nlog(n))
     * ------------------------------------- */

    printf("\nmaxSum : [%d]\n", maxSubSum(0,9));

    return(0);
 }


 int maxSubSum(int left, int right)
 {
   int mid, sum, maxToLeft, maxToRight, maxCrossing, maxInA, maxInB;
   int i;

   if(left>right){return 0;}
   if(left==right){return((0>list[left])?0:list[left]);}
   mid = (left + right)/2;

   sum=0;
   maxToLeft=0;
   for(i=mid; i>=left; i--)
   {
      sum = sum + list[i];
      maxToLeft = (maxToLeft>sum)?maxToLeft:sum;
   }


   sum=0;
   maxToRight=0;
   for(i=mid+1; i<=right; i++)
   {
       sum = sum + list[i];
       maxToRight = (maxToRight>sum)?maxToRight:sum;
   }
```

```
  maxCrossing = maxToLeft + maxToRight;
  maxInA = maxSubSum(left,mid);
  maxInB = maxSubSum(mid+1,right);
  return(((maxCrossing>maxInA)?maxCrossing:maxInA)>maxInB?((maxCrossing
>maxInA)?maxCrossing:maxInA):maxInB);
}
```

Note that, if the array has all negative numbers, then this code will return 0. This is wrong because it should return the maximum sum, which is the least negative integer in the array. This happens because we are setting maxSum to 0 initially. A small change in this code can be used to handle such cases.

How to generate fibonacci numbers? How to find out if a given number is a fibonacci number or not? Write C programs to do both.

Discuss it!

Lets first refresh ourselves with the Fibonacci sequence

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Fibonacci numbers obey the following rule

F(n) = F(n-1) + F(n-2)

Here is an iterative way to generate fibonacci numbers and also return the nth number.

```
int fib(int n)
{
   int f[n+1];
   f[1] = f[2] = 1;

   printf("\nf[1] = %d", f[1]);
   printf("\nf[2] = %d", f[2]);

   for (int i = 3; i <= n; i++)
   {
       f[i] = f[i-1] + f[i-2];
       printf("\nf[%d] = [%d]",i,f[i]);
   }
   return f[n];
}
```

Here is a recursive way to generate fibonacci numbers.

```
int fib(int n)
{
  if (n <= 2) return 1
  else return fib(n-1) + fib(n-2)
}
```

Here is an iterative way to just compute and return the nth number (without storing the previous numbers).

```
int fib(int n)
{
  int a = 1, b = 1;
  for (int i = 3; i <= n; i++)
  {
     int c = a + b;
     a = b;
     b = c;
  }
  return a;
}
```

There are a few slick ways to generate fibonacci numbers, a few of them are listed below

Method1

If you know some basic math, its easy to see that

```
         n
[ 1 1 ]      =   [ F(n+1) F(n)   ]
[ 1 0 ]          [ F(n)   F(n-1) ]
```

or

```
(f(n) f(n+1)) [ 0 1 ] = (f(n+1) f(n+2))
              [ 1 1 ]
```

or

```
                   n
(f(0) f(1)) [ 0 1 ]  = (f(n) f(n+1))
            [ 1 1 ]
```

The n-th power of the 2 by 2 matrix can be computed efficiently in O(log n) time. This implies an O(log n) algorithm for computing the n-th Fibonacci number.

Here is the pseudocode for this

```
int Matrix[2][2] = {{1,0}{0,1}}

int fib(int n)
{
    matrixpower(n-1);
    return Matrix[0][0];
}

void matrixpower(int n)
{
   if (n > 1)
   {
      matrixpower(n/2);
      Matrix = Matrix * Matrix;
   }
   if (n is odd)
   {
      Matrix = Matrix * {{1,1}{1,0}}
   }
}
```

And here is a program in C which calculates fib(n)

```
#include<stdio.h>

int M[2][2]={{1,0},{0,1}};
int A[2][2]={{1,1},{1,0}};
int C[2][2]={{0,0},{0,0}};  // Temporary matrix used for
multiplication.
```

```
void matMul(int n);
void mulM(int m);

int main()
{
   int n;
   n=6;

   matMul(n-1);

   // The nth fibonacci will be stored in M[0][0]
   printf("\n%dth Fibonaci number : [%d]\n\n", n, M[0][0]);
   return(0);

}


// Recursive function with divide and conquer strategy

void matMul(int n)
{
   if(n>1)
   {
     matMul(n/2);
     mulM(0);     // M * M
   }
   if(n%2!=0)
   {
    mulM(1);     //  M * {{1,1}{1,0}}
   }
}


// Function which does some basic matrix multiplication.

void mulM(int m)
{
   int i,j,k;

   if(m==0)
   {
     // C = M * M

     for(i=0;i<2;i++)
      for(j=0;j<2;j++)
      {
        C[i][j]=0;
        for(k=0;k<2;k++)
           C[i][j]+=M[i][k]*M[k][j];
      }
   }
   else
   {
     // C = M *  {{1,1}{1,0}}

     for(i=0;i<2;i++)
```

```
      for(j=0;j<2;j++)
      {
        C[i][j]=0;
        for(k=0;k<2;k++)
         C[i][j]+=A[i][k]*M[k][j];
      }
   }

   // Copy back the temporary matrix in the original matrix M

   for(i=0;i<2;i++)
     for(j=0;j<2;j++)
     {
       M[i][j]=C[i][j];
     }
}
```

Method2

f(n) = (1/sqrt(5)) * (((1+sqrt(5))/2) ^ n - ((1-sqrt(5))/2) ^ n)

So now, how does one find out if a number is a fibonacci or not?.

The cumbersome way is to generate fibonacci numbers till this number and see if this number is one of them. But there is another slick way to check if a number is a fibonacci number or not.

N is a Fibonacci number if and only if (5*N*N + 4) or (5*N*N - 4) is a perfect square!

Dont believe me?

3 is a Fibonacci number since (5*3*3 + 4) is 49 which is 7*7
5 is a Fibonacci number since (5*5*5 - 4) is 121 which is 11*11
4 is not a Fibonacci number since neither (5*4*4 + 4) = 84 nor (5*4*4 - 4) = 76 are perfect squares.

To check if a number is a perfect square or not, one can take the square root, round it to the nearest integer and then square the result. If this is the same as the original whole number then the original was a perfect square.

Solve the Rat In A Maze problem using backtracking.

Discuss it!

This is one of the classical problems of computer science. There is a rat trapped in a maze. There are multiple paths in the maze from the starting point to the ending point. There is some cheese at the exit. The rat starts from the entrance of the maze and wants to get to the cheese.

This problem can be attacked as follows.

Have a m*m matrix which represents the maze.

For the sake of simplifying the implementation, have a boundary around your matrix and fill it up with all ones. This is so that you know when the rat is trying to go out of the boundary of the maze. In the real world, the rat would know not to go out of the maze, but hey! So, initially the matrix (I mean, the maze) would be something like (the ones represent the "exra" boundary we have added). The ones inside specify the obstacles.

```
111111111111111111111
100000000000000000001
100000010000000000001
100000010000000000001
100000000100001000001
100001000010000000001
100000000100000000001
100000000000000000001
111111111111111111111
```

The rat can move in four directions at any point in time (well, right, left, up, down). Please note that the rat can't move diagonally. Imagine a real maze and not a matrix. In matrix language

Moving right means adding {0,1} to the current coordinates.

Moving left means adding {0,-1} to the current coordinates.

Moving up means adding {-1,0} to the current coordinates.

Moving right means adding {1,0} to the current coordinates.

The rat can start off at the first row and the first column as the entrance point.

From there, it tries to move to a cell which is currently free. A cell is free if it has a zero in it.

It tries all the 4 options one-by-one, till it finds an empty cell. If it finds one, it moves to that cell and marks it with a 1 (saying it has visited it once). Then it continues to move ahead from that cell to other cells.

If at a particular cell, it runs out of all the 4 options (that is it cant move either right, left, up or down), then it needs to backtrack. It backtracks till a point where it can move ahead and be closer to the exit.

If it reaches the exit point, it gets the cheese, ofcourse.

The complexity is O(m*m).

Here is some pseudocode to chew upon

```
findpath()
{
  Position offset[4];
  Offset[0].row=0; offset[0].col=1;//right;
  Offset[1].row=1; offset[1].col=0;//down;
  Offset[2].row=0; offset[2].col=-1;//left;
  Offset[3].row=-1; offset[3].col=0;//up;

  // Initialize wall of obstacles around the maze
  for(int i=0; i<m+1;i++)
    maze[0][i] = maze[m+1][i]=1; maze[i][0] = maze[i][m+1]=1;

  Position here;
  Here.row=1;
  Here.col=1;

  maze[1][1]=1;
  int option = 0;
  int lastoption = 3;

  while(here.row!=m || here.col!=m)
  {
     //Find a neighbor to move
     int r,c;

       while (option<=LastOption)
       {
          r=here.row + offset[position].row;
          c=here.col + offset[option].col;
          if(maze[r][c]==0)break;
```

```
         option++;
      }

      //Was a neighbor found?
      if(option<=LastOption)
      {
        path->Add(here);
        here.row=r;here.col=c;
        maze[r][c]=1;
        option=0;
      }
      else
      {
         if(path->Empty())return(False);
         Position  next;
         Path->Delete(next);
         If(new.row==here.row)
            Option=2+next.col - here.col;
         Else { option = 3 + next.row - here.col;}
            Here=next;
      }
      return(TRUE);
   }
}
```

What Little-Endian and Big-Endian? How can I determine whether a machine's byte order is big-endian or little endian? How can we convert from one to another?

Discuss it!

```
First of all, Do you know what Little-Endian and Big-Endian mean?

Little Endian means that the lower order byte of the number is stored
in memory at the lowest address, and the higher order byte is stored at
the highest address. That is, the little end comes first.

For example, a 4 byte, 32-bit integer


    Byte3 Byte2 Byte1 Byte0


will be arranged in memory as follows:


    Base_Address+0   Byte0
    Base_Address+1   Byte1
    Base_Address+2   Byte2
    Base_Address+3   Byte3


Intel processors use "Little Endian" byte order.
```

"Big Endian" means that the higher order byte of the number is stored in memory at the lowest address, and the lower order byte at the highest address. The big end comes first.

```
  Base_Address+0   Byte3
  Base_Address+1   Byte2
  Base_Address+2   Byte1
  Base_Address+3   Byte0
```

Motorola, Solaris processors use "Big Endian" byte order.

In "Little Endian" form, code which picks up a 1, 2, 4, or longer byte number proceed in the same way for all formats. They first pick up the lowest order byte at offset 0 and proceed from there. Also, because of the 1:1 relationship between address offset and byte number (offset 0 is byte 0), multiple precision mathematic routines are easy to code. In "Big Endian" form, since the high-order byte comes first, the code can test whether the number is positive or negative by looking at the byte at offset zero. Its not required to know how long the number is, nor does the code have to skip over any bytes to find the byte containing the sign information. The numbers are also stored in the order in which they are printed out, so binary to decimal routines are particularly efficient.

Here is some code to determine what is the type of your machine

```
int num = 1;
if(*(char *)&num == 1)
{
  printf("\nLittle-Endian\n");
}
else
{
  printf("Big-Endian\n");
}
```

And here is some code to convert from one Endian to another.

```
int myreversefunc(int num)
{
  int byte0, byte1, byte2, byte3;

  byte0 = (num & x000000FF) >>  0 ;
  byte1 = (num & x0000FF00) >>  8 ;
  byte2 = (num & x00FF0000) >> 16 ;
  byte3 = (num & xFF000000) >> 24 ;
  return((byte0 << 24) | (byte1 << 16) | (byte2 << 8) | (byte3 << 0));
}
```

Write C code to solve the Tower of Hanoi problem.

Discuss it!

Here is an example C program to solve the Tower Of Hanoi problem...

```
main()
{
    towers_of_hanio(n,'L','R','C');
}

towers_of_hanio(int n, char from, char to, char temp)
{
    if(n>0)
    {
        tower_of_hanio(n-1, from, temp, to);
        printf("\nMove disk %d from %c to %c\n", n, from, to);
        tower_of_hanio(n-1, temp, to, from);
    }
}
```

Write C code to return a string from a function

Discuss it!

This is one of the most popular interview questions

This C program wont work!

```
char *myfunction(int n)
{
   char buffer[20];
   sprintf(buffer, "%d", n);
   return retbuf;
}
```

This wont work either!

