

WWW.CRACKTHEINTERVIEW.COM

Contents

- Linked Lists
 - How do you reverse a singly linked list? How do you reverse a doubly linked list? Write a C program to do the same. **Updated!**
 - Given only a pointer to a node to be deleted in a singly linked list, how do you delete it? **Updated!**
 - How do you sort a linked list? Write a C program to sort a linked list.
 - How to declare a structure of a linked list?
 - Write a C program to implement a Generic Linked List.
 - How do you reverse a linked list without using any C pointers? **Updated!**
 - How would you detect a loop in a linked list? Write a C program to detect a loop in a linked list.
 - How do you find the middle of a linked list? Write a C program to return the middle of a linked list
 - If you are using C language to implement the heterogeneous linked list, what pointer type will you use?
 - How to compare two linked lists? Write a C program to compare two linked lists.
 - How to create a copy of a linked list? Write a C program to create a copy of a linked list.
 - Write a C program to free the nodes of a linked list **Updated!**
 - Can we do a Binary search on a linked list?
 - Write a C program to return the nth node from the end of a linked list. **New!**
 - How would you find out if one of the pointers in a linked list is corrupted or not? **New!**
- Write your own
 - Write your own C program to implement the atoi() function **Updated!**
 - Implement the memmove() function. What is the difference between the memmove() and memcpy() function? **Updated!**
 - Write C code to implement the strstr() (search for a substring) function.
 - Write your own printf() function in C
 - Implement the strcpy() function. **Updated!**
 - Implement the strcmp(str1, str2) function. **Updated!**
 - Implement the substr() function in C.
 - Write your own copy() function
 - Write C programs to implement the toupper() and the isupper() functions **Updated!**
 - Write a C program to implement your own strdup() function.
 - Write a C program to implement the strlen() function

- Programs
 - Write a C program to swap two variables without using a temporary variable
 - What is the 8 queens problem? Write a C program to solve it.
 - Write a C program to print a square matrix helically.
 - Write a C program to reverse a string **Updated!**
 - Write a C program to reverse the words in a sentence in place. **Updated!**
 - Write a C program generate permutations.
 - Write a C program for calculating the factorial of a number
 - Write a C program to calculate pow(x,n)?
 - Write a C program which does wildcard pattern matching algorithm
 - How do you calculate the maximum subarray of a list of numbers?
 - 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.
 - Solve the Rat In A Maze problem using backtracking.
 - 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?
 - Write C code to solve the Tower of Hanoi problem.
 - Write C code to return a string from a function **Updated!**
 - Write a C program which produces its own source code as its output
 - Write a C program to convert from decimal to any base (binary, hex, oct etc...)
 - Write C code to check if an integer is a power of 2 or not in a single line?
 - Write a C program to find the GCD of two numbers. **Updated!**
 - Finding a duplicated integer problem *
 - Write code to remove duplicates in a sorted array. **Updated!**
 - Find the maximum of three integers using the ternary operator.
 - How do you initialize a pointer inside a function?
 - Write C code to dynamically allocate one, two and three dimensional arrays (using malloc()) **New!**
 - How would you find the size of structure without using sizeof()?
 - Write a C program to multiply two matrices.
 - Write a C program to check for palindromes.
 - Write a C program to convert a decimal number into a binary number.
 - Write C code to implement the Binary Search algorithm.
 - Write code to evaluate a polynomial.
 - Write code to add two polynomials
 - Write a program to add two long positive numbers (each represented by linked lists).
 - How do you compare floating point numbers? **Updated!**
 - What's a good way to implement complex numbers in C?
 - How can I display a percentage-done indication on the screen?
 - Write a program to check if a given year is a leap year or not?
 - Is there something we can do in C but not in C++?
 - How to swap the two nibbles in a byte ?
 - How to scan a string till we hit a new line using scanf()?
 - Write pseudocode to compare versions (like 115.10.1 vs 115.11.5).

- How do you get the line numbers in C?
- How to fast multiply a number by 7?
- Write a simple piece of code to split a string at equal intervals
- Is there a way to multiply matrices in lesser than $O(n^3)$ time complexity?

- How do you find out if a machine is 32 bit or 64 bit? *
- Write a program to have the output go two places at once (to the screen and to a file also)
- Write code to round numbers
- How can we sum the digits of a given number in single statement?
- Given two strings A and B, how would you find out if the characters in B were a subset of the characters in A?
- 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
- How to add two numbers without using the plus operator?
- How to generate prime numbers? How to generate the next prime after a given prime?
- Write a program to print numbers from 1 to 100 without using loops!
- Write your own trim() or squeeze() function to remove the spaces from a string. **New!**
- Write your own random number generator function in C.*
- Write your own sqrt() function in C*

- Trees
 - Write a C program to find the depth or height of a tree.
 - Write a C program to determine the number of elements (or size) in a tree.

 - Write a C program to delete a tree (i.e, free up its nodes)
 - Write C code to determine if two trees are identical
 - Write a C program to find the minimum value in a binary search tree.
 - Write a C program to compute the maximum depth in a tree?
 - Write a C program to create a mirror copy of a tree (left nodes become right and right nodes become left)!
 - Write C code to return a pointer to the nth node of an inorder traversal of a BST. **New!**
 - Write C code to implement the preorder(), inorder() and postorder() traversals. Whats their time complexities? **Updated!**
 - Write a C program to create a copy of a tree
 - Write C code to check if a given binary tree is a binary search tree or not?

 - Write C code to implement level order traversal of a tree. **New!**
 - Write a C program to delete a node from a Binary Search Tree?
 - Write C code to search for a value in a binary search tree (BST).
 - Write C code to count the number of leaves in a tree
 - Write C code for iterative preorder, inorder and postorder tree traversals **New!**

- Can you construct a tree using postorder and preorder traversal?
- Construct a tree given its inorder and preorder traversal strings. Similarly construct a tree given its inorder and post order traversal strings. *Updated!*
- Find the closest ancestor of two nodes in a tree. *New!*
- Given an expression tree, evaluate the expression and obtain a parenthesized form of the expression.
- How do you convert a tree into an array?
- What is an AVL tree? *Updated!*
- How many different trees can be constructed using n nodes?
- A full N-ary tree has M non-leaf nodes, how many leaf nodes does it have?
- Implement Breadth First Search (BFS) and Depth First Search (DFS) *Updated!*
- Write pseudocode to add a new node to a Binary Search Tree (BST) *Updated!*
- What is a threaded binary tree?
- Bit Fiddling
 - Write a C program to count bits set in an integer? *Updated!*
 - What purpose do the bitwise and, or, xor and the shift operators serve?
 - How to reverse the bits in an integer?
 - Check if the 20th bit of a 32 bit integer is on or off?
 - How to reverse the odd bits of an integer?
 - How would you count the number of bits set in a floating point number?*
- Sorting Techniques
 - What is heap sort?
 - What is the difference between Merge Sort and Quick sort?
 - Give pseudocode for the mergesort algorithm
 - Implement the bubble sort algorithm. How can it be improved? Write the code for selection sort, quick sort, insertion sort.
- C Pointers
 - What does *p++ do? Does it increment p or the value pointed by p?
 - What is a NULL pointer? How is it different from an uninitialized pointer? How is a NULL pointer defined?
 - What is a null pointer assignment error?
 - 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?
 - Is the cast to malloc() required at all?

- 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?
- What's the difference between const char *p, char * const p and const char * const p? **Updated!**
- What is a void pointer? Why can't we perform arithmetic on a void * pointer?
- What do Segmentation fault, access violation, core dump and Bus error mean?
- What is the difference between an array of pointers and a pointer to an array?
- What is a memory leak?
- What are brk() and sbrk() used for? How are they different from malloc()?

- What is a dangling pointer? What are reference counters with respect to pointers?
- What do pointers contain?
- Is (*(p+i)+j) is equivalent to p[i][j]? Is num[i] == i[num] == *(num + i) == *(i + num)?
- What operations are valid on pointers? When does one get the Illegal use of pointer in function error?
- What are near, far and huge pointers? **New!**
- What is the difference between malloc() and calloc()? **New!**
- Why is sizeof() an operator and not a function? **New!**
- What is an opaque pointer?

- C Functions
 - How to declare a pointer to a function?
 - Does extern in a function declaration mean anything?
 - How can I return multiple values from a function?
 - Does C support function overloading?
 - What is the purpose of a function prototype?
 - What are inline functions?
 - How to declare an array of N pointers to functions returning pointers to functions returning pointers to characters?
 - Can we declare a function that can return a pointer to a function of the same type?
 - How can I write a function that takes a variable number of arguments? What are the limitations with this? What is vprintf()?
 - With respect to function parameter passing, what is the difference between call-by-value and call-by-reference? Which method does C use? **Updated!**

 - If I have the name of a function in the form of a string, how can I invoke that function? **New!**
 - What does the error, invalid redeclaration of a function mean?
 - How can I pass the variable argument list passed to one function to another function.

- How do I pass a variable number of function pointers to a variable argument (va_arg) function?
- Will C allow passing more or less arguments than required to a function.

- C Statements
 - Whats short-circuiting in C expressions?
 - Whats wrong with the expression a[i]=i++; ? Whats a sequence point?
Updated!
 - Does the ?: (ternary operator) return a lvalue? How can I assign a value to the output of the ternary operator?
 - Is 5[array] the same as array[5]?
 - What are #pragmas?
 - What is the difference between if(0 == x) and if(x == 0)?
 - Should we use goto or not?
 - Is ++i really faster than i = i + 1?
 - What do lvalue and rvalue mean?
 - What does the term cast refer to? Why is it used?
 - What is the difference between a statement and a block?
 - Can comments be nested in C?
 - What is type checking?
 - Why can't you nest structure definitions?
 - What is a forward reference?
 - What is the difference between the & and && operators and the | and || operators?
 - Is C case sensitive (ie: does C differentiate between upper and lower case letters)?
 - Can goto be used to jump across functions?
 - Whats wrong with #define myptr int *?
 - What purpose do #if, #else, #elif, #endif, #ifdef, #ifndef serve?
 - Can we use variables inside a switch statement? Can we use floating point numbers? Can we use expressions?
 - What is more efficient? A switch() or an if() else()?
 - What is the difference between a deep copy and a shallow copy?

- C Arrays
 - How to write functions which accept two-dimensional arrays when the width is not known before hand?
 - Is char a[3] = "abc"; legal? What does it mean?
 - If a is an array, is a++ valid?

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

- Do Global variables start out as zero?
- Does C have boolean variable type?
- Where may variables be defined in C?
- To what does the term storage class refer? What are auto, static, extern, volatile, const classes?
- What does the typedef keyword do?
- What is the difference between constants defined through #define and the constant keyword?
- What are Trigraph characters?
- How are floating point numbers stored? Whats the IEEE format?
Updated!
- C Structures
 - Can structures be assigned to variables and passed to and from functions?
 - Can we directly compare two structures using the == operator? **Updated!**
 - Can we pass constant values to functions which accept structure arguments?
 - How does one use fread() and fwrite()? Can we read/write structures to/from files?
 - Why do structures get padded? Why does sizeof() return a larger size?
 - Can we determine the offset of a field within a structure and directly access that element?
 - What are bit fields in structures? **Updated!**
 - What is a union? Where does one use unions? What are the limitations of unions? **New!**
- C Macros
 - How should we write a multi-statement macro?
 - How can I write a macro which takes a variable number of arguments?
 - What is the token pasting operator and stringizing operator in C?
 - Define a macro called SQR which squares a number. **Updated!**
- C Headers
 - What should go in header files? How to prevent a header file being included twice? Whats wrong with including more headers?
 - Is there a limit on the number of characters in the name of a header file?
- C File operations
 - How do stat(), fstat(), vstat() work? How to check whether a file exists?

- How can I insert or delete a line (or record) in the middle of a file?
 - How can I recover the file name using its file descriptor?
 - How can I delete a file? How do I copy files? How can I read a directory in a C program?
 - Whats the use of fopen(), fclose(), fprintf(), getc(), putc(), getw(), putw(), fscanf(), feof(), ftell(), fseek(), rewind(), fread(), fwrite(), fgets(), fputs(), freopen(), fflush(), ungetc()?
 - How to check if a file is a binary file or an ascii file? **New!**
- C Declarations and Definitions
 - What is the difference between char *a and char a[]? **Updated!**
 - How can I declare an array with only one element and still access elements beyond the first element (in a valid fashion)?
 - What is the difference between enumeration variables and the preprocessor #defines?
- C Functions - built-in
 - Whats the difference between gets() and fgets()? Whats the correct way to use fgets() when reading a file?
 - How can I have a variable field width with printf?
 - How can I specify a variable width in a scanf() format string?
 - How can I convert numbers to strings (the opposite of atoi)?
 - Why should one use strncpy() and not strcpy()? What are the problems with strncpy()?
 - How does the function strtok() work?
 - Why do we get the floating point formats not linked error?
 - Why do some people put void cast before each call to printf()?
 - What is assert() and when would I use it?
 - What do memcpy(), memchr(), memcmp(), memset(), strdup(), strncat(), strcmp(), strncmp(), strcpy(), strncpy(), strlen(), strchr(), strchr(), strpbrk(), strspn(), strcspn(), strtok() do? **Updated!**
 - What does alloca() do?
 - Can you compare two strings like string1==string2? Why do we need strcmp()?
 - What does printf() return? **New!**
 - What do setjmp() and longjump() functions do? **New!**
- C Functions - The main function
 - Whats the prototype of main()? Can main() return a structure?
 - Is exit(status) equivalent to returning the same status from main()?
 - Can main() be called recursively?
 - How to print the arguments recieved by main()?