```
char *myfunc1()
{
  char temp[] = "string";
  return temp;
}


char *myfunc2()
{
   char temp[] = {'s', 't', 'r', 'i', 'n', 'g', '\0'};
   return temp;
}


int main()
{
    puts(myfunc1());
    puts(myfunc2());
}
```

The returned pointer should be to a static buffer (like static char buffer[20];), or to a buffer passed in by the caller function, or to memory obtained using malloc(), but not to a local array.

This will work

```
char *myfunc()
{
   char *temp = "string";
   return temp;
}

int main()
{
   puts(someFun());
}
```

So will this

```
calling_function()
{
  char *string;
  return_string(&string);
  printf(?\n[%s]\n?, string);
}

boolean return_string(char **mode_string /*Pointer to a pointer! */)
{
   *string = (char *) malloc(100 * sizeof(char));
   DISCARD strcpy((char *)*string, (char *)?Something?);
}
```

Write C code to dynamically allocate one, two and three dimensional arrays (using malloc())

Discuss it!

Its pretty simple to do this in the C language if you know how to use C pointers. Here are some example C code snipptes....

One dimensional array

```
int *myarray = malloc(no_of_elements * sizeof(int));

//Access elements as myarray[i]
```

Two dimensional array

Method1

```
int **myarray = (int **)malloc(no_of_rows * sizeof(int *));
for(i = 0; i < no_of_rows; i++)
{
myarray[i] = malloc(no_of_columns * sizeof(int));
}

// Access elements as myarray[i][j]
```

Method2 (keep the array's contents contiguous)

```
int **myarray = (int **)malloc(no_of_rows * sizeof(int *));
myarray[0] = malloc(no_of_rows * no_of_columns * sizeof(int));

for(i = 1; i < no_of_rows; i++)
  myarray[i] = myarray[0] + (i * no_of_columns);

// Access elements as myarray[i][j]
```

Method3

```
int *myarray = malloc(no_of_rows * no_of_columns * sizeof(int));

// Access elements using myarray[i * no_of_columns + j].
```

Three dimensional array

```
#define MAXX 3
#define MAXY 4
#define MAXZ 5

main()
{
    int ***p,i,j;
    p=(int ***) malloc(MAXX * sizeof(int ***));

    for(i=0;i<MAXX;i++)
    {
        p[i]=(int **)malloc(MAXY * sizeof(int *));
        for(j=0;j<MAXY;j++)
            p[i][j]=(int *)malloc(MAXZ * sizeof(int));
    }

    for(k=0;k<MAXZ;k++)
        for(i=0;i<MAXX;i++)
            for(j=0;j<MAXY;j++)
                p[i][j][k]=<something>;

}
```

How do you initialize a pointer inside a function?

Discuss it!

This is one of the very popular interview questions, so take a good look at it!.

```
myfunction(int *ptr)
{
  int myvar = 100;
  ptr = &myvar;
}

main()
{
int *myptr;
myfunction(myptr);

//Use pointer myptr.

}
```

Do you think this works? It does not!.

Arguments in C are passed by value. The called function changed the passed copy of the pointer, and not the actual pointer.

There are two ways around this problem

Method1

Pass in the address of the pointer to the function (the function needs to accept a pointer-to-a-pointer).

```
calling_function()
{
  char *string;
  return_string(/* Pass the address of the pointer */&string);
  printf(?\n[%s]\n?, string);
}

boolean return_string(char **mode_string /*Pointer to a pointer! */)
{
  *string = (char *) malloc(100 * sizeof(char)); // Allocate memory to
the pointer passed, not its copy.
  DISCARD strcpy((char *)*string, (char *)?Something?);
}
```

Method2

Make the function return the pointer.

```
char *myfunc()
{
  char *temp = "string";
  return temp;
}

int main()
{
  puts(myfunc());
}
```

Find the maximum of three integers using the ternary operator.

Discuss it!

Here is how you do it

```
max = ((a>b)?((a>c)?a:c):((b>c)?b:c));
```

```
Here is another way


max = ((a>b)?a:b)>c?((a>b)?a:b):c;
```

```
Here is some code to find the max of 4 numbers...

Method1


#include <stdio.h>
#include <stdlib.h>

#define max2(x,y)  ((x)>(y)?(x):(y))
#define max4(a,b,c,d)  max2(max2((a),(b)),max2((c),(d)))

int main ( void )
{
    printf ( "Max: %d\n", max4(10,20,30,40));
    printf ( "Max: %d\n", max4(10,0,3,4));
    return EXIT_SUCCESS;
}
```

```
Method2


#include <stdio.h>
#include <stdlib.h>

int retMax(int i1, int i2, int i3, int i4)
{
   return(((i1>i2)?i1:i2) > ((i3>i4)?i3:i4)?
((i1>i2)?i1:i2):((i3>i4)?i3:i4));
}

int main()
{
   int val = 0 ;

   val = retMax(10, 200, 10, 530);
   val = retMax(9, 2, 5, 7);

   return 0;
}
```

Write code to remove duplicates in a sorted array.

Discuss it!

There are a number of ways to remove duplicates from a sorted array.
Here are a few C programs...

Method1

In this simple C program, we change the original array and also send
the new size of the array back to the caller.

```
#include <stdio.h>

int removeDuplicates(int a[], int array_size);

// The main function
int main()
{

  // Different test cases..
  int my_array1[]    = {1, 1, 2, 3, 5, 6, 6, 7, 10, 25, 100, 123, 123};
  int my_array1_size = 13;

  int my_array2[]    = {1, 2, 3, 5, 6};
  int my_array2_size = 5;

  int my_array3[]    = {1, 1, 1, 1, 1};
  int my_array3_size = 5;

  int my_array4[]    = {123, 123};
  int my_array4_size = 2;

  int my_array5[]    = {1, 123, 123};
  int my_array5_size = 3;

  int my_array6[]    = {123, 123, 166};
  int my_array6_size = 3;

  int my_array7[]    = {1, 2, 8, 8 , 24, 60, 60, 60, 60, 75, 100, 100,
123};
  int my_array7_size = 13;


  my_array1_size = removeDuplicates(my_array1, my_array1_size);
  my_array2_size = removeDuplicates(my_array2, my_array2_size);
  my_array3_size = removeDuplicates(my_array3, my_array3_size);
  my_array4_size = removeDuplicates(my_array4, my_array4_size);
  my_array5_size = removeDuplicates(my_array5, my_array5_size);
  my_array6_size = removeDuplicates(my_array6, my_array6_size);
  my_array7_size = removeDuplicates(my_array7, my_array7_size);

  return(0);
}


// Function to remove the duplicates
```

```
int removeDuplicates(int a[], int array_size)
{
  int i, j;

  j = 0;


  // Print old array...
  printf("\n\nOLD : ");
  for(i = 0; i < array_size; i++)
  {
     printf("[%d] ", a[i]);
  }


  // Remove the duplicates ...
  for (i = 1; i < array_size; i++)
  {
    if (a[i] != a[j])
    {
      j++;
      a[j] = a[i]; // Move it to the front
    }
}

  // The new array size..
  array_size = (j + 1);


  // Print new array...
  printf("\n\nNEW : ");
  for(i = 0; i< array_size; i++)
  {
     printf("[%d] ", a[i]);
  }
  printf("\n\n");



  // Return the new size...
  return(j + 1);
}
```

And here is the output...

```
OLD : [1] [1] [2] [3] [5] [6] [6] [7] [10] [25] [100] [123] [123]
NEW : [1] [2] [3] [5] [6] [7] [10] [25] [100] [123]


OLD : [1] [2] [3] [5] [6]
NEW : [1] [2] [3] [5] [6]
```

```
OLD : [1] [1] [1] [1] [1]
NEW : [1]


OLD : [123] [123]
NEW : [123]


OLD : [1] [123] [123]
NEW : [1] [123]


OLD : [123] [123] [166]
NEW : [123] [166]


OLD : [1] [2] [8] [8] [24] [60] [60] [60] [60] [75] [100] [100] [123]
NEW : [1] [2] [8] [24] [60] [75] [100] [123]
```

Method2

If we dont want to change the input array and just want to print the array without any duplicates, the solution is very simple. Check out the removeDuplicatesNoModify() function in the program below. It keeps a track of the most recently seen number and does not print any duplicates of it when traversing the sorted array.

```c
#include <stdio.h>

void removeDuplicatesNoModify(int my_array[], int my_array1_size);
void print_array(int array[], int array_size, int current_pos, int
dup_start, int dup_end);


// The main function
int main()
{
  // Different inputs...
  int my_array1[]    = {1, 1, 2, 3, 5, 6, 6, 7, 10, 25, 100, 123, 123};
  int my_array1_size = 13;

  int my_array2[]    = {1, 2, 3, 5, 6};
  int my_array2_size = 5;

  int my_array3[]    = {1, 1, 1, 1, 1};
  int my_array3_size = 5;

  int my_array4[]    = {123, 123};
  int my_array4_size = 2;
  int my_array5[]    = {1, 123, 123};
  int my_array5_size = 3;

  int my_array6[]    = {123, 123, 166};
  int my_array6_size = 3;

  int my_array7[]    = {1, 2, 8, 8 , 24, 60, 60, 60, 60, 75, 100, 100,
123};
  int my_array7_size = 13;


removeDuplicatesNoModify(my_array1, my_array1_size);
  removeDuplicatesNoModify(my_array2, my_array2_size);
  removeDuplicatesNoModify(my_array3, my_array3_size);
  removeDuplicatesNoModify(my_array4, my_array4_size);
  removeDuplicatesNoModify(my_array5, my_array5_size);
  removeDuplicatesNoModify(my_array6, my_array6_size);
  removeDuplicatesNoModify(my_array7, my_array7_size);

  return(0);
}



//  This function just prints the array without duplicates.
//  It does not modify the original array!

void removeDuplicatesNoModify(int array[], int array_size)
{
   int i, last_seen_unique;

   if(array_size <= 1){return;}

   last_seen_unique = array[0];

   printf("\n\nOld : ", array_size);

   for(i = 0; i < array_size; i++)
   {
     printf("[%2d] ", array[i]);
   }

   printf("\nNew : ", array_size);

   printf("[%2d] ", array[0]);
   for(i=1; i < array_size; i++)
   {
       if(array[i]!=last_seen_unique)
       {
          printf("[%2d] ", array[i]);
          last_seen_unique = array[i];
       }
   }

   printf("\n");
```

```
}
```

And here is the output..

```
Old : [ 1] [ 1] [ 2] [ 3] [ 5] [ 6] [ 6] [ 7] [10] [25] [100] [123]
[123]
New : [ 1] [ 2] [ 3] [ 5] [ 6] [ 7] [10] [25] [100] [123]


Old : [ 1] [ 2] [ 3] [ 5] [ 6]
New : [ 1] [ 2] [ 3] [ 5] [ 6]


Old : [ 1] [ 1] [ 1] [ 1] [ 1]
New : [ 1]


Old : [123] [123]
New : [123]


Old : [ 1] [123] [123]
New : [ 1] [123]


Old : [123] [123] [166]
New : [123] [166]


Old : [ 1] [ 2] [ 8] [ 8] [24] [60] [60] [60] [60] [75] [100] [100]
[123]
New : [ 1] [ 2] [ 8] [24] [60] [75] [100] [123]
```

Method3

Here is a slightly compilcated, but more visual version of the removeDuplicates() function. It shrinks the original array as and when it find duplicates. It is also optimized to identify continuous strings of duplicates and eliminate them at one shot.

```
#include <stdio.h>

void removeDuplicates(int array[], int *array_size) ;
void print_array(int array[], int array_size, int current_pos, int
dup_start, int dup_end);
```

```
// The main function
int main()
{
  // Different inputs...
  int my_array1[]    = {1, 1, 2, 3, 5, 6, 6, 7, 10, 25, 100, 123, 123};
  int my_array1_size = 13;

  int my_array2[]    = {1, 2, 3, 5, 6};
  int my_array2_size = 5;

int my_array3[]    = {1, 1, 1, 1, 1};
  int my_array3_size = 5;

  int my_array4[]    = {123, 123};
  int my_array4_size = 2;

  int my_array5[]    = {1, 123, 123};
  int my_array5_size = 3;

  int my_array6[]    = {123, 123, 166};
  int my_array6_size = 3;

  int my_array7[]    = {1, 2, 8, 8 , 24, 60, 60, 60, 60, 75, 100, 100,
123};
  int my_array7_size = 13;

  removeDuplicates(my_array1, &my_array1_size);
  removeDuplicates(my_array2, &my_array2_size);
  removeDuplicates(my_array3, &my_array3_size);
  removeDuplicates(my_array4, &my_array4_size);
  removeDuplicates(my_array5, &my_array5_size);
  removeDuplicates(my_array6, &my_array6_size);
  removeDuplicates(my_array7, &my_array7_size);

  return(0);
}




// Changes the original array and resets the size of the array if
duplicates
// have been removed.

void removeDuplicates(int array[], int *array_size)
{
   int i, j, k, l;
   int current_pos;
   int dup_start;
   int dup_end;

   printf("\nInitial array (size : [%d])\n\n", *array_size);
   for(i = 0; i < *array_size; i++)
   {
     printf("[%2d] ", array[i]);
   }
   printf("\n\n\n------------------------------------------------\n");


   if(*array_size == 1){return;}


   // Remove the dups...
   for (i = 0; (i < *array_size); i++)
   {
      //Start with the next element in the array and check if its a
duplicate...
      for(j = i+1; (j < *array_size); j++)
      {
         if(array[i]!=array[j])
         {
            // No duplicate, just continue...
            break;
         }
         else
         {
            // The next element is a duplicate.
            // See if there are more duplicates, as we want to optimize
here.
            //
            // That is, if we have something like
            //
            // Array : [1, 1, 1, 2]
            //
            // then, we want to copy 2 directly in the second position
and reduce the
            // array to
            //
            // Array : [1, 2].
            //
            // in a single iteration.

            current_pos = i;
            dup_start = j;

            j++;

            while((array[i]==array[j]) && (j < *array_size))
            {
              j++;
            }

            dup_end = j-1;

            print_array(array, *array_size, current_pos, dup_start,
dup_end);

            // Now remove elements of the array from "dup_start" to
"dup_end"
            // and shrink the size of the array.
            for(k = (dup_end + 1), l = dup_start ; k < *array_size;)
            {
               array[l++]=array[k++];
            }

            // Reduce the array size by the number of elements removed.
            *array_size = *array_size - (dup_end - dup_start + 1);

         }
      }
   }

   printf("\n\n---------------------------------------------");
   printf("\n\nFinal array (size : [%d])\n\n", *array_size);
   for(i = 0; i < *array_size; i++)
   {
     printf("[%2d] ", array[i]);
   }
   printf("\n\n");

}




// This function prints the array with some special pointers to the
numbers that
// are duplicated.
//
// Dont bother too much about this function, it just helps in
understanding
// how and where the duplicates are being removed from.

void print_array(int array[], int array_size, int current_pos, int
dup_start, int dup_end)
{
  int i;

  printf("\n\n");

  for(i = 0; i < array_size; i++)
  {
    printf("[%2d] ", array[i]);
  }

  printf("\n");

  for(i = 0; i < array_size; i++)
  {
    if((i == current_pos) ||
       (i == dup_start && i == dup_end) ||
       ((i == dup_start || i == dup_end) && (dup_start != dup_end)))
    {
       printf("  ^  ");
    }
    else
```

```
      {
         printf("     ");
      }
   }

   printf("\n");

   for(i = 0; i < array_size; i++)
   {
      if((i == current_pos) ||
         (i == dup_start && i == dup_end) ||
         ((i == dup_start || i == dup_end) && (dup_start != dup_end)))
      {
         printf("  |  ");
      }
      else
      {
         printf("     ");
      }
   }

   printf("\n");

   for(i = 0; i < array_size; i++)
   {
      if(i == current_pos)
      {
         printf("  C  ");
      }
      else if(i == dup_start && i == dup_end)
      {
         printf(" S/E ");
      }
      else if((i == dup_start || i == dup_end) && (dup_start != dup_end))
      {
         if(i == dup_start)
         {
            printf("  S--");
         }
         else
         {
            printf("--E  ");
         }
      }
      else if(i>dup_start && i<dup_end)
      {
         printf("-----");
      }
      else
      {
         printf("     ");
      }
   }

}
```

And here is the output (for one of the inputs)...

```
C - Current position.
S - Start of duplicates.
E - End of duplicates.



Old : [ 1] [ 2] [ 8] [ 8] [24] [60] [60] [60] [60] [75] [100] [100]
[123]

----------------------------------------------------------------------
---
[ 1] [ 2] [ 8] [ 8] [24] [60] [60] [60] [60] [75] [100] [100] [123]
            ^    ^
            |    |
            C   S/E


[ 1] [ 2] [ 8] [24] [60] [60] [60] [60] [75] [100] [100] [123]
                      ^    ^         ^
                      |    |         |
                      C    S---------E


[ 1] [ 2] [ 8] [24] [60] [75] [100] [100] [123]
                                ^    ^
                                |    |
                                C   S/E

----------------------------------------------------------------------
---
New : [ 1] [ 2] [ 8] [24] [60] [75] [100] [123]
```

If there are other elegant methods of removing duplicate numbers from an array, please let me know!.

Finding a duplicated integer problem

Discuss it!

Given an array of n integers from 1 to n with one integer repeated..

Here is the simplest of C programs (kind of dumb)

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

int i,j=0,k,a1[10];

main()
{
     printf("Enter the array of numbers between 1 and 100(you can
repeat the numbers):");
     for(i=0;i<=9;i++)
     {
         scanf("%d",&a1[i]);
     }

     while(j<10)
     {
        for(k=0;k<10;k++)
        {
            if(a1[j]==a1[k] && j!=k)
            {
                printf("Duplicate found!");
                printf("The duplicate is %d\n",a1[j]);
                getch();
            }
        }
        j=j+1;
    }

    getch();
    return(0);
}
```

Write a C program to find the GCD of two numbers.

Discuss it!

```
#include <stdio.h>

int gcd(int a, int b);
int gcd_recurse(int a, int b);


int main()
{
  printf("\nGCD(%2d,%2d) = [%d]", 6,4,  gcd(6,4));
  printf("\nGCD(%2d,%2d) = [%d]", 4,6,  gcd(4,6));
  printf("\nGCD(%2d,%2d) = [%d]", 3,17, gcd(3,17));
  printf("\nGCD(%2d,%2d) = [%d]", 17,3, gcd(17,3));
  printf("\nGCD(%2d,%2d) = [%d]", 1,6,  gcd(1,6));
  printf("\nGCD(%2d,%2d) = [%d]", 10,1, gcd(10,1));
  printf("\nGCD(%2d,%2d) = [%d]", 10,6, gcd(10,6));

  printf("\nGCD(%2d,%2d) = [%d]", 6,4,  gcd_recurse(6,4));
  printf("\nGCD(%2d,%2d) = [%d]", 4,6,  gcd_recurse(4,6));
  printf("\nGCD(%2d,%2d) = [%d]", 3,17, gcd_recurse(3,17));
  printf("\nGCD(%2d,%2d) = [%d]", 17,3, gcd_recurse(17,3));
  printf("\nGCD(%2d,%2d) = [%d]", 1,6,  gcd_recurse(1,6));
  printf("\nGCD(%2d,%2d) = [%d]", 10,1, gcd_recurse(10,1));
  printf("\nGCD(%2d,%2d) = [%d]", 10,6, gcd_recurse(10,6));


  getch();
  getch();
  return(0);
}

// Iterative algorithm
int gcd(int a, int b)
{
   int temp;

   while(b)
   {
      temp = a % b;
      a = b;
      b = temp;
   }

   return(a);
}


// Recursive algorithm
int gcd_recurse(int a, int b)
{
   int temp;

   temp = a % b;

   if (temp == 0)
   {
      return(b);
   }
   else
   {
      return(gcd_recurse(b, temp));
   }
}
```

And here is the output ...

```
Iterative
----------------
GCD( 6, 4) = [2]
GCD( 4, 6) = [2]
GCD( 3,17) = [1]
GCD(17, 3) = [1]
GCD( 1, 6) = [1]
GCD(10, 1) = [1]
GCD(10, 6) = [2]

Recursive
----------------
GCD( 6, 4) = [2]
GCD( 4, 6) = [2]
GCD( 3,17) = [1]
GCD(17, 3) = [1]
GCD( 1, 6) = [1]
GCD(10, 1) = [1]
GCD(10, 6) = [2]
```

Note that you should add error handling to check if someone has passed negative numbers and zero.

Write C code to check if an integer is a power of 2 or not in a single line?

Discuss it!

Even this is one of the most frequently asked interview questions. I really dont know whats so great in it. Nevertheless, here is a C program

Method1

```c
if(!(num & (num - 1)) && num)
{
  // Power of 2!
}
```

Method2

```c
if(((~i+1)&i)==i)
{
```

```
//Power of 2!
}
```

I leave it up to you to find out how these statements work.

Write a C progam to convert from decimal to any base (binary, hex, oct etc...)

Discuss it!

Here is some really cool C code

```
#include <stdio.h>

int main()
{
  decimal_to_anybase(10, 2);
  decimal_to_anybase(255, 16);
  getch();
}

decimal_to_anybase(int n, int base)
{
  int i, m, digits[1000], flag;
  i=0;

  printf("\n\n[%d] converted to base [%d] : ", n, base);

  while(n)
  {
     m=n%base;
     digits[i]="0123456789abcdefghijklmnopqrstuvwxyz"[m];
     n=n/base;
     i++;
   }

   //Eliminate any leading zeroes
   for(i--;i>=0;i--)
   {
     if(!flag && digits[i]!='0')flag=1;
     if(flag)printf("%c",digits[i]);
   }
}
```

Write a C program which produces its own source code as its output

Discuss it!

This is one of the most famous interview questions

One of the famous C programs is...

```
char*s="char*s=%c%s%c;main(){printf(s,34,s,34);}";main(){printf(s,34,s,
34);}
```

So how does it work?

It's not difficult to understand this program. In the following statement,

```
printf(f,34,f,34,10);
```

the parameter "f" not only acts as the format string, but also as a value for the %s specifier. The ASCII value of double quotes is 34, and that of new-line is 10. With these fact ready, the solution is just a matter of tracing the program.

How would you find the size of structure without using sizeof()?

Discuss it!

Try using pointers

```
struct MyStruct
{
  int i;
  int j;
};

int main()
{
   struct MyStruct *p=0;
   int size = ((char*)(p+1))-((char*)p);
   printf("\nSIZE : [%d]\nSIZE : [%d]\n", size);
   return 0;
}
```

Write a C program to multiply two matrices.

Discuss it!

```
Are you sure you know this? A lot of people think they already know
this, but guess what? So take a good look at this C program. Its asked
in most of the interviews as a warm up question.


// Matrix A (m*n)
// Matrix B (n*k)
// Matrix C (m*k)

for(i=0; i<m; i++)
{
   for(j=0;j<k;j++)
   {
      c[i][j]=0;
      for(l=0;l<n;l++)
           c[i][j] += a[i][l] * b[l][j];
   }
}
```

Write a C program to check for palindromes.

Discuss it!

```
An example of a palidrome is "avon sees nova"

There a number of ways in which we can find out if a string is a
palidrome or not. Here are a few sample C programs...


Method1


#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

void isPalindrome(char *string);

int main()
{
  isPalindrome("avon sees nova");
  isPalindrome("a");
  isPalindrome("avon sies nova");
  isPalindrome("aa");
  isPalindrome("abc");
  isPalindrome("aba");
  isPalindrome("3a2");
  exit(0);
}

void isPalindrome(char *string)
{
  char *start, *end;

  if(string)
  {
     start = string;
     end   = string + strlen(string) - 1;

     while((*start == *end) && (start!=end))
     {
       if(start<end)start++;
       if(end>start)end--;
     }

     if(*start!=*end)
     {
        printf("\n[%s] - This is not a palidrome!\n", string);
     }
     else
     {
        printf("\n[%s] - This is a palidrome!\n", string);
     }
  }
  printf("\n\n");
}
```

Method2

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

int isPalindrome(char string[]);

int main()
{
  isPalindrome("avon sees nova");
  isPalindrome("a");
  isPalindrome("avon sies nova");
  isPalindrome("aa");
  isPalindrome("abc");
  isPalindrome("aba");
  isPalindrome("3a2");
  return(0);
}

int isPalindrome(char string[])
{
  int count, countback, end, N;

  N   = strlen(string);
  end = N-1;

  for((count=0, countback = end); count <= (end/2); ++count,--
countback)
  {
    if(string[count]!=string[countback])
    {
      return(1);
    }
  }

  printf("\n[%s] is a palidrome!\n", string);
  return(0);
}
```

Write a C program to convert a decimal number into a binary number.

Discuss it!