- OS Concepts
 - What do the system calls fork(), vfork(), exec(), wait(), waitpid() do? What's a Zombie process? What's the difference between fork() and vfork()?
 - How does freopen() work? *
 - What are threads? What is a lightweight process? What is a heavyweight process? How different is a thread from a process? *
 - How are signals handled? *
 - What is a deadlock? *
 - What are semaphores? *
 - What is meant by context switching in an OS?*
 - What is Belady's anomaly?
 - What is thrashing?*
 - What are short-, long- and medium-term scheduling?
 - What are turnaround time and response time?
 - What is the Translation Lookaside Buffer (TLB)?
 - What is cycle stealing?
 - What is a reentrant program?
 - When is a system in safe state?
 - What is busy waiting?
 - What is pages replacement? What are local and global page replacements?*
 - What is meant by latency, transfer and seek time with respect to disk I/O?
 - What are monitors? How are they different from semaphores?*
 - In the context of memory management, what are placement and replacement algorithms?
 - What is paging? What are demand- and pre-paging?*
 - What is mounting?
 - What do you mean by dispatch latency?
 - What is multi-processing? What is multi-tasking? What is multi-threading? What is multi-programming?*
 - What is compaction?
 - What is memory-mapped I/O? How is it different from I/O mapped I/O?
 - List out some reasons for process termination.
- General Concepts
 - What is the difference between statically linked libraries and dynamically linked libraries (dll)? **Updated!**
 - What are the most common causes of bugs in C?
 - What is hashing?
 - What do you mean by Predefined?
 - What is data structure?
 - What are the time complexities of some famous algorithms?
 - What is row major and column major form of storage in matrices?

- Explain the BigOh notation. *Updated!*
 - Give the most important types of algorithms.
 - What is marshalling and demarshalling? *New!*
 - What is the difference between the stack and the heap? Where are the different types of variables of a program stored in memory? *New!*
 - Describe the memory map of a C program.
 - What is infix, prefix, postfix? How can you convert from one representation to another? How do you evaluate these expressions?
 - How can we detect and prevent integer overflow and underflow?*
- Compiling and Linking
 - How to list all the predefined identifiers?
 - How the compiler make difference between C and C++?
 - What are the general steps in compilation?
 - What are the different types of linkages? *New!*
 - What do you mean by scope and duration? *New!*
- HTTP
 - What are the basics of http? What is the difference be http and https? *
- ProC
 - How to work in ProC?
- Miscellaneous
 - Gotcha! programs *New!*
- SQL/PL-SQL/Database
 - Most frequently asked SQL/PL-SQL/Database questions *
- Unix/Linux
 - Most frequently asked unix questions *
- Quantitative Aptitude
 - Most frequently asked Quantitative aptitude questions*

- Graphs
 - Most frequently asked questions on Graphs*

- OOPs/OOAD
 - Most frequently asked OOPs and OOAD questions*

- Computer Networks
 - Most frequently asked questions on Computer Networks*

- Client Server Technologies
 - Most frequently asked Client - Server questions*

Linked Lists

(1 / 268)

How do you reverse a singly linked list? How do you reverse a doubly linked list? Write a C program to do the same.

This is THE most frequently asked interview question. The most!.

Singly linked lists

Here are a few C programs to reverse a singly linked list.

Method1 (Iterative)

```
#include <stdio.h>

// Variables
typedef struct node
{
    int value;
    struct node *next;
}mynode;

// Globals (not required, though).
mynode *head, *tail, *temp;

// Functions
void add(int value);
void iterative_reverse();
void print_list();

// The main() function
int main()
{
    head=(mynode *)0;

    // Construct the linked list.
    add(1);
    add(2);
    add(3);

    //Print it
    print_list();

    // Reverse it.
    iterative_reverse();

    //Print it again
    print_list();

    return(0);
}
```

```

// The reverse function
void iterative_reverse()
{
    mynode *p, *q, *r;

    if(head == (mynode *)0)
    {
        return;
    }

    p      = head;
    q      = p->next;
    p->next = (mynode *)0;

    while (q != (mynode *)0)
    {
        r      = q->next;
        q->next = p;
        p      = q;
        q      = r;
    }

    head = p;
}

// Function to add new nodes to the linked list
void add(int value)
{
    temp = (mynode *) malloc(sizeof(struct node));
    temp->next=(mynode *)0;
    temp->value=value;

    if(head==(mynode *)0)
    {
        head=temp;
        tail=temp;
    }
    else
    {
        tail->next=temp;
        tail=temp;
    }
}

// Function to print the linked list.
void print_list()
{
    printf("\n\n");
    for(temp=head; temp!=(mynode *)0; temp=temp->next)
    {
        printf("[%d]->", (temp->value));
    }
    printf("[NULL]\n\n");
}

```

Method2 (Recursive, without using any temporary variable)

```
#include <stdio.h>

// Variables
typedef struct node
{
    int value;
    struct node *next;
}mynode;

// Globals.
mynode *head, *tail, *temp;

// Functions
void add(int value);
mynode* reverse_recurse(mynode *root);
void print_list();

// The main() function
int main()
{
    head=(mynode *)0;

    // Construct the linked list.
    add(1);
    add(2);
    add(3);

    //Print it
    print_list();

    // Reverse it.
    if(head != (mynode *)0)
    {
        temp = reverse_recurse(head);
        temp->next = (mynode *)0;
    }

    //Print it again
    print_list();

    return(0);
}

// Reverse the linked list recursively
//

// This function uses the power of the stack to make this
// *magical* assignment
//
// node->next->next=node;
//
```

```

// :)

mynode* reverse_recurse(mynode *root)
{
    if(root->next!=(mynode *)0)
    {
        reverse_recurse(root->next);
        root->next->next=root;
        return(root);
    }
    else
    {
        head=root;
    }
}

// Function to add new nodes to the linked list.
void add(int value)
{
    temp = (mynode *) malloc(sizeof(struct node));
    temp->next=(mynode *)0;
    temp->value=value;

    if(head==(mynode *)0)
    {
        head=temp;
        tail=temp;
    }
    else
    {
        tail->next=temp;
        tail=temp;
    }
}

// Function to print the linked list.
void print_list()
{
    printf("\n\n");
    for(temp=head; temp!=(mynode *)0; temp=temp->next)
    {
        printf("[%d]->", (temp->value));
    }
    printf("[NULL]\n\n");
}

```

Method3 (Recursive, but without ANY global variables. Slightly messy!)

```

#include <stdio.h>

// Variables
typedef struct node
{
    int value;
    struct node *next;
}mynode;

```



```

// Functions
void add(mynode **head, mynode **tail, int value);
mynode* reverse_recurse(mynode *current, mynode *next);
void print_list(mynode *);

int main()
{
    mynode *head, *tail;
    head=(mynode *)0;

    // Construct the linked list.
    add(&head, &tail, 1);
    add(&head, &tail, 2);
    add(&head, &tail, 3);

    //Print it
    print_list(head);

    // Reverse it.
    head = reverse_recurse(head, (mynode *)0);

    //Print it again
    print_list(head);

    getch();
    return(0);
}

// Reverse the linked list recursively
mynode* reverse_recurse(mynode *current, mynode *next)
{
    mynode *ret;

    if(current==(mynode *)0)
    {
        return((mynode *)0);
    }

    ret = (mynode *)0;
    if (current->next != (mynode *)0)
    {
        ret = reverse_recurse(current->next, current);
    }
    else
    {
        ret = current;
    }

    current->next = next;
    return ret;
}

// Function to add new nodes to the linked list.
// Takes pointers to pointers to maintain the
// *actual* head and tail pointers (which are local to main()).

```

```

void add(mynode **head, mynode **tail, int value)
{
    mynode *temp1, *temp2;

    temp1 = (mynode *) malloc(sizeof(struct node));
    temp1->next=(mynode *)0;
    temp1->value=value;

    if(*head==(mynode *)0)
    {
        *head=temp1;
        *tail=temp1;
    }
    else
    {
        for(temp2 = *head; temp2->next!= (mynode *)0; temp2=temp2-
>next);
        temp2->next = temp1;
        *tail=temp1;
    }
}

// Function to print the linked list.
void print_list(mynode *head)
{
    mynode *temp;
    printf("\n\n");
    for(temp=head; temp!=(mynode *)0; temp=temp->next)
    {
        printf("[%d]->", (temp->value));
    }
    printf("[NULL]\n\n");
}

```

Doubly linked lists

This is really easy, just keep swapping the prev and next pointers and at the end swap the head and the tail:)

```

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

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

mynode *head, *tail;
void add_node(int value);
void print_list();
void reverse();

int main()
{

```

```

head=NULL;
tail=NULL;

add_node(1);
add_node(2);
add_node(3);
add_node(4);
add_node(5);

print_list();

reverse();

print_list();

return(1);
}

void add_node(int value)
{
    mynode *temp, *cur;
    temp = (mynode *)malloc(sizeof(mynode));
    temp->next=NULL;
    temp->prev=NULL;

    if(head == NULL)
    {
        printf("\nAdding a head pointer\n");
        head=temp;
        tail=temp;
        temp->value=value;
    }
    else
    {
        for(cur=head;cur->next!=NULL;cur=cur->next);
        cur->next=temp;
        temp->prev=cur;
        temp->value=value;
        tail=temp;
    }
}

void print_list()
{
    mynode *temp;

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

```

```

    for (temp=head; temp!=NULL; temp=temp->next)
    {
        printf("\n[%d]\n", temp->value);
    }
}

void reverse()
{
    mynode *cur, *temp, *save_next;
    if (head==tail) return;
    if (head==NULL || tail==NULL) return;
    for (cur=head; cur!=NULL; )
    {
        printf("\ncur->value : [%d]\n", cur->value);
        temp=cur->next;
        save_next=cur->next;
        cur->next=cur->prev;
        cur->prev=temp;
        cur=save_next;
    }

    temp=head;
    head=tail;
    tail=temp;
}

```

Having shown all these different methods, if someone can mail me a really, really good practical application of reversing a linked list (singly or doubly linked list), I would be really thankful to them. I have not found one good application of this. All I see is an urge to understand how well the candidate handles the pointer manipulation.

2- Given only a pointer to a node to be deleted in a singly linked list, how do you delete it?

The solution to this is to copy the data from the next node into this node and delete the next node!. Ofcourse this wont work if the node to be deleted is the last node. Mark it as dummy in that case. If you have a Circular linked list, then this might be all the more interesting. Try writing your own C program to solve this problem. Having a doubly linked list is always better.

3- How do you sort a linked list? Write a C program to sort a linked list.

This is a very popular interview question, which most people go wrong. The ideal solution to this problem is to keep the linked list sorted as you build it. This really saves a lot of time which would have been required to sort it.

Method1 (Usual method)

The general idea is to decide upon a sorting algorithm (say bubble sort). Then, one needs to come up with different scenarios to swap two nodes in the linked list when they are not in the required order. The different scenarios would be something like

1. When the nodes being compared are not adjacent and one of them is the first node.
2. When the nodes being compared are not adjacent and none of them is the first node
3. When the nodes being compared are adjacent and one of them is the first node.
4. When the nodes being compared are adjacent and none of them is the first node.

One example bubble sort for a linked list goes like this

```
for(i = 1; i < n; i++)
{
    p1 = head;
    p2 = head->next;
    p3 = p2->next;

    for(j = 1; j <= (n - i); j++)
    {
        if(p2->value < p3->value)
        {
            p2->next = p3->next;
            p3->next = p2;
            p1->next = p3;
            p1 = p3;
            p3 = p2->next;
        }
        else
        {
            p1 = p2;
            p2 = p3;
            p3 = p3->next;
        }
    }
}
```

As you can see, the code becomes quite messy because of the pointer logic. That's why I have not elaborated too much on the code, nor on variations such as sorting a doubly linked list. You have to do it yourself once to understand it.

Method1 (Divide and Conquer using merge sort)

The pseudocode for this method is

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

mynode *head, *tail;
int size;

mynode *mergesort(mynode *list, int size);
```

```

void display(mynode *list);

mynode *mergesort(mynode *list, int size)
{
    int size1, size2;
    mynode *tempnode1, *tempnode2, *tempnode3;

    if(size<=2)
    {
        if(size==1)
        {
            // Nothing to sort!
            return(list);
        }
        else
        {
            if(list->value < list->next->value
t            {
                // These 2 nodes are already in right order, no need to sor

                return(list);
            }
            else
            {
                // Need to swap these 2 nodes
                /* Here we have 2 nodes
                 *
                 * node 1 -> node2 -> NULL
                 *
                 * This should be converted to
                 *
                 * node2 -> node1 -> NULL
                 *
                 */

                tempnode1 = list;
                tempnode2 = list->next;
                tempnode2->next = tempnode1;
                tempnode1->next = NULL;
                return(tempnode2);
            }
        }
    }
    else
    {
        // The size of the linked list is more than 2.
        // Need to split this linked list, sort the
        // left and right sub-linked lists and merge.

        // Split.
        // tempnode1 will have the first half of the linked list of size
        "size1".
        // tempnode2 will have the second half of the linked list of size
        "size2".