```
#include<stdio.h>
void generatebits(int num);

void generatebits(int num)
{
  int temp;
  if(num)
  {
    temp = num % 2;
    generatebits(num >>= 1);
    printf("%d",temp);
  }
}

int main()
{
  int num;
  printf("\nEnter a number\n");
  scanf("%d", &num);
  printf("\n\n");
  generatebits(num);
  getch();
  return(0);
}
```

The reason we have shown a recursive algorithm is that, because of the magic of recursion, we dont have to reverse the bits generated to produce the final output. One can always write an iterative algorithm to accomplish the same, but it would require you to first store the bits as they are generated and then reverse them before producing the final output.

Write C code to implement the Binary Search algorithm.

Discuss it!

Here is a C function

```
int binarySearch(int arr[],int size, int item)
{
   int left, right, middle;
   left  = 0;
   right = size-1;

   while(left<=right)
   {
      middle = ((left + right)/2);

      if(item == arr[middle])
      {
        return(middle);
      }

      if(item > arr[middle])
      {
        left  = middle+1;
      }
      else
      {
        right = middle-1;
      }
   }

   return(-1);
}
```

Note that the Binary Search algorithm has a prerequisite that the array passed to it must be already sorted in ascending order. This will not work on an unsorted array. The complexity of this algorithm is O(log(n)).

Wite code to evaluate a polynomial.

Discuss it!

```
typedef struct node
{
  float cf;
  float px;
  float py;
  struct node *next;
}mynode;


float evaluate(mynode *head)
{
   float x,y,sum;
   sum = 0;
   mynode *poly;

   for(poly = head->next; poly != head; poly = poly->next)
   {
     sum = sum + poly->cf * pow(x, poly->px) * pow(y, poly->py);
   }

   return(sum);
}
```

Write code to add two polynomials

Discuss it!

```
Here is some pseudocode


mynode *polynomial_add(mynode *h1, mynode *h2, mynode *h3)
{
   mynode *p1, *p2;
   int x1, x2, y1, y2, cf1, cf2, cf;

   p1 = h1->next;

   while(p1!=h1)
   {
      x1  = p1->px;
      y1  = p1->py;
      cf1 = p1->cf;

      // Search for this term in the second polynomial

      p2 = h2->next;

      while(p2 != h2)
      {
         x2  = p2->px;
         y2  = p2->py;
         cf2 = p2->cf;

         if(x1 == x2 && y1 == y2)break;

         p2 = p2->next;

      }
      if(p2 != h2)
      {
          // We found something in the second polynomial.

          cf = cf1 + cf2;
          p2->flag = 1;

          if(cf!=0){h3=addNode(cf,x1,y1,h3);}
      }
      else
      {
         h3=addNode(cf,x1,y1,h3);
      }

      p1 = p1->next;

   }//while


   // Add the remaining elements of the second polynomail to the result

   while(p2 != h2)
   {
      if(p2->flag==0)
      {
          h3=addNode(p2->cf, p2->px, p2->py, h3);
      }
      p2=p2->next;
   }

   return(h3);
}
```

Write a program to add two long positive numbers (each represented by linked lists).

Discuss it!

```
Check out this simple implementation


mynode *long_add(mynode *h1, mynode *h2, mynode *h3)
{
  mynode *c, *c1, *c2;
  int sum, carry, digit;

  carry = 0;
  c1 = h1->next;
  c2 = h2->next;
  while(c1 != h1 && c2 != h2)
  {
     sum   = c1->value + c2->value + carry;
     digit = sum % 10;
     carry = sum / 10;

     h3 = insertNode(digit, h3);

     c1 = c1->next;
     c2 = c2->next;
  }

  if(c1 != h1)
  {
     c = c1;
     h = h1;
  }
  else
  {
     c = c2;
     h = h2;
  }

  while(c != h)
  {
    sum   = c->value + carry;
    digit = sum % 10;
    carry = sum / 10;
    h3 = insertNode(digit, h3);
    c = c->next;
  }

  if(carry==1)
  {
     h3 = insertNode(carry, h3);
  }

  return(h3);
}
```

How do you compare floating point numbers?

Discuss it!

```
This is Wrong!.


double a, b;

if(a == b)
```

```
{
  ...
}
```

The above code might not work always. Thats because of the way floating point numbers are stored.

A good way of comparing two floating point numbers is to have a accuracy threshold which is relative to the magnitude of the two floating point numbers being compared.

```
#include <math.h>
if(fabs(a - b) <= accurary_threshold * fabs(a))
```

There is a lot of material on the net to know how floating point numbers can be compared. Got for it if you really want to understand.

Another way which might work is something like this. I have not tested it!

```
int compareFloats(float f1, float f2)
{
  char *b1, *b2;
  int i;

  b1 = (char *)&f1;
  b2 = (char *)&f2;

  /* Assuming sizeof(float) is 4 bytes) */

  for (i = 0; i<4; i++, b1++, b2++)
  {
    if (*b1 != *b2)
    {
      return(NOT_EQUAL); /* You must have defined this before */
    }
  }

  return(EQUAL);
}
```

How can I display a percentage-done indication on the screen?

Discuss it!

The character '\r' is a carriage return without the usual line feed, this helps to overwrite the current line. The character '\b' acts as a backspace, and will move the cursor one position to the left.

Write a program to check if a given year is a leap year or not?

Discuss it!

Use this if() condition to check if a year is a leap year or not

```
if(year % 4 == 0 && (year % 100 != 0 || year % 400 == 0))
```

Is there something we can do in C but not in C++?

Discuss it!

Declare variable names that are keywords in C++ but not C.

```
#include <stdio.h>
int main(void)
{
  int old, new=3;
  return 0;
}
```

This will compile in C, but not in C++!

How to swap the two nibbles in a byte ?

Discuss it!

```
#include <stdio.h>

unsigned char swap_nibbles(unsigned char c)
{
  unsigned char temp1, temp2;
  temp1 = c & 0x0F;
  temp2 = c & 0xF0;
  temp1=temp1 << 4;
  temp2=temp2 >> 4;

  return(temp2|temp1); //adding the bits
}

int main(void)
{
  char ch=0x34;
  printf("\nThe exchanged value is %x",swap_nibbles(ch));
  return 0;
}
```

How to scan a string till we hit a new line using scanf()?

Discuss it!

```
scanf("%[^\n]", address);
```

Write pseudocode to compare versions (like 115.10.1 vs 115.11.5).

Discuss it!

This question is also quite popular, because it has real practical uses, specially during patching when version comparison is required

The pseudocode is something like

```
while(both version strings are not NULL)
{
  // Extract the next version segment from Version string1.
  // Extract the next version segment from Version string2.

  // Compare these segments, if they are equal, continue
  // to check the next version segments.

  // If they are not equal, return appropriate code depending
  // on which segment is greater.
}
```

And here is some code in PL-SQL

```
----------------------------------------------------------------------
compare_versions()
```

Function to compare releases. Very useful when comparing file versions!

This function compare two versions in pl-sql language. This function

```
can compare
Versions like 115.10.1 vs. 115.10.2 (and say 115.10.2 is greater),
115.10.1 vs. 115.10 (and say
115.10.1 is greater), 115.10 vs. 115.10 (and say both are equal)
----------------------------------------------------------------------

function compare_releases(release_1 in varchar2, release_2 in varchar2)
return boolean is

  release_1_str varchar2(132);
  release_2_str varchar2(132);
  release_1_ver number;
  release_2_ver number;
  ret_status boolean := TRUE;

begin

  release_1_str := release_1 || '.';
  release_2_str := release_2 || '.';

  while release_1_str is not null or release_2_str is not null loop

    -- Parse out a current version segment from release_1

    if (release_1_str is null) then
       release_1_ver := 0;
    else
      release_1_ver := nvl(to_number(substr(release_1_str,1,
instr(release_1_str,'.')-1)),-1);
      release_1_str :=
substr(release_1_str,instr(release_1_str,'.')+1);
    end if;

    -- Next parse out a version segment from release_2
    if (release_2_str is null) then
       release_2_ver := 0;
    else
       release_2_ver := nvl(to_number(substr(release_2_str,1,
instr(release_2_str,'.')-1)),-1);
       release_2_str :=
substr(release_2_str,instr(release_2_str,'.')+1);
    end if;


     if (release_1_ver > release_2_ver) then
        ret_status := FALSE;
        exit;
     elsif (release_1_ver < release_2_ver) then
        exit;
     end if;

     -- Otherwise continue to loop.
   end loop;

   return(ret_status);
```

```
end compare_releases;
```

How do you get the line numbers in C?

[Discuss it!]

Use the following Macros

```
__FILE__ Source file name (string constant) format "patx.c"
__LINE__ Current source line number (integer)
__DATE__ Date compiled (string constant)format "Dec 14 1985"
__TIME__ Time compiled (string constant) format "15:24:26"
__TIMESTAMP__ Compile date/time (string constant)format "Tue Nov 19
11:39:12 1997"
```

Usage example

```
static char stamp[] =    "***\nmodule " __FILE__    "\ncompiled "
__TIMESTAMP__     "\n***";

...

int main()
{
   ...

   if ( (fp = fopen(fl,"r")) == NULL )
   {
      printf( "open failed, line %d\n%s\n",__LINE__, stamp );
      exit( 4 );
   }

   ...
}
```

And the output is something like

```
*** open failed, line 67
******
module myfile.c
compiled Mon Jan 15 11:15:56 1999
***
```

How to fast multiply a number by 7?

Discuss it!

Try

```
(num<<3 - num)
```

This is same as

```
num*8 - num = num * (8-1) = num * 7
```

Write a simple piece of code to split a string at equal intervals

Discuss it!

Suppose you have a big string

This is a big string which I want to split at equal intervals, without caring about the words.

Now, to split this string say into smaller strings of 20 characters each, try this

```
#define maxLineSize 20

split(char *string)
{
  int i, length;
  char dest[maxLineSize + 1];

  i           = 0;
  length      = strlen(string);

  while((i+maxLineSize) <= length)
  {
     strncpy(dest, (string+i), maxLineSize);
     dest[maxLineSize - 1] = '\0';
     i = i + strlen(dest) - 1;
     printf("\nChunk : [%s]\n", dest);
  }

  strcpy(dest, (string + i));
  printf("\nChunk : [%s]\n", dest);
}
```

Is there a way to multiply matrices in lesser than o(n^3) time complexity?

Discuss it!

Yes. Divide and conquer method suggests Strassen's matrix multiplication method to be used. If we follow this method, the time complexity is O(n^2.81) times rather O(n^3) times.

Here are some more details about this method.

Suppose we want to multiply two matrices of size N x N: for example A x B = C

```
[C11 C12]   [A11 A12] [B11 B12]
[C21 C22] = [A21 A22] [B21 B22]
```

Now, this guy called Strassen's somehow :) came up with a bunch of equations to calculate the 4 elements of the resultant matrix

```
C11 = a11*b11 + a12*b21
C12 = a11*b12 + a12*b22
C21 = a21*b11 + a22*b21
C22 = a21*b12 + a22*b22
```

If you are aware, the rudimentary matrix multiplication goes something like this

```
void matrix_mult()
{
  for (i = 1; i <= N; i++)
  {
     for (j = 1; j <= N; j++)
     {
        compute
Ci,j;
     }
  }
}
```

So, essentially, a 2x2 matrix multiplication can be accomplished using 8 multiplications. And the complexity becomes

2^log 8 =2^3

Strassen showed that 2x2 matrix multiplication can be accomplished in 7 multiplications and 18 additions or subtractions. So now the complexity becomes

2^log7 =2^2.807

This is how he did it

```
P1 = (A11+ A22)(B11+B22)
P2 = (A21 + A22) * B11
P3 = A11 * (B12 - B22)
P4 = A22 * (B21 - B11)
P5 = (A11 + A12) * B22
P6 = (A21 - A11) * (B11 + B12)
P7 = (A12 - A22) * (B21 + B22)

C11 = P1 + P4 - P5 + P7
C12 = P3 + P5
C21 = P2 + P4
C22 = P1 + P3 - P2 + P6
```

Now, there is no need to memorize this stuff!

How do you find out if a machine is 32 bit or 64 bit?

Discuss it!

One common answer is that all compilers keep the size of integer the same as the size of the register on a perticular architecture. Thus, to know whether the machine is 32 bit or 64 bit, just see the size of integer on it.

I am not sure how true this is

Write a program to have the output go two places at once (to the screen and to a file also)

Discuss it!

You can write a wrapper function for printf() which prints twice.

```
myprintf(...)
{
  // printf();         -> To screen.
  // write_to_file(); -> To file.
}
```

```
A command in shell, called tee does have this functionality!
```

Write code to round numbers

Discuss it!

```
Use something like


(int)(num < 0 ? (num - 0.5) : (num + 0.5))
```

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

Discuss it!

```
Try something like this


# include<stdio.h>

void main()
{
  int num=123456;
  int sum=0;

  for(;num>0;sum+=num%10,num/=10);  // This is the "single line".

  printf("\nsum = [%d]\n", sum);
}


If there is a simpler way to do this, let me know!
```

Given two strings A and B, how would you find out if the characters in B were a subset of the characters in A?

Discuss it!

```
Here is a simple, yet efficient C program to accomplish the same...


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

int isSubset(char *a, char *b);

int main()
{
char str1[]="defabc";
char str2[]="abcfed";

if(isSubset(str1, str2)==0)
{
   printf("\nYes, characters in B=[%s] are a subset of characters in
A=[%s]\n",str2,str1);
}
else
{
   printf("\nNo, characters in B=[%s] are not a subset of characters in
A=[%s]\n",str2,str1);
}

getch();
return(0);
}


// Function to check if characters in "b" are a subset
// of the characters in "a"

int isSubset(char *a, char *b)
{
int letterPresent[256];
int i;

for(i=0; i<256; i++)
    letterPresent[i]=0;

for(i=0; a[i]!='\0'; i++)
    letterPresent[a[i]]++;

for(i=0; b[i]!='\0'; i++)
    if(!letterPresent[b[i]])
       return(1);

return(0);
}
```

Write a program to merge two arrays in sorted order, so that if an integer is in both the arrays, it gets added into the final array only once. *

Write a program to check if the stack grows up or down

Discuss it!

```
Try noting down the address of a local variable. Call another function
with a local variable declared in it and check the address of that
local variable and compare!.


#include <stdio.h>
```

```
#include <stdlib.h>

void stack(int *local1);

int main()
{
  int local1;
  stack(&local1);
  exit(0);
}

void stack(int *local1)
{
   int local2;
   printf("\nAddress of first  local : [%u]", local1);
   printf("\nAddress of second local : [%u]", &local2);
   if(local1 < &local2)
   {
     printf("\nStack is growing downwards.\n");
   }
   else
   {
     printf("\nStack is growing upwards.\n");
   }
   printf("\n\n");
}
```

How to add two numbers without using the plus operator?

Discuss it!

```
Actually,


SUM   = A XOR B
CARRY = A AND B


On a wicked note, you can add two numbers wihtout using the + operator
as follows


a - (- b)
```

How to generate prime numbers? How to generate the next prime after a given prime?

Discuss it!

```
This is a very vast subject. There are numerous methods to generate
```

primes or to find out if a given number is a prime number or not. Here are a few of them. I strongly recommend you to search on the Internet for more elaborate information.

Brute Force
Test each number starting with 2 and continuing up to the number of primes we want to generate. We divide each numbr by all divisors upto the square root of that number. If no factors are found, its a prime.

Using only primes as divisors
Test each candidate only with numbers that have been already proven to be prime. To do so, keep a list of already found primes (probably using an array, a file or bit fields).

Test with odd candidates only
We need not test even candidates at all. We could make 2 a special case and just print it, not include it in the list of primes and start our candidate search with 3 and increment by 2 instead of one everytime.

Table method
Suppose we want to find all the primes between 1 and 64. We write out a table of these numbers, and proceed as follows. 2 is the first integer greater than 1, so its obviously prime. We now cross out all multiples of two. The next number we haven't crossed out is 3. We circle it and cross out all its multiples. The next non-crossed number is 5, sp we circle it and cross all its mutiples. We only have to do this for all numbers less than the square root of our upper limit, since any composite in the table must have atleast one factor less than the square root of the upper limit. Whats left after this process of elimination is all the prime numbers between 1 and 64.

Write a program to print numbers from 1 to 100 without using loops!

Discuss it!

Another "Yeah, I am a jerk, kick me! kind of a question. I simply dont know why they ask these questions.

Nevertheless, for the benefit of mankind...

Method1 (Using recursion)

```
void printUp(int startNumber, int endNumber)
{
  if (startNumber > endNumber)
    return;
  printf("[%d]\n", startNumber++);
  printUp(startNumber, endNumber);
}
```

Method2 (Using goto)

```
void printUp(int startNumber, int endNumber)
{
start:

    if (startNumber > endNumber)
    {
       goto end;
    }
    else
    {
       printf("[%d]\n", startNumber++);
       goto start;
    }

end:
    return;
}
```

Write your own trim() or squeeze() function to remove the spaces from a string.

Discuss it!

Here is one version...

```
#include <stdio.h>

char *trim(char *s);

int main(int argc, char *argv[])
{
  char str1[]=" Hello   I am Good ";
  printf("\n\nBefore trimming : [%s]", str1);
  printf("\n\nAfter trimming  : [%s]", trim(str1));

  getch();
}
```

```
// The trim() function...
```

```
char *trim(char *s)
{
    char *p, *ps;

    for (ps = p = s; *s != '\0'; s++)
    {
        if (!isspace(*s))
        {
            *p++ = *s;
        }
    }

    *p = '\0';

    return(ps);
}
```

And here is the output...

```
Before trimming : [ Hello   I am Good ]
After trimming  : [HelloIamGood]
```

Another version of this question requires one to reduce multiple spaces, tabs etc to single spaces...

Write your own random number generator function in C.*
Write your own sqrt() function in C

Write a C program to find the depth or height of a tree.

Discuss it!

```
#define max(x,y) ((x)>(y)?(x):(y))

struct Bintree{
    int element;
    struct Bintree *left;
    struct Bintree *right;
};

typedef struct Bintree* Tree;

int height(Tree T)
{
    if(!T)
```

```
        return -1;
    else
        return (1 + max(height(T->left), height(T->right)))
}
```

Write a C program to determine the number of elements (or size) in a tree.

Discuss it!

```
struct Bintree{
    int element;
    struct Bintree *left;
    struct Bintree *right;
};

typedef struct Bintree* Tree;

int CountElements( Tree T )
{
    if(!T)
        return 0;
    else
        return (1 + CountElements(T->left) + CountElements(T->right));
}
```

Write a C program to delete a tree (i.e, free up its nodes)

Discuss it!

```
struct Bintree{
    int element;
    struct Bintree *left;
    struct Bintree *right;
};

typedef struct Bintree* Tree;

Tree findMin( Tree T) // Recursively finding min element
{
    if( !T )
        return NULL;
    else if( !T->left )
        return T;
    else
        return findMin( T->left );
}

Tree DeleteTree( int x, Tree T) // To Delete whole tree recursively
{
    if(!T)
    {
        DeleteTree(T->left);
        DeleteTree(T->right);
        free(T);
    }
    return NULL;
}

Tree DeleteNode( int x, Tree T ) // To Delete a Node with element x
{
    Tree tmp;
    if(!T)
        return NULL;
    else if( x < T->element )
        T->left = DeleteNode( x, T->left );
    else if( x > T->element )
        T->right = DeleteNode( x, T->right );
    else if( T->left && T->right )
    {
        tmp = findMin( T-> right );
        T->element = tmp->element;
        T->right = DeleteNode( T->element, T->right );
    }
    else
    {
        tmp = T;
        if( T->left )
            T = T->left;
        else if( T->right )
            T = T->right;
        free(tmp);
    }
}
```

Write C code to determine if two trees are identical

Discuss it!

```
struct Bintree{
    int element;
    struct Bintree *left;
    struct Bintree *right;
};
```

```
typedef struct Bintree* Tree;

int CheckIdentical( Tree T1, Tree T2 )
{
    if(!T1 && !T2) // If both tree are NULL then return true
        return 1;
    else if((!T1 && T2) || (T1 && !T2)) //If either of one is NULL,
return false
        return 0;
    else
        return ((T1->element == T2->element) && CheckIdentical(T1-
>left, T2-i>left) && CheckIdentical(T1->right, T2->right));
        // if element of both tree are same and left and right tree is
also same then both trees are same
}
```

Write a C program to find the mininum value in a binary search tree.

Discuss it!

```
#define NOT_FOUND -999

struct Bintree{
    int element;
    struct Bintree *left;
    struct Bintree *right;
};

typedef struct Bintree* Tree;

int findMinElement( Tree T) // Recursively finding min element
{
    if( !T )
        return NOT_FOUND;
    else if( !T->left )
        return T->element;
    else
        return findMin( T->left );
}
```

Write a C program to compute the maximum depth in a tree?

Discuss it!

```
int maxDepth(struct node* node)
{
  if (node==NULL)
  {
    return(0);
  }
  else
  {
    int leftDepth  = maxDepth(node->left);
    int rightDepth = maxDepth(node->right);
    if (leftDepth > rightDepth) return(leftDepth+1);
    else return(rightDepth+1);
  }
}
```

Write a C program to create a mirror copy of a tree (left nodes become right and right nodes become left)!

Discuss it!

```
This C code will create a new mirror copy tree.


mynode *copy(mynode *root)
{
  mynode *temp;

  if(root==NULL)return(NULL);

  temp = (mynode *) malloc(sizeof(mynode));
  temp->value = root->value;

  temp->left  = copy(root->right);
  temp->right = copy(root->left);

  return(temp);
}




This code will will only print the mirror of the tree



void tree_mirror(struct node* node)
{
   struct node *temp;

   if (node==NULL)
   {
     return;
   }
   else
   {
      tree_mirror(node->left);
      tree_mirror(node->right);

      // Swap the pointers in this node
      temp = node->left;
      node->left = node->right;
      node->right = temp;
   }
}
```

Write C code to return a pointer to the nth node of an inorder traversal of a BST.

Discuss it!

```
typedef struct node
{
  int value;
  struct node *left;
  struct node *right;
}mynode;

mynode *root;
static ctr;

void nthnode(mynode *root, int n, mynode **nthnode /* POINTER TO A
POINTER! */);

int main()
{
  mynode *temp;
  root = NULL;

  // Construct the tree
  add(19);
  add(20);
  ...
  add(11);

  // Plain Old Inorder traversal
  // Just to see if the next function is really returning the nth node?
  inorder(root);

  // Get the pointer to the nth Inorder node
  nthinorder(root, 6, &temp);

  printf("\n[%d]\n, temp->value);
  return(0);
}

// Get the pointer to the nth inorder node in "nthnode"
void nthinorder(mynode *root, int n, mynode **nthnode)
{
  static whichnode;
  static found;
  if(!found)
  {
    if(root)
    {
       nthinorder(root->left, n , nthnode);
       if(++whichnode == n)
       {
          printf("\nFound %dth node\n", n);
          found = 1;
          *nthnode = root; // Store the pointer to the nth node.
        }
       nthinorder(root->right, n , nthnode);
    }
  }
}


inorder(mynode *root)
{
  // Plain old inorder traversal
}


// Function to add a new value to a Binary Search Tree
add(int value)
{
  mynode *temp, *prev, *cur;

  temp = malloc(sizeof(mynode));
  temp->value = value;
  temp->left  = NULL;
  temp->right = NULL;

  if(root == NULL)
  {
    root = temp;
  }
  else
  {
    prev = NULL;
    cur  = root;

    while(cur)
    {
      prev = cur;
      cur  = (value < cur->value)? cur->left : cur->right;
    }

    if(value > prev->value)
       prev->right = temp;
    else
       prev->left  = temp;
  }
}
```

There seems to be an easier way to do this, or so they say. Suppose each node also has a weight associated with it. This weight is the number of nodes below it and including itself. So, the root will have the highest weight (weight of its left subtree + weight of its right subtree + 1). Using this data, we can easily find the nth inorder node.

Note that for any node, the (weight of the leftsubtree of a node + 1) is its inorder rankin the tree!. Thats simply because of how the inorder traversal works (left->root->right). So calculate the rank of each node and you can get to the nth inorder node easily. But frankly speaking, I really dont know how this method is any simpler than the one I have presented above. I see more work to be done here (calculate thw weights, then calculate the ranks and then get to the nth node!).

Also, if (n > weight(root)), we can error out saying that this tree does not have the nth node you are looking for.

Write C code to implement the preorder(), inorder() and postorder() traversals. Whats their time complexities?

Discuss it!

Preorder

```
preorder(mynode *root)
{
  if(root)
  {
    printf("Value : [%d]", root->value);
    preorder(root->left);
    preorder(root->right);
  }
}
```

Postorder

```
postorder(mynode *root)
{
  if(root)
  {
    postorder(root->left);
    postorder(root->right);
    printf("Value : [%d]", root->value);
  }
}
```

```
Inorder

inorder(mynode *root)
{
  if(root)
  {
    inorder(root->left);
    printf("Value : [%d]", root->value);
    inorder(root->right);
  }
}



Time complexity of traversals is O(n).
```

Write a C program to create a copy of a tree

Discuss it!