        <CODE TO SPLIT THE LINKED LIST INTO TWO>

        // Sort the two halves recursively

        tempnode1 = mergesort(tempnode1, size1);
    }
}

```

```

        tempnode2 = mergesort(tempnode2, size2);

        // Now merge the sorted lists back, let tempnode3 point to that new list.

        <CODE TO MERGE THE 2 LINKED LISTS BACK INTO A SINGLE SORTED LINKED LIST>

        return(tempnode3);
    }
}

```

The code to merge the two already sorted sub-linked lists into a sorted linked list could be something like this..

```

mynode * merge(mynode *a, mynode *b)
{
    mynode *i, *j, *k, *c;

    i = a;
    j = b;
    c = getNewNode();
    k = getNewNode();

    while(i!=NULL && j!=NULL)
    {
        if(i->value < j->value)
        {
            k->next=i;
            i=i->next;
        }
        else
        {
            k->next=j;
            j=j->next;
        }
    }

    if(i!=NULL)
        k->next=i;
    else
        k->next=j;

    return(c->next);
}

```

4- How to declare a structure of a linked list?

The right way of declaring a structure for a linked list in a C program is

```

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

```

Note that the following are not correct

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

The typedef is not defined at the point where the "next" field is declared.

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

You can only have pointer to structures, not the structure itself as its recursive!

5 -Write a C program to implement a Generic Linked List.

Here is a C program which implements a generic linked list. This is also one of the very popular interview questions thrown around. The crux of the solution is to use the void C pointer to make it generic. Also notice how we use function pointers to pass the address of different functions to print the different generic data.

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

typedef struct list {
    void *data;
    struct list *next;
} List;

struct check {
    int i;
    char c;
    double d;
} chk[] = { { 1, 'a', 1.1 },
            { 2, 'b', 2.2 },
            { 3, 'c', 3.3 } };

void insert(List **, void *, unsigned int);
void print(List *, void (*)(void *));
void printstr(void *);
void printint(void *);
void printchar(void *);
void printcomp(void *);

List *list1, *list2, *list3, *list4;

int main(void)
{
    char c[]    = { 'a', 'b', 'c', 'd' };
    int i[]     = { 1, 2, 3, 4 };
    char *str[] = { "hello1", "hello2", "hello3", "hello4" };
}
```



```

list1 = list2 = list3 = list4 = NULL;

insert(&list1, &c[0], sizeof(char));
insert(&list1, &c[1], sizeof(char));
insert(&list1, &c[2], sizeof(char));
insert(&list1, &c[3], sizeof(char));

insert(&list2, &i[0], sizeof(int));
insert(&list2, &i[1], sizeof(int));
insert(&list2, &i[2], sizeof(int));
insert(&list2, &i[3], sizeof(int));

insert(&list3, str[0], strlen(str[0])+1);
insert(&list3, str[1], strlen(str[0])+1);
insert(&list3, str[2], strlen(str[0])+1);
insert(&list3, str[3], strlen(str[0])+1);

insert(&list4, &chk[0], sizeof chk[0]);
insert(&list4, &chk[1], sizeof chk[1]);
insert(&list4, &chk[2], sizeof chk[2]);

printf("Printing characters:");
print(list1, printchar);
printf(" : done\n\n");

printf("Printing integers:");
print(list2, printint);
printf(" : done\n\n");

printf("Printing strings:");
print(list3, printstr);
printf(" : done\n\n");

printf("Printing composite:");
print(list4, printcomp);
printf(" : done\n");

return 0;
}

void insert(List **p, void *data, unsigned int n)
{
    List *temp;
    int i;

    /* Error check is ignored */
    temp = malloc(sizeof(List));
    temp->data = malloc(n);
    for (i = 0; i < n; i++)
        *(char *) (temp->data + i) = *(char *) (data + i);
    temp->next = *p;
    *p = temp;
}

void print(List *p, void (*f)(void *))
{
    while (p)
    {
        (*f) (p->data);
        p = p->next;
    }
}

```

```

    }
}

void printstr(void *str)
{
    printf(" \"%s\"", (char *)str);
}

void printint(void *n)
{
    printf(" %d", *(int *)n);
}

void printchar(void *c)
{
    printf(" %c", *(char *)c);
}

void printcomp(void *comp)
{
    struct check temp = *(struct check *)comp;
    printf(" 'd:%c:%f", temp.i, temp.c, temp.d);
}

```

6-How do you reverse a linked list without using any C pointers?

One way is to reverse the data in the nodes without changing the pointers themselves. One can also create a new linked list which is the reverse of the original linked list. A simple C program can do that for you. Please note that you would still use the "next" pointer fields to traverse through the linked list (So in effect, you are using the pointers, but you are not changing them when reversing the linked list).

7- How would you detect a loop in a linked list? Write a C program to detect a loop in a linked list.

This is also one of the classic interview questions

There are multiple answers to this problem. Here are a few C programs to attack this problem.

Brute force method

Have a double loop, where you check the node pointed to by the outer loop, with every node of the inner loop.

```

typedef struct node
{
    void *data;
    struct node *next;
}mynode;

mynode * find_loop(NODE * head)
{
    mynode *current = head;

```

```

while(current->next != NULL)
{
    mynode *temp = head;
    while(temp->next != NULL && temp != current)
    {
        if(current->next == temp)
        {
            printf("\nFound a loop.");
            return current;
        }
        temp = temp->next;
    }
    current = current->next;
}
return NULL;
}

```

Visited flag

Have a visited flag in each node of the linked list. Flag it as visited when you reach the node. When you reach a node and the flag is already flagged as visited, then you know there is a loop in the linked list.

Fastest method

Have 2 pointers to start of the linked list. Increment one pointer by 1 node and the other by 2 nodes. If there's a loop, the 2nd pointer will meet the 1st pointer somewhere. If it does, then you know there's one.

Here is some code

```

p=head;
q=head->next;

while(p!=NULL && q!=NULL)
{
    if(p==q)
    {
        //Loop detected!
        exit(0);
    }
    p=p->next;
    q=(q->next)?(q->next->next):q->next;
}

// No loop.

```

8 -How do you find the middle of a linked list? Write a C program to return the middle of a linked list

Another popular interview question

Here are a few C program snippets to give you an idea of the possible solutions.

Method1

```

p=head;
q=head;

if (q->next->next!=NULL)
{
    p=p->next;
    q=q->next->next;
}

printf("The middle element is %d",p->data);

```

Here p moves one step, where as q moves two steps, when q reaches end, p will be at the middle of the linked list.

Method2

```

struct node *middle(struct node *head)
{
    struct node *middle=NULL;
    int i;

    for(i=1; head; head=head->next, i++)
    {
        if(i==1)
            middle=head;
        else if ((i%2)==1)
            middle=middle->next;
    }

    return middle;
}

```

In a similar way, we can find the 1/3 th node of linked list by changing (i%2==1) to (i%3==1) and in the same way we can find nth node of list by changing (i%2==1) to (i%n==1) but make sure ur (n<=i).

9-If you are using C language to implement the heterogeneous linked list, what pointer type will you use?

The heterogeneous linked list contains different data types in its nodes and we need a link, pointer to connect them. It is not possible to use ordinary pointers for this. So we go for void pointer. Void pointer is capable of storing pointer to any type as it is a generic pointer type.

Check out the C program to implement a Generic linked list in the same FAQ.

10- How to compare two linked lists? Write a C program to compare two linked lists.

```

int compare_linked_lists(struct node *q, struct node *r)
{
    static int flag;

```

```

    if((q==NULL ) && (r==NULL))
    {
        flag=1;
    }
    else
    {
        if(q==NULL || r==NULL)
        {
            flag=0;
        }
        if(q->data!=r->data)
        {
            flag=0;
        }
        else
        {
            compare_linked_lists(q->link,r->link);
        }
    }
    return(flag);
}

```

Another way is to do it on similar lines as strcmp() compares two strings, character by character (here each node is like a character).

11- How to create a copy of a linked list? Write a C program to create a copy of a linked list.

Check out this C program which creates an exact copy of a linked list.

```

copy_linked_lists(struct node *q, struct node **s)
{
    if(q!=NULL)
    {
        *s=malloc(sizeof(struct node));
        (*s)->data=q->data;
        (*s)->link=NULL;
        copy_linked_list(q->link, &((*s)->link));
    }
}

```

12- Write a C program to free the nodes of a linked list

Before looking at the answer, try writing a simple C program (with a for loop) to do this. Quite a few people get this wrong.

This is the wrong way to do it

```

struct list *listptr, *nextptr;
for(listptr = head; listptr != NULL; listptr = listptr->next)
{
    free(listptr);
}

```

If you are thinking why the above piece of code is wrong, note that once you free the listptr node, you cannot do something like listptr = listptr->next!. Since listptr is already

freed, using it to get listptr->next is illegal and can cause unpredictable results!

This is the right way to do it

```
struct list *listptr, *nextptr;
for(listptr = head; listptr != NULL; listptr = nextptr)
{
    nextptr = listptr->next;
    free(listptr);
}
```

13- Can we do a Binary search on a linked list?

Great C datastructure question!

The answer is ofcourse, you can write a C program to do this. But, the question is, do you really think it will be as efficient as a C program which does a binary search on an array?

Think hard, real hard.

Do you know what exactly makes the binary search on an array so fast and efficient? Its the ability to access any element in the array in constant time. This is what makes it so fast. You can get to the middle of the array just by saying array[middle]!. Now, can you do the same with a linked list? The answer is No. You will have to write your own, possibly inefficient algorithm to get the value of the middle node of a linked list. In a linked list, you loose the ability to get the value of any node in a constant time.

One solution to the inefficiency of getting the middle of the linked list during a binary search is to have the first node contain one additional pointer that points to the node in the middle. Decide at the first node if you need to check the first or the second half of the linked list. Continue doing that with each half-list.

14- Write a C program to return the nth node from the end of a linked list.

Here is a solution which is often called as the solution that uses frames.

Suppose one needs to get to the 6th node from the end in this LL. First, just keep on incrementing the first pointer (ptr1) till the number of increments cross n (which is 6 in this case)

```
STEP 1      :    1(ptr1,ptr2) -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -
> 10
```

```
STEP 2      :    1(ptr2) -> 2 -> 3 -> 4 -> 5 -> 6(ptr1) -> 7 -> 8 -> 9 -
> 10
```

Now, start the second pointer (ptr2) and keep on incrementing it till the first pointer (ptr1) reaches the end of the LL.

```
STEP 3      :    1 -> 2 -> 3 -> 4(ptr2) -> 5 -> 6 -> 7 -> 8 -> 9 -
> 10 (ptr1)
```

So here you have!, the 6th node from the end pointed to by ptr2!

Here is some C code..

```
struct node
{
    int data;
    struct node *next;
}mynode;

mynode * nthNode(mynode *head, int n /*pass 0 for last node*/)
{
    mynode *ptr1,*ptr2;
    int count;

    if(!head)
    {
        return(NULL);
    }

    ptr1 = head;
    ptr2 = head;
    count = 0;

    while(count < n)
    {
        count++;
        if((ptr1=ptr1->next)==NULL)
        {
            //Length of the linked list less than n. Error.
            return(NULL);
        }
    }

    while((ptr1=ptr1->next)!=NULL)
    {
        ptr2=ptr2->next;
    }

    return(ptr2);
}
```

15- How would you find out if one of the pointers in a linked list is corrupted or not?

This is a really good interview question. The reason is that linked lists are used in a wide variety of scenarios and being able to detect and correct pointer corruptions might be a very valuable tool. For example, data blocks associated with files in a file system are usually stored as linked lists. Each data block points to the next data block. A single corrupt pointer can cause the entire file to be lost!

- Discover and fix bugs when they corrupt the linked list and not when effect becomes visible in some other part of the program. Perform frequent consistency checks (to see if the linked list is indeed holding the data that you inserted into it).
- It is good programming practice to set the pointer value to NULL immediately after freeing the memory pointed at by the pointer. This will help in debugging, because it will tell you that the object was freed somewhere beforehand. Keep track of how many objects are pointing to a object using reference counts if required.
- Use a good debugger to see how the datastructures are getting corrupted and trace down the problem. Debuggers like ddd on linux and memory profilers like Purify, Electric fence are good starting points. These tools should help you track down heap corruption issues easily.
- Avoid global variables when traversing and manipulating linked lists. Imagine what would happen if a function which is only supposed to traverse a linked list using a global head pointer accidentally sets the head pointer to NULL!.
- Its a good idea to check the addNode() and the deleteNode() routines and test them for all types of scenarios. This should include tests for inserting/deleting nodes at the front/middle/end of the linked list, working with an empty linked list, running out of memory when using malloc() when allocating memory for new nodes, writing through NULL pointers, writing more data into the node fields then they can hold (resulting in corrupting the (probably adjacent) "prev" and "next" pointer fields), make sure bug fixes and enhancements to the linked list code are reviewed and well tested (a lot of bugs come from quick and dirty bug fixing), log and handle all possible errors (this will help you a lot while debugging), add multiple levels of logging so that you can dig through the logs. The list is endless...
- Each node can have an extra field associated with it. This field indicates the number of nodes after this node in the linked list. This extra field needs to be kept up-to-date when we insert or delete nodes in the linked list (It might become slightly complicated when insertion or deletion happens not at end, but anywhere in the linked list). Then, if for any node, $p \rightarrow \text{field} > 0$ and $p \rightarrow \text{next} == \text{NULL}$, it surely points to a pointer corruption.
- You could also keep the count of the total number of nodes in a linked list and use it to check if the list is indeed having those many nodes or not.