```
mynode *copy(mynode *root)
{
  mynode *temp;

  if(root==NULL)return(NULL);

  temp = (mynode *) malloc(sizeof(mynode));
  temp->value = root->value;

  temp->left  = copy(root->left);
  temp->right = copy(root->right);

  return(temp);
}
```

Write C code to check if a given binary tree is a binary search tree or not?

Discuss it!

```
int isThisABST(struct node* mynode)
{
    if (mynode==NULL) return(true);
    if (node->left!=NULL && maxValue(mynode->left) > mynode->data)
       return(false);
    if (node->right!=NULL && minValue(mynode->right) <= mynode->data)
       return(false);
    if (!isThisABST(node->left) || !isThisABST(node->right))
```

```
       return(false);

    return(true);
}
```

Write C code to implement level order traversal of a tree.

Discuss it!

```
If this is the tree,


        1
   2         3
5    6     7    8


its level order traversal would be


1 2 3 5 6 7 8



You need to use a queue to do this kind of a traversal


Let t be the tree root.
while (t != null)
{
  visit t and put its children on a FIFO queue;
  remove a node from the FIFO queue and
  call it t;
  // Remove returns null when queue is empty
}



Pseduocode


Level_order_traversal(p)
{
   while(p)
   {
     Visit(p);
     If(p->left)Q.Add(p->left);
     If(p->right)Q.Add(p->right);
     Delete(p);
   }
}
```

Here is some C code (working :))..

```
#include <stdio.h>

typedef struct node
{
  int value;
  struct node *right;
  struct node *left;
}mynode;

mynode *root;

add_node(int value);

void levelOrderTraversal(mynode *root);

int main(int argc, char* argv[])
{
  root = NULL;

  add_node(5);
  add_node(1);
  add_node(-20);
  add_node(100);
  add_node(23);
  add_node(67);
  add_node(13);


  printf("\n\n\nLEVEL ORDER TRAVERSAL\n\n");
  levelOrderTraversal(root);

  getch();
}




// Function to add a new node...
add_node(int value)
{
   mynode *prev, *cur, *temp;

   temp        = (mynode *) malloc(sizeof(mynode));
   temp->value = value;
   temp->right = NULL;
   temp->left  = NULL;

   if(root==NULL)
   {
     printf("\nCreating the root..\n");
     root = temp;
```

```
      return;
    }

    prev=NULL;
    cur=root;

    while(cur!=NULL)
    {
       prev=cur;
       cur=(value<cur->value)?cur->left:cur->right;
    }

    if(value < prev->value)
      prev->left=temp;
    else
      prev->right=temp;
}



// Level order traversal..
void levelOrderTraversal(mynode *root)
{
  mynode *queue[100] = {(mynode *)0}; // Important to initialize!
  int size = 0;
  int queue_pointer = 0;

  while(root)
  {
      printf("[%d] ", root->value);

      if(root->left)
      {
        queue[size++] = root->left;
      }

      if(root->right)
      {
        queue[size++] = root->right;
      }

      root = queue[queue_pointer++];
  }
}
```

Write a C program to delete a node from a Binary Search Tree?

Discuss it!

The node to be deleted might be in the following states

The node does not exist in the tree - In this case you have nothing to delete.

The node to be deleted has no children - The memory occupied by this node must be freed and either the left link or the right link of the parent of this node must be set to NULL.

The node to be deleted has exactly one child - We have to adjust the pointer of the parent of the node to be deleted such that after deletion it points to the child of the node being deleted.

The node to be deleted has two children - We need to find the inorder successor of the node to be deleted. The data of the inorder successor must be copied into the node to be deleted and a pointer should be setup to the inorder successor. This inorder successor would have one or zero children. This node should be deleted using the same procedure as for deleting a one child or a zero child node. Thus the whole logic of deleting a node with two children is to locate the inorder successor, copy its data and reduce the problem to a simple deletion of a node with one or zero children.

Here is some C code for these two situations

Situation 1

```
     100 (parent)

   50 (cur == psuc)

20    80 (suc)

         90

       85  95
```

Situation 2

```
           100 (parent)

        50 (cur)

     20    90

         80

       70 (suc)

          75

        72   76
```

```
mynode *delete(int item, mynode *head)
{
     mynode *cur, *parent, *suc, *psuc, q;

     if(head->left==NULL){printf("\nEmpty tree!\n");return(head);}

     parent = head;
     cur    = head->left;

     while(cur!=NULL && item != cur->value)
     {
         parent  =  cur;
         cur     =  (item < cur->next)? cur->left:cur->right;
     }

     if(cur == NULL)
     {
        printf("\nItem to be deleted not found!\n");
        return(head);
     }


     // Item found, now delete it

     if(cur->left == NULL)
         q = cur->right;
     else if(cur->right == NULL)
         q = cur->left;
     else
     {
           // Obtain the inorder successor and its parent

           psuc = cur;
           cur  = cur->left;

           while(suc->left!=NULL)
           {
               psuc = suc;
               suc  = suc->left;
           }


           if(cur==psuc)
           {
                // Situation 1

                suc->left = cur->right;
           }
           else
           {
                // Situation 2

                suc->left  = cur->left;
                psuc->left = suc->right;
                suc->right = cur->right;
           }

           q = suc;

     }


     // Attach q to the parent node

     if(parent->left == cur)
        parent->left=q;
     else
        parent->rlink=q;

     freeNode(cur);

     return(head);
}
```

Write C code to search for a value in a binary search tree (BST).

Discuss it!

```
mynode *search(int value, mynode *root)
{
   while(root!=NULL && value!=root->value)
   {
     root = (value < root->value)?root->left:root->right;
   }

   return(root);
}



Here is another way to do the same


mynode *recursive_search(int item, mynode *root)
{
  if(root==NULL || item == root->value){return(root);}
  if(item<root->info)return{recursive_search(item, root->left);}
  return{recursive_search(item, root->right);}
}
```

Write C code to count the number of leaves in a tree

Discuss it!

```
void count_leaf(mynode *root)
{
   if(root!=NULL)
   {
      count_leaf(root->left);

      if(root->left == NULL && root->right==NULL)
      {
        // This is a leaf!
        count++;
      }

      count_leaf(root->right);
   }
}
```

Write C code for iterative preorder, inorder and postorder tree traversals

Discuss it!

Here is a complete C program which prints a BST using both recursion and iteration. The best way to understand these algorithms is to get a pen and a paper and trace out the traversals (with the stack or the queue) alongside. Dont even try to memorize these algorithms!

```
#include <stdio.h>

typedef struct node
{
  int value;
  struct node *right;
  struct node *left;
}mynode;

mynode *root;

add_node(int value);
void postorder(mynode *root);
void inorder(mynode *root);
void preorder(mynode *root);

void iterativePreorder(mynode *root);
void iterativeInorder (mynode *root);
void iterativePostorder(mynode *root);


int main(int argc, char* argv[])
{
  root = NULL;

  add_node(5);
  add_node(1);
  add_node(-20);
  add_node(100);
  add_node(23);
  add_node(67);
  add_node(13);

  printf("\nPreorder (R)    : ");
  preorder(root);
  printf("\nPreorder (I)    : ");
  iterativePreorder(root);

  printf("\n\nPostorder (R)   : ");
  postorder(root);
  printf("\nPostorder (R)   : ");
  iterativePostorder(root);


  printf("\n\nInorder (R)     : ");
  inorder(root);
  printf("\nInorder (I)     : ");
  iterativeInorder(root);

}

// Function to add a new node to the BST
add_node(int value)
{
   mynode *prev, *cur, *temp;

   temp        = (mynode *) malloc(sizeof(mynode));
   temp->value = value;
   temp->right = NULL;
   temp->left  = NULL;

   if(root==NULL)
   {
     printf("\nCreating the root..\n");
     root = temp;
     return;
   }

   prev=NULL;
   cur=root;

   while(cur!=NULL)
   {
      prev=cur;
      cur=(value<cur->value)?cur->left:cur->right;
   }

   if(value < prev->value)
     prev->left=temp;
   else
     prev->right=temp;
}
```

```
// Recursive Preorder
void preorder(mynode *root)
{
  if(root)
  {
    printf("[%d] ", root->value);
    preorder(root->left);
    preorder(root->right);
  }
}

// Iterative Preorder
void iterativePreorder(mynode *root)
{
  mynode *save[100];
  int top = 0;

  if (root == NULL)
  {
    return;
  }

  save[top++] = root;
  while (top != 0)
  {
    root = save[--top];

    printf("[%d] ", root->value);

    if (root->right != NULL)
      save[top++] = root->right;
    if (root->left != NULL)
      save[top++] = root->left;
  }
}

// Recursive Postorder
void postorder(mynode *root)
{
  if(root)
  {
    postorder(root->left);
    postorder(root->right);
    printf("[%d] ", root->value);
  }
}

// Iterative Postorder
void iterativePostorder(mynode *root)
{
  struct
  {
    mynode *node;
    unsigned vleft :1;   // Visited left?
    unsigned vright :1;  // Visited right?
  }save[100];

  int top = 0;

  save[top++].node = root;

  while ( top != 0 )
  {
      /* Move to the left subtree if present and not visited */
      if(root->left != NULL && !save[top].vleft)
      {
          save[top].vleft = 1;
          save[top++].node = root;
          root = root->left;
          continue;
      }

      /* Move to the right subtree if present and not visited */
      if(root->right != NULL && !save[top].vright )
      {
          save[top].vright = 1;
          save[top++].node = root;
          root = root->right;
          continue;
      }

      printf("[%d] ", root->value);

      /* Clean up the stack */
      save[top].vleft = 0;
      save[top].vright = 0;

      /* Move up */
      root = save[--top].node;
   }
}


// Recursive Inorder
void inorder(mynode *root)
{
  if(root)
  {
    inorder(root->left);
    printf("[%d] ", root->value);
    inorder(root->right);
  }
}


// Iterative Inorder..
void iterativeInorder (mynode *root)
{
  mynode *save[100];
  int top = 0;
  while(root != NULL)
  {
      while (root != NULL)
      {
           if (root->right != NULL)
           {
             save[top++] = root->right;
           }
           save[top++] = root;
           root = root->left;
      }

      root = save[--top];
      while(top != 0 && root->right == NULL)
      {
          printf("[%d] ", root->value);
          root = save[--top];
      }

      printf("[%d] ", root->value);
      root = (top != 0) ? save[--top] : (mynode *) NULL;
  }
}
```

And here is the output...

```
Creating the root..

Preorder (R)    : [5] [1] [-20] [100] [23] [13] [67]
Preorder (I)    : [5] [1] [-20] [100] [23] [13] [67]

Postorder (R)   : [-20] [1] [13] [67] [23] [100] [5]
Postorder (R)   : [-20] [1] [13] [67] [23] [100] [5]

Inorder (R)     : [-20] [1] [5] [13] [23] [67] [100]
Inorder (I)     : [-20] [1] [5] [13] [23] [67] [100]
```

Can you construct a tree using postorder and preorder traversal?

Discuss it!

No

Consider 2 trees below

```
Tree1

  a
b


Tree 2

a
  b


preorder  = ab
postorder = ba
```

Preorder and postorder do not uniquely define a binary tree. Nor do preorder and level order (same example). Nor do postorder and level order.

Construct a tree given its inorder and preorder traversal strings. Similarly construct a tree given its inorder and post order traversal strings.

Discuss it!

For Inorder And Preorder traversals

```
inorder  = g d h b e i a f j c
preorder = a b d g h e i c f j
```

Scan the preorder left to right using the inorder sequence to separate left and right subtrees. For example, "a" is the root of the tree; "gdhbei" are in the left subtree; "fjc" are in the right subtree. "b" is the next root; "gdh" are in the left subtree; "ei" are in the right subtree. "d" is the next root; "g" is in the left subtree; "h" is in the right subtree.

For Inorder and Postorder traversals

Scan postorder from right to left using inorder to separate left and right subtrees.

```
inorder   = g d h b e i a f j c
postorder = g h d i e b j f c a
```

Tree root is "a"; "gdhbei" are in left subtree; "fjc" are in right subtree.

For Inorder and Levelorder traversals

Scan level order from left to right using inorder to separate left and right subtrees.