The problem in detecting such pointer corruptions in C is that its only the programmer who knows that the pointer is corrupted. The program has no way of knowing that something is wrong. So the best way to fix these errors is check your logic and test your code to the maximum possible extent. I am not aware of ways in C to recover the lost nodes of a corrupted linked list.

I have a hunch that interviewers who ask this question are probably hinting at something called Smart Pointers in C++. Smart pointers are particularly useful in the face of exceptions as they ensure proper destruction of dynamically allocated objects. They can also be used to keep track of dynamically allocated objects shared by multiple owners.

This topic is out of scope here, but you can find lots of material on the Internet for Smart Pointers.

16-Write your own (16 / 287)

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

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 essence is to understand the core logic first.

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

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, "ABCDEFGH");

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.
        // memcpy() would have done this directly.

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

        return(to);
    }
}

```

And here is the output

```

-----

From (before) = [ABCDEFGH]
To   (before) = [CDEFGH]

From (after) = [ABABCDEFGH]
To   (after) = [ABCDEFGH]

-----

From (before) = [ABCDEFGH]
To   (before) = [α␣ABCDEFGH]

From (after) = [CDEFGH]
To   (after) = [ABCDEFGH]

-----

From (before) = [ABCDEFGH]
To   (before) = [FEδ!!&:F]

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

-----

```

So then, what's the difference between the implementation of `memmove()` and `memcpy()`. It's 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 use depending on the type of the overlap. Also note that the C code proves that the results are the same irrespective of the Endianness of the machine.

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

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 occurrence 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 occurrence 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 occurrence 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 occurrence 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.

19 - Write your own printf() function in C

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 complicated 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 * fmt,...)
{
    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, 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);
}

```

20 - Implement the strcpy() function.

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);
}

```

21 - Implement the strcmp(str1, str2) function.

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]
```

22 - Implement the substr() function in C.

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);
}
```

23 - Write your own copy() function

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);
}

```

24 - Write C programs to implement the toupper() and the isupper() functions toUpper()

```

int toUpper(int ch)
{
    if(ch>='a' && c<='z')
        return('A' + ch - 'a');
    else
        return(ch);
}

```

isUpper()

```

int isUpper(int ch)
{
    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 UP
    PER 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.

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

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;
}

```

26 - a C program to implement the strlen() function

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);
}

```

Programs (27 / 268)

27 -e a C program to swap two variables without using a temporary variable

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 won't 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, don't try to be smart when writing code to swap variables. Use a temporary variable. It's not only fool proof, but also easier to understand and maintain.

28 - is the 8 queens problem? Write a C program to solve 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);
            }
        }
    }
}

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.
```

29 - a C program to print a square matrix helically.

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;
}
```

30 - Write a C program to reverse a string

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;
}

```

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

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!

32 - Write a C program generate permutations.

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;
        }
    }
}

```

33 - a C program for calculating the factorial of a number

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.

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

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.

35 - Write a C program which does wildcard pattern matching algorithm

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 = "hereheroheerr";
    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;
}

```

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

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.

37 - 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.

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

$$\begin{bmatrix} 1 & 1 \end{bmatrix}^n = \begin{bmatrix} F(n+1) & F(n) \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \end{bmatrix} \quad \begin{bmatrix} F(n) & F(n-1) \end{bmatrix}$$

or

$$\begin{pmatrix} f(n) & f(n+1) \end{pmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{pmatrix} f(n+1) & f(n+2) \end{pmatrix}$$

or

$$\begin{pmatrix} f(0) & f(1) \end{pmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n = \begin{pmatrix} f(n) & f(n+1) \end{pmatrix}$$

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 Fibonacci 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.

38 - Solve the Rat In A Maze problem using backtracking.

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 "extra" boundary we have added). The ones inside specify the obstacles.

```

111111111111111111111111111111111111
100000000000000000000000000000000001
100000001000000000000000000000000001
100000001000000000000000000000000001
100000000010000010000001000000000001
100000100000100000000000000000000001
100000000010000000000000000000000001
100000000000000000000000000000000001
111111111111111111111111111111111111

```

- 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 can't 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);
}
}

```

39 - 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?

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));
}
```

40 - Write C code to solve the Tower of Hanoi problem.

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);
    }
}
```

41 - Write C code to return a string from a function

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?);
}
```

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

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.

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

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]);
    }
}
```

```

    }
}

```

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

Even this is one of the most frequently asked interview questions. I really don't know what's so great in it. Nevertheless, here is a C program

Method1

```

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

```

Method2

```

if(((~i+1)&i)==i)
{
    //Power of 2!
}

```

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

Here is a C program

```

#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.

46 - Finding a duplicated integer problem

47 - Write code to remove duplicates in a sorted array.

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.

```
#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] [12
3]
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 complicated, but more visual version of the `removeDuplicates()` function. It shrinks the original array as and when it finds 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 dupli
cates
// 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 dup
licate...
        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 "du
p_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 num
bers that
// are duplicated.
//
// Dont bother too much about this function, it just helps in understan

```

```

ding
// 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!.

48 - Find the maximum of three integers using the ternary operator.

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;
}
```

49 - How do you initialize a pointer inside a function?

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());
}

```

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

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>;
    }

```

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

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;
}

```

52 - Write a C program to multiply two matrices.

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];
    }
}

```

53 - Write a C program to check for palindromes.

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

```

boolean palindrome(char string[])
{
    int count, countback, end, N;

    N = strlen(string);
    end = N-1;

    for((count=0, count <= end); count <= (end/2); ++count, --countback)
    {
        if(string[count]!=string[countback])
        {

```

```

        return(FALSE);
    }
}

return(TRUE);
}

```

54 - Write a C program to convert a decimal number into a binary number.

99% of the people who attend interviews can't answer this question, believe me!. Here is a C program which does this....

```

#include<stdio.h>

generatebits(int num)
{
    int temp;
    if(num)
    {
        temp = num % 2;
        bit(num >>= 1);
        printf("%d",temp);
    }
}

int main()
{
    int num;

    while(1)
    {
        scanf("Enter a number : %d",&num);
        printf("\n\n");
        generatebits(num);
    }

    getch();
    return(0);
}

```

55 - Write C code to implement the Binary Search algorithm.

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 $O(\log(n))$.

56- code to evaluate a polynomial.

```

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);
}

```

57 - Write code to add two polynomials

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;
}

// Add the remaining elements of the second polynomial to the result

while(p2 != h2)
{
    if(p2->flag==0)
    {
        h3=addNode(p2->cf, p2->px, p2->py, h3);
    }
    p2=p2->next;
}

return(h3);
}

```

58 - Write a program to add two long positive numbers (each represented by linked lists).

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);
}
```

59 - How do you compare floating point numbers?

This is Wrong!.

```
double a, b;
```



```

if(a == b)
{
    ...
}

```

The above code might not work always. That's because of the way floating point numbers are stored.

A good way of comparing two floating point numbers is to have an accuracy threshold which is relative to the magnitude of the two floating point numbers being compared.

```

#include <math.h>
if(fabs(a - b) <= accuracy_threshold * fabs(a))

```

There is a lot of material on the net to know how floating point numbers can be compared. Go 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);
}

```

60 - What's a good way to implement complex numbers in C?

Use structures and some routines to manipulate them.

61 - How can I display a percentage-done indication on the screen?

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.

62 - Write a program to check if a given year is a leap year or not?

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))

```

63 - Is there something we can do in C but not in C++?

I have a really funny answer

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++!

64 - How to swap the two nibbles in a byte ?

Try this

```
#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;
}
```

65 - How to scan a string till we hit a new line using scanf()?

Use

```
scanf("%[^\n]", address);
```

66 - Write pseudocode to compare versions (like 115.10.1 vs 115.11.5).

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;

```

67 - How do you get the line numbers in C?

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
***

```

68 - How to fast multiply a number by 7?

Try

```
(num<<3 - num)
```

This is same as

```
num*8 - num = num * (8-1) = num * 7
```

69 - Write a simple piece of code to split a string at equal intervals

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);
}
```

70 - Is there a way to multiply matrices in lesser than $O(n^3)$ time complexity?

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 $N \times N$: for example $A \times B = C$

```
[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

$$2^{\log 7} = 2^{2.807}$$

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!

71 - How do you find out if a machine is 32 bit or 64 bit?

72 - Write a program to have the output go two places at once (to the screen and to a file also)

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!

73 - Write code to round numbers

Use something like

```
(int)(num < 0 ? (num - 0.5) : (num + 0.5))
```

74 - How can we sum the digits of a given number in single statement?

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);
}
```

75 - Given two strings A and B, how would you find out if the characters in B were a subset of the characters in A?

```
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(FALSE);

    return(TRUE);
}
```

76 - 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.

77 - Write a program to check if the stack grows up or down

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");
}

```

78 - How to add two numbers without using the plus operator?

```

SUM    = A XOR B
CARRY = A AND B

```

On a wicked note, you can add two numbers without using the + operator as follows

$a - (-b)$

79 - How to generate prime numbers? How to generate the next prime after a given prime?



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 number by all divisors up to the square root of that number. If no factors are found, it's 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 it's 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, so we circle it and cross all its multiples. 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.

80 - Write a program to print numbers from 1 to 100 without using loops!

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;
}
```

81 - Write your own trim() or squeeze() function to remove the spaces from a string.

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...

82 - Write your own random number generator function in C.

83 - Write your own sqrt() function in C

Trees

84 - Write a C program to find the depth or height of a tree.

Here is some C code to get the height of the tree

```
tree_height(mynode *p)
{
    if (p==NULL) return (0);
    if (p->left) {h1=tree_height(p->left);}
    if (p->right) {h2=tree_height(p->right);}
    return (max(h1,h2)+1);
}
```

The degree of the leaf is zero. The degree of a tree is the max of its element degrees. A binary tree of height n , $n > 0$, has at least n and at most $(2^n - 1)$ elements in it. The height of a binary tree that contains n , $n > 0$, elements is at most n and at least $\log(n+1)$ to the base 2.

$\log(n+1)$ to the base 2 = h

$n = (2^h - 1)$

85 - Write a C program to determine the number of elements (or size) in a tree.

```
int tree_size(struct node* node)
{
    if (node==NULL)
    {
        return(0);
    }
    else
    {
        return(tree_size(node->left) + tree_size(node->right) + 1);
    }
}
```

86 - Write a C program to delete a tree (i.e, free up its nodes)

Free up the nodes using Postorder traversal!.

87 - Write C code to determine if two trees are identical

```
int identical(struct node* a, struct node* b)
{
    if (a==NULL && b==NULL){return(true);}
    else if (a!=NULL && b!=NULL)
    {
        return(a->data == b->data &&
               identical(a->left, b->left) &&
               identical(a->right, b->right));
    }
    else return(false);
}
```

88 - Write a C program to find the minimum value in a binary search tree.

Here is some sample C code. The idea is to keep on moving till you hit the left most node in the tree

```
int minValue(struct node* node)
{
    struct node* current = node;

    while (current->left != NULL)
    {
        current = current->left;
    }

    return(current->data);
}
```

On similar lines, to find the maximum value, keep on moving till you hit the right most node of the tree.

89 - Write a C program to compute the maximum depth in a tree?

```
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);
    }
}

```

90 - Write a C program to create a mirror copy of a tree (left nodes become right and right nodes become left)!

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 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;
    }
}

```

91 - Write C code to return a pointer to the nth node of an inorder traversal of a BST.

Here is a C program. Study it carefully, it has some Gotchas!

```

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 POIN

```

```

TER! */);

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 rank in 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.

92 - Write C code to implement the preorder(), inorder() and postorder() traversals. Whats their time complexities?

Here are the C program snippets to implement these traversals...

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 $O(n)$.

93 - Write a C program to create a copy of a tree

Here is a C program which does that...

```

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);
}

```

94 - Write C code to check if a given binary tree is a binary search tree or not?

Here is a C program which checks if a given tree is a Binary Search Tree or not...

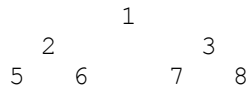
```

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);
}

```

95 - Write C code to implement level order traversal of a tree.

If this is the tree,



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
}
```

Pseudocode

```
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\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++];
}
}

```

96 - Write a C program to delete a node from a Binary Search Tree?

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);
}

```

97 - Write C code to search for a value in a binary search tree (BST).

```

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);}
}

```

98 - Write C code to count the number of leaves in a tree

```

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);
    }
}

```

```

    }
}

```

99 - Write C code for iterative preorder, inorder and postorder tree traversals

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]

```


100 - Can you construct a tree using postorder and preorder traversal?

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.

**101 - Construct a tree given its inorder and preorder traversal strings.
Similarly construct a tree given its inorder and post order traversal strings.**

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 her
e
        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 separate 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");
}

```

102 - Find the closest ancestor of two nodes in a tree.

```

#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\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]

103 - Given an expression tree, evaluate the expression and obtain a parenthesized form of the expression.

The code below prints the parenthesized form of a tree.

```
infix_exp(p)
{
    if(p)
    {
        printf("(");
        infix_exp(p->left);
        printf(p->data);
        infix_exp(p->right);
        printf(")");
    }
}
```

Creating a binary tree for a postfix expression

```
mynode *create_tree(char postfix[])
{
    mynode *temp, *st[100];
    int i,k;
    char symbol;

    for(i=k=0; (symbol = postfix[i])!='\0'; i++)
```

```

{
    temp = (mynode *) malloc(sizeof(struct node));
    temp->value = symbol;
    temp->left = temp->right = NULL;

    if(isalnum(symbol))
        st[k++] = temp;
    else
    {
        temp->right = st[--k];
        temp->left  = st[--k];
        st[k++]    = temp;
    }
}
return(st[--k]);
}

```

Evaluate a tree

```

float eval(mynode *root)
{
    float num;

    switch(root->value)
    {
        case '+' : return(eval(root->left) + eval(root->right)); break;
        case '-' : return(eval(root->left) - eval(root->right)); break;
        case '/' : return(eval(root->left) / eval(root->right)); break;
        case '*' : return(eval(root->left) * eval(root->right)); break;
        case '$' : return(eval(root->left) $ eval(root->right)); break;
        default  : if(isalpha(root->value))
                    {
                        printf("%c = ", root->value);
                        scanf("%f", &num);
                        return(num);
                    }
                    else
                    {
                        return(root->value - '0');
                    }
    }
}

```

104 - How do you convert a tree into an array?

The conversion is based on these rules

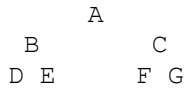
If $i > 1$, $i/2$ is the parent

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



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.

105- What is an AVL tree?

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.

Here is some recursive C code that sets the Balance factor for all nodes starting from the root....

```

setbf(mynode *p)
{
    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.
    p->bf = tempr - templ;

    return(count);
}

```

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 insertion 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 0    0 (tmp)
 /  \ /  \ /  \
0   0 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
  /      \
 0         0 (tmp)
 /  \      /  \
0    0    0    0
 /  \    /  \
0    0  (p)  (b)
 /  \
0    0
 /  \
0    0
 /  \
0    0

```

Right rotation

BEFORE

```

      0 (par)
    /  \
   0    0 (p)
  /      \
 0         0 (tmp)
 /  \      /  \
0    0    0    0
 /  \    /  \
0    0  (a)  (b)

```

Here we right rotate the tree around node p

```

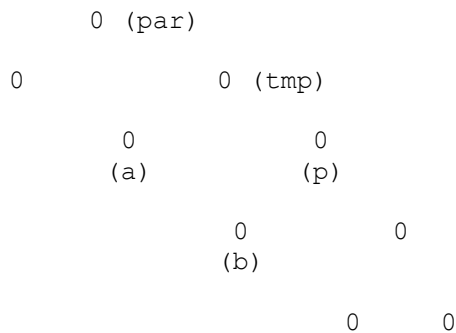
tmp          = p->left;
p->left      = tmp->right;
tmp->right   = p;

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



106 - How many different trees can be constructed using n nodes?

Good question!

Its

$$2^n - n$$

So, if there are 10 nodes, you will have $(1024 - 10) = 1014$ different trees!! Confirm it yourself with a small number if you dont believe the formula.

107 - A full N-ary tree has M non-leaf nodes, how many leaf nodes does it have?

Use Geometric progression.

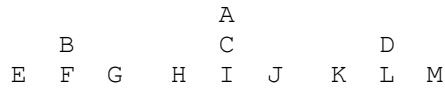
$$M + (N \wedge (n-1)) = (1 - (N \wedge n)) / (1 - N)$$

Here (N^{n-1}) is the number of leaf-nodes.

Solving for this leads to the answer

Leaf nodes = $M * (N - 1) + 1$

Suppose you have a 3-ary tree



So, here $M=4$ and $N=3$. So using the formula above,

Leaf nodes = $M * (N - 1) + 1 = 4 * (3 - 1) + 1 = 9$

108 – Implement Breadth First Search (BFS) and Depth First Search (DFS)

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 mark the vertices, 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++];
}
}

```

109 - Write pseudocode to add a new node to a Binary Search Tree (BST)

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);
    }
}

```

Bit Fiddling

111 - Write a C program to count bits set in an integer?

This is one of the most frequently asked interview questions of all times...

There are a number of ways to count the number of bits set in an integer. Here are some C programs to do the same.

Method1

This is the most basic way of doing it.

```

#include<stdio.h>
int main()
{
    unsigned int num=10;
    int ctr=0;

    for(;num!=0;num>>=1)
    {
        if(num&1)
        {
            ctr++;
        }
    }

    printf("\n Number of bits set in [%d] = [%d]\n", num, ctr);
    return(0);
}

```

Method2

This is a faster way of doing the same thing. Here the control goes into the while loop only as many times as the number of bits set to 1 in the integer!.

```

#include<stdio.h>
int main()
{
    int num=10;
    int ctr=0;

    while(num)
    {
        ctr++;
        num = num & (num -
1); // This clears the least significant bit set.
    }
}

```

```

    printf("\n Number of bits set in [%d] = [%d]\n", num, ctr);
    return(0);
}

```

Method3

This method is very popular because it uses a lookup table. This speeds up the computation. What it does is it keeps a table which hardcodes the number of bits set in each integer from 0 to 256.

For example

```

0 - 0 Bit(s) set.
1 - 1 Bit(s) set.
2 - 1 Bit(s) set.
3 - 2 Bit(s) set.
...

```

So here is the code...

```

const unsigned char LookupTable[] = { 0, 1, 1, 2, 1, 2, 2, 3, 1
, 2, 2, 3, 2, 3, 3, 4,
, 3, 3, 4, 3, 4, 4, 5,
, 3, 3, 4, 3, 4, 4, 5,
, 4, 4, 5, 4, 5, 5, 6,
, 3, 3, 4, 3, 4, 4, 5,
, 4, 4, 5, 4, 5, 5, 6,
, 4, 4, 5, 4, 5, 5, 6,
, 5, 5, 6, 5, 6, 6, 7,
, 3, 3, 4, 3, 4, 4, 5,
, 4, 4, 5, 4, 5, 5, 6,
, 4, 4, 5, 4, 5, 5, 6,
, 5, 5, 6, 5, 6, 6, 7,
, 4, 4, 5, 4, 5, 5, 6,
, 5, 5, 6, 5, 6, 6, 7,
, 5, 5, 6, 5, 6, 6, 7,
, 6, 6, 7, 6, 7, 7, 8};

```

```

unsigned int num;
unsigned int ctr;    // Number of bits set.

ctr = LookupTable[num & 0xff] +
      LookupTable[(num >> 8) & 0xff] +
      LookupTable[(num >> 16) & 0xff] +
      LookupTable[num >> 24];

```

112 - What purpose do the bitwise and, or, xor and the shift operators serve?

The AND operator

```

Truth Table
-----
0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

x AND 0 = 0
x AND 1 = x

```

We use bitwise "and" to test if certain bit(s) are one or not. And'ing a value against a pattern with ones only in the bit positions you are interested in will give zero if none of them are on, nonzero if one or more is on. We can also use bitwise "and" to turn off (set to zero) any desired bit(s). If you "and" a pattern against a variable, bit positions in the pattern that are ones will leave the target bit unchanged, and bit positions in the pattern that are zeros will set the target bit to zero.

The OR operator

```

Truth Table
-----
0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

x OR 0 = x
x OR 1 = 1

```

Use bitwise "or" to turn on (set to one) desired bit(s).

The XOR operator

```

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

```

```
x XOR 0 = x
x XOR 1 = ~x
```

Use bitwise "exclusive or" to flip or reverse the setting of desired bit(s) (make it a one if it was zero or make it zero if it was one).

The `>>`, `<<`, `>>=`, `<<=` operators

Operators `>>` and `<<` can be used to shift the bits of an operand to the right or left a desired number of positions. The number of positions to be shifted can be specified as a constant, in a variable or as an expression. Bits shifted out are lost. For left shifts, bit positions vacated by shifting always filled with zeros. For right shifts, bit positions vacated by shifting filled with zeros for unsigned data type and with copy of the highest (sign) bit for signed data type. The right shift operator can be used to achieve quick multiplication by a power of 2. Similarly the right shift operator can be used to do a quick division by power of 2 (unsigned types only). The operators `>>` and `<<`, dont change the operand at all. However, the operators `>>=` and `<<=` also change the operand after doing the shift operations.

```
x << y -
  Gives value x shifted left y bits (Bits positions vacated by shift
  are filled with zeros).
x <<= y -
  Shifts variable x left y bits (Bits positions vacated by shift are
  filled with zeros).
```

A left shift of n bits multiplies by 2 raise to n.

```
x >> y - Gives value x shifted right y bits.
x >>= y - Shifts variable x right y bits.
```

For the right shift, All bits of operand participate in the shift. For unsigned data type, bits positions vacated by shift are filled with zeros. For signed data type, bits positions vacated by shift are filled with the original highest bit (sign bit). Right shifting n bits divides by 2 raise to n. Shifting signed values may fail because for negative values the result never gets past -1:

-5 >> 3 is -1 and not 0 like -5/8.

Good interview question

A simple C command line utility takes a series of command line options. The options are given to the utility like this : <utility_name>

options=[no]option1,[no]options2,[no]option3?... Write C code using bitwise operators to use these flags in the code.

```
//Each option will have a bit reserved in the global_options_bits in
//eger. The global_options_bits
// integer will have a bit set or not set depending on how the optio
// n was specified by the user.
// For example, if the user said nooption1, the bit for OPTION1 in g
//lobal_options_bits
// will be 0. Likewise, if the user specified option3, the bit for O
//PTION3 in global_options_bits
// will be set to 1.

#define OPTION1 "option1" // For strcmp() with the passed arguments.
#define OPTION1_BITPOS (0x00000001) // Bit reserved for this option.
#define OPTION2 "option2"
#define OPTION2_BITPOS (0x00000002)
#define OPTION3 "option3"
#define OPTION3_BITPOS (0x00000004)

//Required to do the bit operations.
#define ALL_BITPOS (0x0001ffff)

// Assume you have already parsed the command line option and that
// parsed_argument_without_no has option1 or option2 or option3 (dep
//ending on what has
// been provided at the command line) and the variable negate_argume
//nt says if the
// option was negated or not (i.e, if it was option1 or nooption1)

if (strcmp((char *) parsed_argument_without_no, (char *) OPTION1) ==
    0)
{
    // Copy the global_options_bits to a temporary variable.
    tmp_action= global_options_bits;

    if (negate_argument)
    {
        // Setting the bit for this particular option to 0 as the op
        //tion has
        // been negated.
        action_mask= ~(OPTION1_BITPOS) & ALL_BITPOS;
        tmp_action= tmp_action & action_mask;
    }
    else
    {
        //Setting the bit for this particular option to 1.
        action_mask= (OPTION1_BITPOS);
        tmp_action= tmp_action | action_mask;
    }

    // Copy back the final bits to the global_options_bits variable
```



```

        global_options_bits= tmp_action;
    }
    else if (strcmp((char *) parsed_argument_without_no, (char *) OPTION
2) == 0)
    {
        //Similar code for OPTION2
    }
    else if (strcmp((char *) parsed_argument_without_no, (char *) OPTION
3) == 0)
    {
        //Similar code for OPTION3
    }

//Now someone who wishes to check if a particular option was set or
not can use the
// following type of code anywhere else in the code.
if(((global_options_bits & OPTION1_BITPOS) == OPTION1_BITPOS)
{
    //Do processing for the case where OPTION1 was active.
}
else
{
    //Do processing for the case where OPTION1 was NOT active.
}

```

113 - How to reverse the bits in an interger?

Here are some ways to reverse the bits in an integer.

Method1

```

unsigned int num;                // Reverse the bits in this number.
unsigned int temp = num;        // temp will have the reversed bits of
num.

int i;

for (i = (sizeof(num)*8-1); i; i--)
{
    temp = temp | (num & 1);
    temp <<= 1;
    num >>= 1;
}

temp = temp | (num & 1);

```

Method2

In this method, we use a lookup table.

114 - Check if the 20th bit of a 32 bit integer is on or off? AND it with x00001000 and check if its equal to x00001000

```

if((num & x00001000)==x00001000)

```

Note that the digits represented here are in hex.

0	0	0	0	1	0	0	0
				^			
x0000	0000	0000	0000	0001	0000	0000	0000 = 32 bits
^				^		^	
0th bit				20th bit		32nd bit	

115 - How to reverse the odd bits of an integer?

XOR each of its 4 hex parts with 0101.

116 - How would you count the number of bits set in a floating point number?

117 - What is heap sort?

A Heap is an almost complete binary tree. In this tree, if the maximum level is i , then, upto the $(i-1)$ th level should be complete. At level i , the number of nodes can be less than or equal to 2^i . If the number of nodes is less than 2^i , then the nodes in that level should be completely filled, only from left to right.

The property of an ascending heap is that, the root is the lowest and given any other node i , that node should be less than its left child and its right child. In a descending heap, the root should be the highest and given any other node i , that node should be greater than its left child and right child.

To sort the elements, one should create the heap first. Once the heap is created, the root has the highest value. Now we need to sort the elements in ascending order. The root can not be exchanged with the n th element so that the item in the n th position is sorted. Now, sort the remaining $(n-1)$ elements. This can be achieved by reconstructing the heap for $(n-1)$ elements.