```
inorder     = g d h b e i a f j c
level order = a b c d e f g h i j
```

Tree root is "a"; "gdhbei" are in left subtree; "fjc" are in right subtree.

Here is some working code which creates a tree out of the Inorder and Postorder
traversals. Note that here the tree has been represented as an array.
This really simplifies the whole implementation.

Converting a tree to an array is very easy

Suppose we have a tree like this

```
      A
  B       C
D E     F G
```

The array representation would be

```
a[1] a[2] a[3] a[4] a[5] a[6] a[7]
  A    B    C    D    E    F    G
```

That is, for every node at position j in the array, its left child will be stored at position (2*j) and right child at (2*j + 1). The root starts at position 1.

```
// CONSTRUCTING A TREE GIVEN THE INORDER AND PREORDER SEQUENCE

#include<stdio.h>
#include<string.h>
#include<ctype.h>

/*---------------------------------------------------------------
* Algorithm
*
* Inorder And Preorder
* inorder = g d h b e i a f j c
* preorder = a b d g h e i c f j
* Scan the preorder left to right using the inorder to separate left
* and right subtrees. a is the root of the tree; gdhbei are in the
* left subtree; fjc are in the right subtree.
*-------------------------------------------------------------*/

static char io[]="gdhbeiafjc";
static char po[]="abdgheicfj";
static char t[100][100]={'\0'};  //This is where the final tree will be
stored
static int hpos=0;

void copy_str(char dest[], char src[], int pos, int start, int end);
void print_t();

int main(int argc, char* argv[])
{
  int i,j,k;
  char *pos;
  int posn;

  // Start the tree with the root and its
  // left and right elements to start off

  for(i=0;i<strlen(io);i++)
  {
     if(io[i]==po[0])
     {
        copy_str(t[1],io,1,i,i);             // We have the root here
        copy_str(t[2],io,2,0,i-1);           // Its left subtree
        copy_str(t[3],io,3,i+1,strlen(io));  // Its right subtree
        print_t();
     }
  }

  // Now construct the remaining tree
  for(i=1;i<strlen(po);i++)
  {
     for(j=1;j<=hpos;j++)
     {
        if((pos=strchr((const char *)t[j],po[i]))!=(char *)0 &&
strlen(t[j])!=1)
        {
            for(k=0;k<strlen(t[j]);k++)
            {
               if(t[j][k]==po[i]){posn=k;break;}
            }
            printf("\nSplitting [%s] for po[%d]=[%c] at %d..\n",
t[j],i,po[i],posn);

            copy_str(t[2*j],t[j],2*j,0,posn-1);
            copy_str(t[2*j+1],t[j],2*j+1,posn+1,strlen(t[j]));
            copy_str(t[j],t[j],j,posn,posn);
            print_t();
        }
```

```
      }
   }
}

// This function is used to split a string into three seperate strings
// This is used to create a root, its left subtree and its right
subtree

void copy_str(char dest[], char src[], int pos, int start, int end)
{
  char mysrc[100];
  strcpy(mysrc,src);
  dest[0]='\0';
  strncat(dest,mysrc+start,end-start+1);
  if(pos>hpos)hpos=pos;
}


void print_t()
{
  int i;
  for(i=1;i<=hpos;i++)
  {
    printf("\nt[%d] = [%s]", i, t[i]);
  }
  printf("\n");
}
```

Find the closest ancestor of two nodes in a tree.

Discuss it!

Here is some working C code...

```
#include <stdio.h>

typedef struct node
{
  int value;
  struct node *right;
  struct node *left;
}mynode;

mynode *root;

mynode *add_node(int value);
void levelOrderTraversal(mynode *root);
mynode *closestAncestor(mynode* root, mynode* p, mynode* q);


int main(int argc, char* argv[])
{
  mynode *node_pointers[7], *temp;
```

```
    root = NULL;

    // Create the BST.
    // Store the node pointers to use later...
    node_pointers[0] = add_node(5);
    node_pointers[1] = add_node(1);
    node_pointers[2] = add_node(-20);
    node_pointers[3] = add_node(100);
    node_pointers[4] = add_node(23);
    node_pointers[5] = add_node(67);
    node_pointers[6] = add_node(13);


    printf("\n\n\nLEVEL ORDER TRAVERSAL\n\n");
    levelOrderTraversal(root);


    // Calculate the closest ancestors of a few nodes..

    temp = closestAncestor(root, node_pointers[5], node_pointers[6]);
    printf("\n\nClosest ancestor of [%d] and [%d] is [%d]\n\n",
           node_pointers[5]->value,
           node_pointers[6]->value,
           temp->value);


    temp = closestAncestor(root, node_pointers[2], node_pointers[6]);
    printf("\n\nClosest ancestor of [%d] and [%d] is [%d]\n\n",
           node_pointers[2]->value,
           node_pointers[6]->value,
           temp->value);


    temp = closestAncestor(root, node_pointers[4], node_pointers[5]);
    printf("\n\nClosest ancestor of [%d] and [%d] is [%d]\n\n",
           node_pointers[4]->value,
           node_pointers[5]->value,
           temp->value);


    temp = closestAncestor(root, node_pointers[1], node_pointers[3]);
    printf("\n\nClosest ancestor of [%d] and [%d] is [%d]\n\n",
           node_pointers[1]->value,
           node_pointers[3]->value,
           temp->value);


    temp = closestAncestor(root, node_pointers[2], node_pointers[6]);
    printf("\n\nClosest ancestor of [%d] and [%d] is [%d]\n\n",
           node_pointers[2]->value,
           node_pointers[6]->value,
           temp->value);

  }
```

```
// Function to add a new node to the tree..
mynode *add_node(int value)
{
   mynode *prev, *cur, *temp;

   temp        = (mynode *) malloc(sizeof(mynode));
   temp->value = value;
   temp->right = NULL;
   temp->left  = NULL;

   if(root==NULL)
   {
     printf("\nCreating the root..\n");
     root = temp;
     return;
   }

   prev=NULL;
   cur=root;

   while(cur!=NULL)
   {
      prev=cur;
      cur=(value<cur->value)?cur->left:cur->right;
   }

   if(value < prev->value)
     prev->left=temp;
   else
     prev->right=temp;

   return(temp);

}



// Level order traversal..
void levelOrderTraversal(mynode *root)
{
  mynode *queue[100] = {(mynode *)0};
  int size = 0;
  int queue_pointer = 0;

  while(root)
  {
      printf("[%d] ", root->value);

      if(root->left)
      {
        queue[size++] = root->left;
      }

      if(root->right)
      {
        queue[size++] = root->right;
      }

      root = queue[queue_pointer++];
  }
}


// Function to find the closest ancestor...
mynode *closestAncestor(mynode* root, mynode* p, mynode* q)
{
   mynode *l, *r, *tmp;

   if(root == NULL)
   {
      return(NULL);
   }

   if(root->left==p || root->right==p || root->left==q || root-
>right==q)
   {
     return(root);
   }
   else
   {
      l = closestAncestor(root->left, p, q);
      r = closestAncestor(root->right, p, q);

      if(l!=NULL && r!=NULL)
      {
        return(root);
      }
      else
      {
         tmp = (l!=NULL) ? l : r;
         return(tmp);
      }
   }
}
```

Here is the tree for you to visualize...

```
                        5 (node=0)

       1 (node=1)                    100 (node=3)

-20 (node=2)                 23 (node=4)

                   13 (node=5)   67 (node=6)
```

```
Here is the output...

LEVEL ORDER TRAVERSAL

[5] [1] [100] [-20] [23] [13] [67]

Closest ancestor of [67] and [13] is [23]

Closest ancestor of [-20] and [13] is [5]

Closest ancestor of [23] and [67] is [100]

Closest ancestor of [1] and [100] is [5]

Closest ancestor of [-20] and [13] is [5]
```

Given an expression tree, evaluate the expression and obtain a paranthesized form of the expression.

Discuss it!

The code below prints the paranthesized form of a tree.

```
infix_exp(p)
{
   if(p)
   {
      printf("(");
      infix_exp(p->left);
      printf(p->data);
      infix_exp(p->right);
      printf(")");
   }
}
```

How do you convert a tree into an array?

Discuss it!

The conversion is based on these rules

```
If i > 1, i/2 is the parent
If 2*i > n, then there is no left child, else 2*i is the left child.
If (2*i + 1) > n, then there is no right child, else (2*i + 1) is the
```

right child.

Converting a tree to an array is very easy

Suppose we have a tree like this

```
      A
  B       C
D E     F G
```

The array representation would be

```
a[1] a[2] a[3] a[4] a[5] a[6] a[7]
  A    B    C    D    E    F    G
```

That is, for every node at position i in the array, its left child will be stored at position (2*i) and right child at (2*i + 1). The root starts at position 1.

What is an AVL tree?

Discuss it!

AVL trees are self-adjusting, height-balanced binary search trees and are named after the inventors: Adelson-Velskii and Landis. A balanced binary search tree has O(log n) height and hence O(log n) worst case search and insertion times. However, ordinary binary search trees have a bad worst case. When sorted data is inserted, the binary search tree is very unbalanced, essentially more of a linear list, with O(n) height and thus O(n) worst case insertion and lookup times. AVL trees overcome this problem.

An AVL tree is a binary search tree in which every node is height balanced, that is, the difference in the heights of its two subtrees is at most 1. The balance factor of a node is the height of its right subtree minus the height of its left subtree (right minus left!). An equivalent definition, then, for an AVL tree is that it is a binary search tree in which each node has a balance factor of -1, 0, or +1. Note that a balance factor of -1 means that the subtree is left-heavy, and a balance factor of +1 means that the subtree is right-heavy. Each node is associated with a Balancing factor.

Balance factor of each node = height of right subtree at that node - height of left subtree at that node.

Please be aware that we are talking about the height of the subtrees and not the weigths of the subtrees. This is a very important point. We are talking about the height!.

Here is some recursive, working! C code that sets the Balance factor for all nodes starting from the root....

```
#include <stdio.h>

typedef struct node
{
  int value;
  int visited;
  int bf;
  struct node *right;
  struct node *left;
}mynode;

mynode *root;

mynode *add_node(int value);
void levelOrderTraversal(mynode *root);
int setbf(mynode *p);


// The main function
int main(int argc, char* argv[])
{
  root = NULL;

  // Construct the tree..
  add_node(5);
  add_node(1);
  add_node(-20);
  add_node(100);
  add_node(23);
  add_node(67);
  add_node(13);

  // Set the balance factors
  setbf(root);

  printf("\n\n\nLEVEL ORDER TRAVERSAL\n\n");
  levelOrderTraversal(root);
  getch();
}

// Function to add a new node to the tree...
mynode *add_node(int value)
{
   mynode *prev, *cur, *temp;

   temp        = (mynode *) malloc(sizeof(mynode));
   temp->value = value;
   temp->visited = 0;
   temp->bf = 0;
   temp->right = NULL;
   temp->left  = NULL;

   if(root==NULL)
   {
     //printf("\nCreating the root..\n");
     root = temp;
     return;
   }

   prev=NULL;
   cur=root;

   while(cur!=NULL)
   {
      prev=cur;
      cur=(value<cur->value)?cur->left:cur->right;
   }

   if(value < prev->value)
     prev->left=temp;
   else
     prev->right=temp;

   return(temp);

}


// Recursive function to set the balancing factor
// of each node starting from the root!
int setbf(mynode *p)
{
   int templ, tempr;
   int count;
   count = 1;

   if(p == NULL)
   {
     return(0);
   }
   else
   {
       templ = setbf(p->left);
       tempr = setbf(p->right);

       if(templ < tempr)
         count = count + tempr;
       else
         count = count + templ;
   }

   // Set the nodes balancing factor.
   printf("\nNode = [%3d], Left sub-tree height = [%1d], Right sub-tree
height = [%1d], BF = [%1d]\n",
          p->value, templ, tempr, (tempr - templ));
   p->bf = tempr - templ;
   return(count);
}



// Level order traversal..
void levelOrderTraversal(mynode *root)
{
  mynode *queue[100] = {(mynode *)0};
  int size = 0;
  int queue_pointer = 0;

  while(root)
  {
      printf("\n[%3d] (BF : %3d) ", root->value, root->bf);

      if(root->left)
      {
        queue[size++] = root->left;
      }

      if(root->right)
      {
        queue[size++] = root->right;
      }

      root = queue[queue_pointer++];
  }
}
```

And here is the output...