A highly simplified pseudocode is given below

```

heapsort()
{
    n = array();    // Convert the tree into an array.
    makeheap(n);    // Construct the initial heap.

    for(i=n; i>=2; i--)
    {
        swap(s[1], s[i]);
        heapsize--;
        keepheap(i);
    }
}

```

```

}

makeheap(n)
{
    heapsize=n;
    for(i=n/2; i>=1; i--)
        keepheap(i);
}

keepheap(i)
{
    l = 2*i;
    r = 2*i + 1;

    p = s[l];
    q = s[r];
    t = s[i];

    if(l<=heapsize && p->value > t->value)
        largest = l;
    else
        largest = i;

    m = s[largest];

    if(r<=heapsize && q->value > m->value)
        largest = r;

    if(largest != i)
    {
        swap(s[i], s[largest]);
        keepheap(largest);
    }
}

```

118 - What is the difference between Merge Sort and Quick sort?

Both Merge-sort and Quick-sort have same time complexity i.e. $O(n \log n)$. In merge sort the file $a[1:n]$ was divided at its midpoint into sub-arrays which are independently sorted and later merged. Whereas, in quick sort the division into two sub-arrays is made so that the sorted sub-arrays do not need to be merged latter.

119 - Give pseudocode for the mergesort algorithm

```

Mergesort(a, left, right)
{
    if(left<right)
    {
        I=(left+right)/2;
        Mergesort(a, left, I);
        Mergesort(a, I+1, right);
        Merge(a, b, left, I, right);
        Copy(b, a, left, right);
    }
}

```

The merge would be something like this

```

merge(int a[], int b[], int c[], int m, int n)
{
    int i, j, k;
    i = 0;
    j = 0;
    k = 0;

    while(i<=m && j<=n)
    {
        if(a[i] < a[j])
        {
            c[k]=a[i];
            i=i+1;
        }
        else
        {
            c[k]=a[j];
            j=j+1;
        }
        k=k+1;
    }

    while(i<=m)
        c[k++]=a[i++];

    while(j<=n)
        c[k++]=a[j++];
}

```

120 - Implement the bubble sort algorithm. How can it be improved? Write the code for selection sort, quick sort, insertion sort.

C Pointers

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

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)++.

122 - What is a NULL pointer? How is it different from an uninitialized pointer? How is a NULL pointer defined?

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 ("").

123 - What is a null pointer assignment error?

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

124 - 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?

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, what's 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.

125 - Is the cast to `malloc()` required at all?

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.

126 - 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?

`malloc()` is used to allocate memory. It's 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 returning 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. It's 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!

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

```
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 itse  
lf.
```

```
    *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 poin  
ter.
```



```
*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 its 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...

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

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

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

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 multi-byte number at an odd address).
- Accessing a memory location outside its address space.
- Accessing a physical address that does not correspond to any device.
- Out-of-bounds array references.
- References through uninitialized or mangled 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;
}

```

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

This is an array of pointers

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

This is a pointer to a 10 element array

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

131 - What is a memory leak?

Its an scenario where the program has lost a reference to an area in the memory. Its a programming term describing the loss of memory. This happens when the program allocates some memory but fails to return it to the system

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

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 pieces 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. This is done to avoid fragmentation.

realloc() can be used only with a preallocated/malloced/reallocated 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.

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

A pointer which points to an object that no longer exists. It's 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.

134 - What do pointers contain?

Since addresses are always whole numbers, pointers always contain whole numbers.

C Functions

141 - How to declare a pointer to a function?

Use something like this

```
int myfunc(); // The function.
int (*fp)(); // Pointer to that function.

fp = myfunc; // Assigning the address of the function to the pointer.

(*fp)(); // Calling the function.
fp(); // Another way to call the function.
```

So then, what are function pointers, huh?

In simple words, a function pointer is a pointer variable which holds the address of a function and can be used to invoke that function indirectly. Function pointers are useful when you want to invoke separate functions based on different scenarios. In that case, you just pass in the pointer to the function you want to be invoked. Function pointers are used extensively when writing code with call back scenarios.

For example, when writing code involving a XML parser, you pass in the pointers to your event handling functions and whenever there is a specific event, the XML parser calls your functions through their pointers. This function whose pointer is passed is generally called as the call back function.

Here is an implementation of a Generic linked list which uses function pointers really well...

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

typedef struct list {
    void *data;
    struct list *next;
} List;

struct check {
    int i;
    char c;
    double d;
} chk[] = { { 1, 'a', 1.1 },
            { 2, 'b', 2.2 },
            { 3, 'c', 3.3 } };

// See how the print() function takes in a pointer to a function!
void print(List *, void (*)(void *));

void insert(List **, void *, unsigned int);
void printstr(void *);
void printint(void *);
void printchar(void *);
void printcomp(void *);
```

```

List *list1, *list2, *list3, *list4;

int main(void)
{
    char c[]      = { 'a', 'b', 'c', 'd' };
    int i[]       = { 1, 2, 3, 4 };
    char *str[]  = { "hello1", "hello2", "hello3", "hello4" };

    list1 = list2 = list3 = list4 = NULL;

    insert(&list1, &c[0], sizeof(char));
    insert(&list1, &c[1], sizeof(char));
    insert(&list1, &c[2], sizeof(char));
    insert(&list1, &c[3], sizeof(char));

    insert(&list2, &i[0], sizeof(int));
    insert(&list2, &i[1], sizeof(int));
    insert(&list2, &i[2], sizeof(int));
    insert(&list2, &i[3], sizeof(int));

    insert(&list3, str[0], strlen(str[0])+1);
    insert(&list3, str[1], strlen(str[0])+1);
    insert(&list3, str[2], strlen(str[0])+1);
    insert(&list3, str[3], strlen(str[0])+1);

    insert(&list4, &chk[0], sizeof chk[0]);
    insert(&list4, &chk[1], sizeof chk[1]);
    insert(&list4, &chk[2], sizeof chk[2]);

    printf("Printing characters:");
    print(list1, printchar);
    printf(" : done\n\n");

    printf("Printing integers:");
    print(list2, printint);
    printf(" : done\n\n");

    printf("Printing strings:");
    print(list3, printstr);
    printf(" : done\n\n");

    printf("Printing composite:");
    print(list4, printcomp);
    printf(" : done\n");

    return 0;
}

void insert(List **p, void *data, unsigned int n)
{
    List *temp;
    int i;

    /* Error check is ignored */
    temp = malloc(sizeof(List));
    temp->data = malloc(n);
    for (i = 0; i < n; i++)
        *(char *) (temp->data + i) = *(char *) (data + i);
    temp->next = *p;
}

```

```

        *p = temp;
    }

void print(List *p, void (*f)(void *))
{
    while (p)
    {
        (*f)(p->data);
        p = p->next;
    }
}

void printstr(void *str)
{
    printf(" \"%s\"", (char *)str);
}

void printint(void *n)
{
    printf(" %d", *(int *)n);
}

void printchar(void *c)
{
    printf(" %c", *(char *)c);
}

void printcomp(void *comp)
{
    struct check temp = *(struct check *)comp;
    printf(" '%d:%c:%f", temp.i, temp.c, temp.d);
}

```

142 - Does extern in a function declaration mean anything?

The extern in a function's declaration is sometimes used to indicate that the function's definition is in some other source file, but there is no difference between

```
extern int function_name();
```

and

```
int function_name();
```

143 - How can I return multiple values from a function?

You can pass pointers to locations which the function being called can populate, or have the function return a structure containing the desired values, or use global variables.

144 - Does C support function overloading?

Function overload is not present in C. In C, either use different names or pass a union of supported types (with additional identifier that give hints of the type to be used).

Most people think its supported because they might unknowingly be using a C++ compiler to compile their C code and C++ does have function overloading.

145 - What is the purpose of a function prototype?

A function prototype tells the compiler to expect a given function to be used in a given way. That is, it tells the compiler the nature of the parameters passed to the function (the quantity, type and order) and the nature of the value returned by the function.

146 - What are inline functions?

Before reading the answer, please be aware that inline functions are non-standard C. They are provided as compiler extensions. But, nevertheless, one should know what they are used for.

The inline comment is a request to the compiler to copy the code into the object at every place the function is called. That is, the function is expanded at each point of call. Most of the advantage of inline functions comes from avoiding the overhead of calling an actual function. Such overhead includes saving registers, setting up stack frames, and so on. But with large functions the overhead becomes less important. Inlining tends to blow up the size of code, because the function is expanded at each point of call.

```
int myfunc(int a)
{
    ...
}

inline int myfunc(int a)
{
    ...
}
```

Inlined functions are not the fastest, but they are the kind of better than macros (which people use normally to write small functions).

```
#define myfunc(a)    \
{ \
    ...              \
}
```

The problem with macros is that the code is literally copied into the location it was called from. So if the user passes a "double" instead of an "int" then problems could occur. However, if this senerio happens with an inline function the compiler will complain about incompatible types. This will save you debugging time stage.

Good time to use inline functions is

1. There is a time criticle function.
2. That is called often.
3. Its small. Remember that inline functions take more space than norma l functions.

Some typical reasons why inlining is sometimes not done are:

1. The function calls itself, that is, is recursive.
2. The function contains loops such as `for(;;)` or `while()`.
3. The function size is too large.

147 - How to declare an array of N pointers to functions returning pointers to functions returning pointers to characters?

Declare it this way

```
char *(*(*a[N])())();
```

Even this seems to be correct

```
(char*(*fun_ptr)())(*func[n])();
```

148 - Can we declare a function that can return a pointer to a function of the same type?

We cannot do it directly. Either have the function return a generic function pointer, with casts to adjust the types as the pointers are passed around; or have it return a structure containing only a pointer to a function returning that structure.

149 - How can I write a function that takes a variable number of arguments? What are the limitations with this? What is `vprintf()`?

The header `stdarg.h` provides this functionality. All functions like `printf()`, `scanf()` etc use this functionality.

The program below uses a `var_arg` type of function to count the overall length of strings passed to the function.

```
#include <stdarg.h>

int myfunction(char *first_argument,...)
{
    int length;
    va_list argp;
    va_start(argp, first);
    char *p;

    length = strlen(first_argument);
    while((p = va_arg(argp, char *)) != NULL)
    {
        length = length + strlen(p);
    }

    va_end(argp);
}
```

```

    return(length);
}

int main()
{
    int length;
    length = myfunction("Hello", "Hi", "Hey!", (char *)NULL);
    return(0);
}

```

How can I find how many arguments a function was passed?

Any function which takes a variable number of arguments must be able to determine from the arguments themselves, how many of them there have been passed. `printf()` and some similar functions achieve this by looking for the format string. This is also why these functions fail badly if the format string does not match the argument list. Another common technique, applicable when the arguments are all of the same type, is to use a sentinel value (often 0, -1, or an appropriately-cast null pointer) at the end of the list. Also, one can pass an explicit count of the number of variable arguments. Some older compilers did provided a `nargs()` function, but it was never portable.

Is this allowed?

```

int f(...)
{
    ...
}

```

No! Standard C requires at least one fixed argument, in part so that you can hand it to `va_start()`.

So how do I get floating point numbers passed as arguments?

Arguments of type `float` are always promoted to type `double`, and types `char` and `short int` are promoted to `int`. Therefore, it is never correct to invoke

```
va_arg(argp, float);
```

instead you should always use

```
va_arg(argp, double)
```

Similarly, use

```
va_arg(argp, int)
```

to retrieve arguments which were originally char, short, or int.

How can I create a function which takes a variable number of arguments and passes them to some other function (which takes a variable number of arguments)?

You should provide a version of that other function which accepts a `va_list` type of pointer.

So how can I call a function with an argument list built up at run time?

There is no portable way to do this. Instead of an actual argument list, you might want to pass an array of generic (`void *`) pointers. The called function can then step through the array, much like `main()` steps through `char *argv[]`.

What is the use of `vprintf()`, `fprintf()` and `vsprintf()`?

Below, the `myerror()` function prints an error message, preceded by the string "error: " and terminated with a newline:

```
#include <stdio.h>
#include <stdarg.h>
void myerror(char *fmt, ...)
{
    va_list argp;
    fprintf(stderr, "error: ");
    va_start(argp, fmt);
    fprintf(stderr, fmt, argp);
    va_end(argp);
    fprintf(stderr, "\n");
}
```

150 - With respect to function parameter passing, what is the difference between call-by-value and call-by-reference? Which method does C use?

In the case of call-by-reference, a pointer reference to a variable is passed into a function instead of the actual value. The function's operations will effect the variable in a global as well as local sense. Call-by-value (C's method of parameter passing), by contrast, passes a copy of the variable's value into the function. Any changes to the variable made by function have only a local effect and do not alter the state of the variable passed into the function.

Does C really have pass by reference?

No.

C uses pass by value, but it can pretend doing pass by reference, by having functions that have pointer arguments and by using the `&` operator when calling the function. This way, the compiler will simulate this feature (like when you pass an array to a function, it

actually passes a pointer instead). C does not have something like the formal pass by reference or C++ reference parameters.

How will you decide whether to use pass by reference, by pointer and by value?

The selection of the argument passing depends on the situation.

- A function uses passed data without modifying it:
 - If the data object is small, such as a built-in data type or a small structure then pass it by value.
 - If the data object is an array, use a pointer because that is the only choice. Make the pointer a pointer to const.
 - If the data object is a good-sized structure, use a const pointer or a const reference to increase program efficiency. You save the time and space needed to copy a structure or a class design, make the pointer or reference const.
 - If the data object is a class object, use a const reference. The semantics of class design often require using a reference. The standard way to pass class object arguments is by reference.
- A function modifies data in the calling function:
 - If the data object is a built-in data type, use a pointer. If you spot a code like `fixit(&x)`, where `x` is an int, its clear that this function intends to modify `x`.
 - If the data object is an array, use the only choice, a pointer.
 - If the data object is a structure, use a reference or a pointer.
 - If the data object is a class object, use a reference.

151 - If I have the name of a function in the form of a string, how can I invoke that function?

Keep a table of names and their function pointers:

```
int myfunc1(), myfunc2();

struct
{
    char *name;
    int (*func_ptr)();
} func_table[] = {"myfunc1", myfunc1,
                 "myfunc2", myfunc2,};
```

Search the table for the name, and call via the associated function pointer.

152 - What does the error, invalid redeclaration of a function mean?

If there is no declaration in scope then it is assumed to be declared as returning an int and

without any argument type information. This can lead to discrepancies if the function is later declared or defined. Such functions must be declared before they are called. Also check if there is another function in some header file with the same name.

153 - How can I pass the variable argument list passed to one function to another function.

Good question

Something like this wont work

```
#include <stdarg.h>
main()
{
    display("Hello", 4, 12, 13, 14, 44);
}

display(char *s,...)
{
    show(s,...);
}

show(char *t,...)
{
    va_list ptr;
    int a;
    va_start(ptr,t);
    a = va_arg(ptr, int);
    printf("%f", a);
}
```

This is the right way of doing it

```
#include <stdarg.h>
main()
{
    display("Hello", 4, 12, 13, 14, 44);
}

display(char *s,...)
{
    va_list ptr;
    va_start(ptr, s);
    show(s,ptr);
}

show(char *t, va_list ptr1)
{
    int a, n, i;
    a=va_arg(ptr1, int);

    for(i=0; i<a; i++)
    {
        n=va_arg(ptr1, int);
    }
}
```

```

        printf("\n%d", n);
    }
}

```

154 - How do I pass a variable number of function pointers to a variable argument (va_arg) function?

Glad that you thought about doing something like this!

Here is some code

```

#include <stdarg.h>

main()
{
    int (*p1)();
    int (*p2)();
    int fun1(), fun2();

    p1 = fun1;
    p2 = fun2;
    display("Bye", p1, p2);
}

display(char *s,...)
{
    int (*pp1)(), (*pp2)();
    va_list ptr;
    typedef int (*f)(); //This typedef is very important.

    va_start(ptr,s);

    pp1 = va_arg(ptr, f); // va_arg(ptr, int (*)()); would NOT have worked!
    pp2 = va_arg(ptr, f);

    (*pp1)();
    (*pp2)();
}

fun1()
{
    printf("\nHello!\n");
}

fun2()
{
    printf("\nHi!\n");
}

```

155 - Will C allow passing more or less arguments than required to a function.

It won't if the prototype is around. It will ideally scream out with an error like

Too many arguments

or

Too few arguments

But if the prototype is not around, the behavior is undefined.

Try this out

```
#include <stdio.h>

/*
int foo(int a);
int foo2(int a, int b);
*/

int main(int a)
{
    int (*fp)(int a);

    a = foo();
    a = foo2(1);

    exit(0);
}

int foo(int a)
{
    return(a);
}

int foo2(int a, int b)
{
    return(a+b);
}
```

156- Whats short-circuiting in C expressions?

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.

C Statements

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

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

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

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;
```

159 - Is `5[array]` the same as `array[5]`?

Yes!

Since array subscripting is commutative in C. That is

```
array[n] == *((array)+(n)) == *((n)+(array)) == n[array]
```

But frankly, this feature is of no use to anyone. Anyone asking this question is more or less a fool trying to be cool.

160 - What are `#pragmas`?

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).

161 - What is the difference between `if(0 == x)` and `if(x == 0)`?

Nothing!. But, it's a good trick to prevent the common error of writing

```
if(x = 0)
```

The error above is the source of a lot of serious bugs and is very difficult to catch. If you cultivate the habit of writing the constant before the `==`, the compiler will complain if you accidentally type

```
if(0 = x)
```

Of course, the trick only helps when comparing to a constant.

162 - Should we use `goto` or not?

You should use `gotos` wherever it suits your needs really well. There is nothing wrong in using them. Really.

There are cases where each function must have a single exit point. In these cases, it makes much sense to use `gotos`.

```

myfunction()
{
    if(error_condition1)
    {
        // Do some processing.
        goto failure;
    }

    if(error_condition2)
    {
        // Do some processing.
        goto failure;
    }

success:
    return(TRUE);

failure:

    // Do some cleanup.
    return(FALSE);
}

```

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

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.

164 - What do lvalue and rvalue mean?

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.

165 - What does the term cast refer to? Why is it used?

Casting is a mechanism built into C that allows the programmer to force the conversion of data types. This may be needed because most C functions are very particular about the data types they process. A programmer may wish to override the default way the C compiler promotes data types.

166 - What is the difference between a statement and a block?

A statement is a single C expression terminated with a semicolon. A block is a series of statements, the group of which is enclosed in curly-braces.

168 - What is type checking?

The process by which the C compiler ensures that functions and operators use data of the appropriate type(s). This form of check helps ensure the semantic correctness of the program.

169 - Why can't you nest structure definitions?

This is a trick question!

You can nest structure definitions.

```
struct salary
{
    char empname[20];
    struct
    {
        int dearness;
    }
    allowance;
}employee;
```

170 - What is a forward reference?

It is a reference to a variable or function before it is defined to the compiler. The cardinal rule of structured languages is that everything must be defined before it can be used.

There are rare occasions where this is not possible. It is possible (and sometimes necessary) to define two functions in terms of each other. One will obey the cardinal rule while the other will need a forward declaration of the former in order to know of the former's existence.

171 - What is the difference between the & and && operators and the | and || operators?

& and | are bitwise AND and OR operators respectively. They are usually used to manipulate the contents of a variable on the bit level. && and || are logical AND and OR operators respectively. They are usually used in conditionals

172 - an goto be used to jump across functions?

No!

This wont work

```
main()
{
    int i=1;
    while (i<=5)
    {
```

```

        printf("%d",i);
        if (i>2)
            goto here;
        i++;
    }
}

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

```

174 - Whats wrong with #define myptr int *?

```

#define myptr int *
myptr p, q;

```

Only p will be a pointer to an int, and q will just be an int!.

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

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 integer 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

```

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

No

The only things that can 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;

177 - What is more efficient? A switch() or an if() else()?

Both are equally efficient. Usually the compiler implements them using jump instructions. But each of them has their own unique advantages.

C Arrays

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

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]
}
```

180 - Is `char a[3] = "abc";` legal? What does it mean?

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.

181 - If `a` is an array, is `a++` valid?

No

You will get an error like

```
Error message : Lvalue required in function main.
```

Doing `a++` is asking the compiler to change the base address of the array. This the only thing that the compiler remembers about an array once its declared and it wont allow you to change the base address. If it allows, it would be unable to remember the beginning of the array.

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

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.

C Variables

183 - Do Global variables start out as zero?

Glad you asked!

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).

184 - Does C have boolean variable type?

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.

185 - Where may variables be defined in C?

Outside a function definition (global scope, from the point of definition downward in the source code). Inside a block before any statements other than variable declarations (local scope with respect to the block).

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

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 loses its contents and identity when the function in 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.

187 - What does the `typedef` keyword do?

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.

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

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 its value in the program - 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 what's 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 read-only 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.

189 - What are Trigraph characters?

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

```
??=    #
??(    [
??)    ]
??<    {
??>    }
??!    |
??/    \
```

??' ^
 ??- ~

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

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:

$$\begin{aligned} 5.00 &\times 100 \\ 0.05 &\times 10^2 \\ 5000 &\times 10^{-3} \end{aligned}$$

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.

C Structures

191 - Can structures be assigned to variables and passed to and from functions?

Yes, they can!

But note that when structures are passed, returned or assigned, the copying is done only at one level (The data pointed to by any pointer fields is not copied!).

192 - Can we directly compare two structures using the == operator?

No, you cannot!

The only way to compare two structures is to write your own function that compares the structures field by field. Also, the comparison should be only on fields that contain data (You would not want to compare the next fields of each structure!).

A byte by byte comparison (say using `memcmp()`) will also fail. This is because the comparison might fonder on random bits present in unused "holes" in the structure (padding used to keep the alignment of the later fields correct). So a `memcmp()` of the two structure will almost never work. Also, any strings inside the strucutres must be compared using `strcmp()` for similar reasons.

There is also a very good reason why structures can be compared directly - unions!. It is because of unions that structures cannot be compared for equality. The possibility that a structure might contain a union makes it hard to compare such structures; the compiler can't tell what the union currently contains and so wouldn't know how to compare the structures. This sounds a bit hard to swallow and isn't 100% true, most structures don't contain unions, but there is also a philosophical issue at stake about just what is meant by "equality" when applied to structures. Anyhow, the union business gives the Standard a good excuse to avoid the issue by not supporting structure comparison.

If your structures dont have stuff like floating point numbers, pointers, unions etc..., then you could possibly do a `memset()` before using the structure variables..

```
memset (&myStruct, 0, sizeof(myStruct));
```

This will set the whole structure (including the padding) to all-bits-zero. We can then do a `memcmp()` on two such structures.

```
memcmp (&s1, &s2, sizeof(s1));
```

But this is very risky and can end up being a major bug in your code!. So try not to do this kind of memcmp() operations on structures variables as far as possible!

193 - Can we pass constant values to functions which accept structure arguments?

If you are trying to do something like this

```
myfunction((struct mystruct){10,20});
```

then, it wont work!. Use a temporary structure variable.

194 - How does one use fread() and fwrite()? Can we read/write structures to/from files?

Its easy to write a structure into a file using fwrite()

```
fwrite(&somestruct, sizeof somestruct, 1, fp);
```

A similat fread() invocation can read it back in. But, data files so written will not be portable (specially if they contain floating point numbers). Also that if the structure has any pointers, only the pointer values will be written, and they are most unlikely to be valid when read back in. One must use the "rb/wb" flag when opening the files.

A more portable solution is to write code to write and read a structure, field-by-field, in a portable (perhaps ASCII) way!. This is simpler to port and maintain.

195- Why do structures get padded? Why does sizeof() return a larger size?

Padding enables the CPU to access the members faster. If they are not aligned (say to word boundaries), then accessing them might take up more time. So the padding results in faster access. This is also required to ensure that alignment properties are preserved when an array of contiguous structures is allocated. Even if the structure is not part of an array, the end padding remains, so that sizeof() can always return a consistent size.

196 - Can we determine the offset of a field within a structure and directly access that element?

The offsetof() macro(<stddef.h>) does exactly this.

A probable implementation of the macro is

```
#define offsetof(type, mem) ((size_t)((char *)&((type *)0)->mem -  
    (char *) (type *)0))
```

This can be used as follows. The offset of field a in struct mystruct is

```
offset_of_a = offsetof(struct mystruct, a)
```

If structpointer is a pointer to an instance of this structure, and field a is an int, its value

can be set indirectly with

```
*(int *)((char *)structpointer + offset_of_a) = some_value;
```

197 - What are bit fields in structures?

To avoid wastage of memory in structures, a group of bits can be packed together into an integer and its called a bit field.

```
struct tag-name
{
    data-type name1:bit-length;
    data-type name2:bit-length;
    ...
    ...
    data-type nameN:bit-length;
}
```

A real example

```
struct student;
{
    char name[30];
    unsigned sex:1;
    unsigned age:5;
    unsigned rollno:7;
    unsigned branch:2;
};

struct student a[100];

scanf("%d", &sex);
a[i].sex=sex;
```

There are some limitations with respect to these bit fields, however:

1. Cannot scanf() directly into bit fields.
2. Pointers cannot be used on bit fields to access them.
3. Cannot have an array of bit fields.

The main use of bitfields is either to allow tight packing of data or to be able to specify the fields within some externally produced data files. C gives no guarantee of the ordering of fields within machine words, so if you do use them for the latter reason, your program will not only be non-portable, it will be compiler-dependent too. The Standard says that fields are packed into 'storage units', which are typically machine words. The packing order, and whether or not a bitfield may cross a storage unit boundary, are implementation defined. To force alignment to a storage unit boundary, a zero width field is used before the one that you want to have aligned. Be careful using them. It can require a surprising amount of run-time code to manipulate these things and you can end up

using more space than they save. Bit fields do not have addresses?you can't have pointers to them or arrays of them.

198 - What is a union? Where does one use unions? What are the limitations of unions?