```
Node = [-20], Left sub-tree height = [0], Right sub-tree height = [0],
BF = [0]
Node = [  1], Left sub-tree height = [1], Right sub-tree height = [0],
BF = [-1]
Node = [ 13], Left sub-tree height = [0], Right sub-tree height = [0],
BF = [0]
Node = [ 67], Left sub-tree height = [0], Right sub-tree height = [0],
BF = [0]
Node = [ 23], Left sub-tree height = [1], Right sub-tree height = [1],
BF = [0]
Node = [100], Left sub-tree height = [2], Right sub-tree height = [0],
BF = [-2]
Node = [  5], Left sub-tree height = [2], Right sub-tree height = [3],
BF = [1]


LEVEL ORDER TRAVERSAL

[  5] (BF :   1)
[  1] (BF :  -1)
[100] (BF :  -2)
[-20] (BF :   0)
[ 23] (BF :   0)
[ 13] (BF :   0)
[ 67] (BF :   0)
```

Here is the tree which we were dealing with above

```
            5
       1         100
-20           23
           13   67
```

After insertion, the tree might have to be readjusted as needed in order to maintain it as an AVL tree. A node with balance factor -2 or 2 is considered unbalanced and requires rebalancing the tree. The balance factor is either stored directly at each node or computed from the heights of the subtrees, possibly stored at nodes. If, due to an instertion or deletion, the tree becomes unbalanced, a corresponding left rotation or a right rotation is performed on that tree at a particular node. A balance factor > 1 requires a left rotation (i.e. the right subtree is heavier than the left subtree) and a balance factor < -1 requires a right rotation (i.e. the left subtree is heavier than the right subtree).

Here is some pseudo code to demonstrate the two types of rotations...

Left rotation

BEFORE

```
        0 (par)

   0          0 (p)

         0         0 (tmp)

       0   0     0   0
                (a) (b)
```

Here we left rotate the tree around node p

```
tmp         = p->right;
p->right    = tmp->left;
tmp->left   = p;

if(par)
{
   if(p is the left child of par)
   {
     par->left=tmp;
   }
   else
   {
     par->right=tmp;
   }
}
else
{
   root=tmp;
}

// Reclaculate the balance factors
setbf(root);
```

AFTER

```
        0 (par)

   0             0
               (tmp)

          0           0
         (p)         (b)

      0      0
            (a)

   0     0
```

Right rotation

BEFORE

```
        0 (par)

   0            0 (p)

         0 (tmp)      0
```

```
      0    0       0     0
     (a)  (b)
```

```
Here we right rotate the tree around node p


tmp         = p->left;
p->left     = tmp->right;
tmp->right  = p;

if(par)
{
   if(p is the left child of par)
   {
     par->left=tmp;
   }
   else
   {
     par->right=tmp;
   }
}
else
{
   root=tmp;
}

// Recalculate the balancing factors...
setbf(root);
```

```
AFTER

        0 (par)

   0            0 (tmp)

          0           0
         (a)         (p)

                 0         0
                (b)

                        0     0
```

Implement Breadth First Search (BFS) and Depth First Search (DFS)

Discuss it!

Depth first search (DFS)

Depth First Search (DFS) is a generalization of the preorder traversal. Starting at some arbitrarily chosen vertex v, we mark v so that we know we've visited it, process v, and then recursively traverse all unmarked vertices adjacent to v (v will be a different vertex with every new method call). When we visit a vertex in which all of its neighbors have been visited, we return to its calling vertex, and visit one of its unvisited neighbors, repeating the recursion in the same manner. We continue until we have visited all of the starting vertex's neighbors, which means that we're done. The recursion (stack) guides us through the graph.

```
public void depthFirstSearch(Vertex v)
{
        v.visited = true;

        // print the node

        for(each vertex w adjacent to v)
        {
           if(!w.visited)
           {
              depthFirstSearch(w);
           }
        }
}
```

Here is some working C code for a DFS on a BST..

```
#include <stdio.h>

typedef struct node
{
  int value;
  int visited;
  struct node *right;
  struct node *left;
}mynode;

mynode *root;

mynode *add_node(int value);
void treeDFS(mynode *root);
```

```
int main(int argc, char* argv[])
{
  root = NULL;

  // Construct the tree..
  add_node(5);
  add_node(1);
  add_node(-20);
  add_node(100);
  add_node(23);
  add_node(67);
  add_node(13);

  // Do a DFS..
  printf("\n\nDFS : ");
  treeDFS(root);

  getch();
}

// Function to add a new node to the tree...
mynode *add_node(int value)
{
   mynode *prev, *cur, *temp;

   temp        = (mynode *) malloc(sizeof(mynode));
   temp->value = value;
   temp->visited = 0;
   temp->right = NULL;
   temp->left  = NULL;

   if(root==NULL)
   {
     printf("\nCreating the root..\n");
     root = temp;
     return;
   }

   prev=NULL;
   cur=root;

   while(cur!=NULL)
   {
      prev=cur;
      cur=(value<cur->value)?cur->left:cur->right;
   }

   if(value < prev->value)
     prev->left=temp;
   else
     prev->right=temp;

   return(temp);

}
```

```
// DFS..
void treeDFS(mynode *root)
{
    printf("[%d] ", root->value);
    root->visited = 1;

    if (root->left)
    {
      if(root->left->visited==0)
      {
        treeDFS(root->left);
      }
    }

    if (root->right)
    {
      if(root->right->visited==0)
      {
        treeDFS(root->right);
      }
    }
}
```

Breadth First Search

Breadth First Search (BFS) searches the graph one level (one edge away from the starting vertex) at a time. In this respect, it is very similar to the level order traversal that we discussed for trees. Starting at some arbitrarily chosen vertex v, we mark v so that we know we've visited it, process v, and then visit and process all of v's neighbors. Now that we've visited and processed all of v's neighbors, we need to visit and process all of v's neighbors neighbors. So we go to the first neighbor we visited and visit all of its neighbors, then the second neighbor we visited, and so on. We continue this process until we've visited all vertices in the graph. We don't use recursion in a BFS because we don't want to traverse recursively. We want to traverse one level at a time. So imagine that you visit a vertex v, and then you visit all of v's neighbors w. Now you need to visit each w's neighbors. How are you going to remember all of your w's so that you can go back and visit their neighbors? You're already marked and processed all of the w's. How are you going to find each w's neighbors if you don't remember where the w's are? After all, you're not using recursion, so there's no stack to keep track of them. To perform a BFS, we use a queue. Every time we visit vertex w's neighbors, we dequeue w and enqueue w's neighbors. In this way, we can keep track of which neighbors belong to which vertex. This is the same technique that we saw for the level-order traversal of a tree. The only new trick is that we need to makr the verticies, so we don't visit them more than once -- and this isn't even new, since this technique was used for the blobs problem during our discussion of recursion.

```
public void breadthFirstSearch(vertex v)
{
   Queue q = new Queue();

   v.visited = true;
   q.enQueue(v);

   while( !q.isEmpty() )
   {
       Vertex w = (Vertex)q.deQueue();

       // Print the node.

       for(each vertex x adjacent to w)
       {
           if( !x.visited )
           {
              x.visited = true;
              q.enQueue(x);
           }
        }
    }
}
```

BFS traversal can be used to produce a tree from a graph.

Here is some C code which does a BFS (level order traversal) on a BST...

```
#include <stdio.h>

typedef struct node
{
  int value;
  struct node *right;
  struct node *left;
}mynode;

mynode *root;

add_node(int value);

void levelOrderTraversal(mynode *root);

int main(int argc, char* argv[])
{
  root = NULL;

  add_node(5);
  add_node(1);
  add_node(-20);
  add_node(100);
  add_node(23);
  add_node(67);
  add_node(13);


  printf("\n\n\nLEVEL ORDER TRAVERSAL\n\n");
  levelOrderTraversal(root);

  getch();
}




// Function to add a new node...
add_node(int value)
{
   mynode *prev, *cur, *temp;

   temp        = (mynode *) malloc(sizeof(mynode));
   temp->value = value;
   temp->right = NULL;
   temp->left  = NULL;

   if(root==NULL)
   {
     printf("\nCreating the root..\n");
     root = temp;
     return;
   }

   prev=NULL;
   cur=root;

   while(cur!=NULL)
   {
      prev=cur;
      cur=(value<cur->value)?cur->left:cur->right;
   }

   if(value < prev->value)
     prev->left=temp;
   else
     prev->right=temp;
}



// Level order traversal..
void levelOrderTraversal(mynode *root)
```

```
{
  mynode *queue[100] = {(mynode *)0}; // Important to initialize!
  int size = 0;
  int queue_pointer = 0;

  while(root)
  {
      printf("[%d] ", root->value);

      if(root->left)
      {
        queue[size++] = root->left;
      }

      if(root->right)
      {
        queue[size++] = root->right;
      }

      root = queue[queue_pointer++];
  }
}
```

Write pseudocode to add a new node to a Binary Search Tree (BST)

Discuss it!

```
Here is a C code to construct a BST right from scratch...


#include <stdio.h>

typedef struct node
{
  int value;
  struct node *right;
  struct node *left;
}mynode;

mynode *root;

add_node(int value);
void postorder(mynode *root);
void inorder(mynode *root);
void preorder(mynode *root);

int main(int argc, char* argv[])
{
  root = NULL;

  add_node(5);
  add_node(1);
  add_node(-20);
  add_node(100);
  add_node(23);
  add_node(67);
  add_node(13);

  printf("\nPreorder    : ");
  preorder(root);
  printf("\n\nPostorder : ");
  postorder(root);
  printf("\n\nInorder   : ");
  inorder(root);

  return(0);
}

// Function to add a new node...
add_node(int value)
{
   mynode *prev, *cur, *temp;

   temp        = (mynode *) malloc(sizeof(mynode));
   temp->value = value;
   temp->right = NULL;
   temp->left = NULL;

   if(root==NULL)
   {
     printf("\nCreating the root..\n");
     root = temp;
     return;
   }

   prev=NULL;
   cur=root;

   while(cur!=NULL)
   {
      prev=cur;
      cur=(value<cur->value)?cur->left:cur->right;
   }

   if(value < prev->value)
     prev->left=temp;
   else
     prev->right=temp;
}



void preorder(mynode *root)
{
  if(root)
  {
    printf("[%d] ", root->value);
    preorder(root->left);
    preorder(root->right);
  }
}
```

```
void postorder(mynode *root)
{
  if(root)
  {
    postorder(root->left);
    postorder(root->right);
    printf("[%d] ", root->value);
  }
}
```

```
void inorder(mynode *root)
{
  if(root)
  {
    inorder(root->left);
    printf("[%d] ", root->value);
    inorder(root->right);
  }
}
```

What is a threaded binary tree?

Discuss it!

Since traversing the three is the most frequent operation, a method must be devised to improve the speed. This is where Threaded tree comes into picture. If the right link of a node in a tree is NULL, it can be replaced by the address of its inorder successor. An extra field called the rthread is used. If rthread is equal to 1, then it means that the right link of the node points to the inorder success. If its equal to 0, then the right link represents an ordinary link connecting the right subtree.

```
struct node
{
  int value;
  struct node *left;
  struct node *right;
  int rthread;
}
```

Function to find the inorder successor

```
mynode *inorder_successor(mynode *x)
{
  mynode *temp;
  temp = x->right;

  if(x->rthread==1)return(temp);

  while(temp->left!=NULL)temp = temp->left;

  return(temp);
}
```

Function to traverse the threaded tree in inorder

```
void inorder(mynode *head)
{
   mynode *temp;

   if(head->left==head)
   {
      printf("\nTree is empty!\n");
      return;
   }

   temp = head;

   for(;;)
   {
       temp = inorder_successor(temp);
       if(temp==head)return;
       printf("%d ", temp->value);
   }

}
```

Inserting toward the left of a node in a threaded binary tree.

```
void insert(int item, mynode *x)
{
   mynode *temp;
   temp = getnode();
   temp->value = item;
   x->left = temp;
   temp->left=NULL;
   temp->right=x;
   temp->rthread=1;
}
```

Function to insert towards the right of a node in a threaded binary tree.

```
void insert_right(int item, mynode *x)
```

```
{
   mynode *temp, r;

   temp=getnode();
   temp->info=item;
   r=x->right;
   x->right=temp;
   x->rthread=0;
   temp->left=NULL;
   temp->right=r;
   temp->rthread=1;
}
```

Function to find the inorder predecessor (for a left threaded binary three)

```
mynode *inorder_predecessor(mynode *x)
{
     mynode *temp;

     temp = x->left;

     if(x->lthread==1)return(temp);

     while(temp->right!=NULL)
       temp=temp->right;

     return(temp);
}
```