A union is a variable type that can contain many different variables (like a structure), but only actually holds one of them at a time (not like a structure). This can save memory if you have a group of data where only one of the types is used at a time. The size of a union is equal to the size of it's largest data member. The C compiler allocates just enough space for the largest member. This is because only one member can be used at a time, so the size of the largest, is the most you will need. Here is an example:

```
union person
{
    int age;
    char name[100];
}person1;
```

The union above could be used to either store the age or it could be used to hold the name of the person. There are cases when you would want one or the other, but not both (This is a bad example, but you get the point). To access the fields of a union, use the dot operator(.) just as you would for a structure. When a value is assigned to one member, the other member(s) get whipped out since they share the same memory. Using the example above, the precise time can be accessed like this:

```
person1.age;
```

In larger programs it may be difficult to keep track of which field is the currently used field. This is usually handled by using another variable to keep track of that. For example, you might use an integer called field. When field equals one, the age is used. If field is two, then name is used. The C compiler does no more than work out what the biggest member in a union can be and allocates enough storage (appropriately aligned if neccessary). In particular, no checking is done to make sure that the right sort of use is made of the members. That is your task, and you'll soon find out if you get it wrong. The members of a union all start at the same address?there is guaranteed to be no padding in front of any of them.

ANSI Standard C allows an initializer for the first member of a union. There is no standard way of initializing any other member (nor, under a pre-ANSI compiler, is there generally any way of initializing a union at all).

It is because of unions that structures cannot be compared for equality. The possibility that a structure might contain a union makes it hard to compare such structures; the compiler can't tell what the union currently contains and so wouldn't know how to compare the structures. This sounds a bit hard to swallow and isn't 100% true, most structures don't contain unions, but there is also a philosophical issue at stake about just what is meant by "equality" when applied to structures. Anyhow, the union business

gives the Standard a good excuse to avoid the issue by not supporting structure comparison.

199 - How should we write a multi-statement macro?

This is the correct way to write a multi statement macro.

```
#define MYMACRO(argument1, argument2) do { \
    statement1; \
    statement2; \
} while(1) /* no trailing semicolon */
```

200 - How can I write a macro which takes a variable number of arguments?

One can define the macro with a single, parenthesized "argument" which in the macro expansion becomes the entire argument list, parentheses and all, for a function such as printf():

```
#define DEBUG(args) (printf("DEBUG: "), printf args)

if(n != 0) DEBUG(("n is %d\n", n));
```

The caller must always remember to use the extra parentheses. Other possible solutions are to use different macros depending on the number of arguments. C9X will introduce formal support for function-like macros with variable-length argument lists. The notation ... will appear at the end of the macro "prototype" (just as it does for varargs functions).

201- What is the token pasting operator and stringizing operator in C?

Token pasting operator

ANSI has introduced a well-defined token-pasting operator, ##, which can be used like this:

```
#define f(g,g2) g##g2

main()
{
    int var12=100;
    printf("%d",f(var,12));
}
```

O/P

100

Stringizing operator

```
#define sum(xy) printf(#xy " = %f\n",xy);

main()
{
    sum(a+b); // As good as printf("a+b = %f\n", a+b);
}
```

So what does the message "warning: macro replacement within a string literal" mean?

```
#define TRACE(var, fmt) printf("TRACE: var = fmt\n", var)
TRACE(i, %d);
```

gets expanded as

```
printf("TRACE: i = %d\n", i);
```

In other words, macro parameters were expanded even inside string literals and character constants. Macro expansion is *not* defined in this way by K&R or by Standard C.

When you do want to turn macro arguments into strings, you can use the new # preprocessing operator, along with string literal concatenation:

```
#define TRACE(var, fmt) printf("TRACE: " #var " = " #fmt "\n", var)
```

See and try to understand this special application of the stringizing operator!

```
#define Str(x) #x
#define Xstr(x) Str(x)
#define OP plus
```

```
char *opname = Xstr(OP); //This code sets opname to "plus" rather than "OP".
```

Here are some more examples

Example1

Define a macro DEBUG such that the following program

```
int main()
{
    int x=4;
    float a = 3.14;
    char ch = 'A';

    DEBUG(x, %d);
    DEBUG(a, %f);
    DEBUG(ch, %c);
}
```

outputs

```
DEBUG: x=4
DEBUG: y=3.140000
DEBUG: ch=A
```

The macro would be

```
#define DEBUG(var, fmt) printf("DEBUG:" #var "=" #fmt "\n", var);
```

Example2

Write a macro PRINT for the following program

```
int main()
{
    int x=4, y=4, z=5;
    int a=1, b=2, c=3;
    PRINT(x,y,z);
    PRINT(a,b,c);
}
```

such that it outputs

```
x=4 y=4 z=5
a=1 b=2 c=3
```

Here is a macro that will do this

```
#define PRINT(v1,v2,v3) printf("\n" #v1 "=%d" #v2 "=%d" #v3 "=%d", v1,
v2, v3);
```

202 - Define a macro called SQR which squares a number.

This is the wrong way of doing it.

```
#define SQR(x) x*x
```

Thats because, something like

```
#include <stdio.h>
#define SQR(x) x*x

int main()
{
    int i;
    i = 64/SQR(4);
    printf("[%d]", i);
    return(0);
}
```

will produce an output of 64, because of how macros work (The macro call square(4) will substituted by 4*4 so the expression becomes $i = 64/4*4$. Since / and * has equal priority the expression will be evaluated as $(64/4)*4$ i.e. $16*4 = 64$).

This is the right way of doing it.

```
#define SQR(x) ((x)*(x))
```

But, if x is an expression with side effects, the macro evaluates them twice. The following is one way to do it while evaluating x only once.

```
#include <math.h>
#define SQR(x) pow((x),2.0)
```

203 - What should go in header files? How to prevent a header file being included twice? Whats wrong with including more headers?

Generally, a header (.h) file should have (A header file need not have a .h extension, its just a convention):

1. Macro definitions (preprocessor #defines).
2. Structure, union, and enumeration declarations.
3. Any typedef declarations.
4. External function declarations.
5. Global variable declarations.

Put declarations / definitions in header files if they will be shared between several other files. If some of the definitions / declarations should be kept private to some .c file, don;t add it to the header files.

How does one prevent a header file from included twice?. Including header files twice can lead to multiple-definition errors.

There are two methods to prevent a header file from included twice

Method1

```
#ifndef HEADER_FILE
#define HEADER_FILE
...header file contents...
#endif
```

Method2

A line like

```
#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).

So, what's the difference between #include <> and #include "" ?

The <> syntax is typically used with Standard or system-supplied headers, while "" is

typically used for a program's own header files. Headers named with <> syntax are searched for in one or more standard places. Header files named with font class=announcelink>"" syntax are first searched for in the "current directory," then (if not found) in the same standard places.

What happens if you include unwanted headers?

You will end up increasing the size of your executables!

204 - Is there a limit on the number of characters in the name of a header file?

The limitation is only that identifiers be significant in the first six characters, not that they be restricted to six characters in length.

205 - How do stat(), fstat(), vstat() work? How to check whether a file exists?

The functions stat(), fstat(), lstat() are used to get the file status.

Here are their prototypes

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *file_name, struct stat *buf);
int fstat(int file_desc, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);
```

These functions return information about the specified file. You do not need any access rights to the file to get this information but you need search rights to all directories named in the path leading to the file.

```
stat      -
           stats the file pointed to by file_name and fills in buf.
lstat     -
           identical to stat, except in the case of a symbolic link,
           where the link itself is stat-
ed, not the file that it refers to.
fstat     -
           identical to stat, only the open file pointed to by file_desc
           is stated in place of file_name.
```

They all return a stat structure, which contains the following fields:

```
struct stat {
    dev_t      st_dev;      /* device */
    ino_t      st_ino;      /* inode */
    mode_t     st_mode;     /* protection */
    nlink_t    st_nlink;    /* number of hard links */
    uid_t      st_uid;      /* user ID of owner */
    gid_t      st_gid;      /* group ID of owner */
    dev_t      st_rdev;     /* device type (if inode d
evice) */
```

```

        off_t          st_size;      /* total size, in bytes */
        unsigned long st_blksize;    /* blocksize for filesystem
m I/O */

        unsigned long st_blocks;    /* number of blocks allocated */
        ted */

        time_t         st_atime;     /* time of last access */
        time_t         st_mtime;     /* time of last modification
on */

        time_t         st_ctime;     /* time of last change */
};

```

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

Here is a small piece of code which returns the size of a file by accessing the st_size member of the stat structure.

```

boolean get_file_size(char *file_path,
                      int *file_size)
{
    struct stat stat_buffer;

    if (stat((char *)file_path, &stat_buffer) != 0)
        return(FALSE);

    *file_size = stat_buffer.st_size;
    return(TRUE);
}

```

So how do we check if a file exists or not?

Use functions like stat() as used above to find out if a file exists or not. Also, one can use fopen(). When using fopen(), just open for reading and close immediately. If the file does not exist, fopen() will give an error.

206 - How can I insert or delete a line (or record) in the middle of a file?

The only way is to rewrite the file.

207 - How can I recover the file name using its file descriptor?

Have a wrapper around fopen() to remember the names of files as you open them.

208 - How can I delete a file? How do I copy files? How can I read a directory in a C program?

Deleting a file

The Standard C Library function is remove(). If that's not there, use unlink().

Copying a file

Directly use `system()` to invoke the operating system's `copy()` utility, or open the source and destination files, read characters or blocks of characters from the source file, and write them to the destination file.

How to read directories?

Use the `opendir()` and `readdir()` functions, which are part of the POSIX standard and are available on most Unix variants.

209 - Whats the use of `fopen()`, `fclose()`, `fprintf()`, `getc()`, `putc()`, `getw()`, `putw()`, `fscanf()`, `feof()`, `ftell()`, `fseek()`, `rewind()`, `fread()`, `fwrite()`, `fgets()`, `fputs()`, `freopen()`, `fflush()`, `ungetc()`?
`fopen()`

This function is used to open a stream.

```
FILE *fp;
fp = fopen("filename", "mode");
fp = fopen("data", "r");
fp = fopen("results", "w");
```

Modes

"r"	-> Open for reading.
"w"	-> Open for writing.
"a"	-> Open for appending.
"r+"	-> Both reading and writing.
"w+"	-> Both reading and writing, create new file if it exists,
"a+"	-> Open for both reading and appending.

`fopen()`

`fclose()` is used to close a stream .

```
fclose(fp);
```

`putc()`, `getc()`, `putw()`, `getw()`, `fgetc()`, `getchar()`, `putchar()`, `fputs()`

These functions are used to read/write different types of data to the stream.

```
putc(ch, fp);
c=getc(fp);
putw(integer, fp);
integer=getw(fp);
```

`fprintf()`, `fscanf()`

Read/Write formatted data from/to the stream.

```
fprintf(fp,"control string",list);
fscanf(fp,"control string", list);
```

feof()

Check the status of a stream

```
if (feof(fp) != 0)
```

ftell(), fseek(), rewind(), fgetpos(), fsetpos()

Reposition the file pointer of a stream

```
n=ftell(fp); //Relative offset (in bytes) of the current position.
rewind(fp);
```

```
fseek(fp, offset, position);
```

Position can be

```
0->start of file
1->current position
2->end of file
```

```
fseek(fp, 0L, 0); // Same as rewind.
fseek(fp, 0L, 1); // Stay at current position.
fseek(fp, 0L, 2); // Past end of file.
fseek(fp, m, 0); // Move to (m+1) byte.
fseek(fp, m, 1) // Go forward m bytes.
fseek(fp, -m, 1); // Go backward m bytes from current position.
fseek(fp, -m, 2); // Go backward from end of file.
```

fread(), fwrite()

Binary stream input/output.

```
fwrite(&customer, sizeof(record), 1, fp);
fread(&customer, sizeof(record), 1, fp);
```

Here is a simple piece of code which reads from a file

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

int main()
{
    FILE *f;
    char buffer[1000];
    f=fopen("E:\\Misc\\__Temp\\FileDrag\\Main.cpp", "r");
```

```

if(f)
{
    printf("\nOpenened the file!\n");
    while(fgets(buffer,1000,f))
    {
        printf("(%d)-> %s\n",strlen(buffer),buffer);
    }
}

fclose(f);
getch();
return(0);
}

```

210 - How to check if a file is a binary file or an ascii file?

Here is some sample C code. The idea is to check the bytes in the file to see if they are ASCII or not...

```

#include <stdio.h>

int main(int argc, char *argv[])
{
    unsigned char ch;
    FILE *file;
    int binaryFile = FALSE;

    file = fopen(<FILE_PATH>, "rb");           // Open in Binary mode for
    the first time.

    while((fread(&ch, 1, 1, file) == 1) && (binaryFile == FALSE))
    {
        if(ch < 9 || ch == 11 || (ch > 13 && ch < 32) || ch == 255)
        {
            binaryFile = 1;
        }
    }

    fclose(file);

    if(binaryFile)
        file = fopen(<FILE_PATH>, "rb");
    else
        file = fopen(<FILE_PATH>, "r");

    if(binaryFile)
    {
        while(fread(&ch, 1, 1, file) == 1)
        {
            // Do whatever you want here with the binary file byte...
        }
    }
    else
    {
        while(fread(&ch, 1, 1, file) == 1)
        {
            // This is ASCII data, can easily print it!

```

```

        putchar(ch);
    }

fclose(file);

return(0);
}

```

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

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';
```

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

This is a trick question :)

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 structure.
};
```

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

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

214 - Whats the difference between gets() and fgets()? Whats the correct way to use fgets() when reading a file?

Unlike `fgets()`, `gets()` cannot be told the size of the buffer it's to read into, so it cannot be prevented from overflowing that buffer. As a general rule, always use `fgets()`. `fgets()` also takes in the size of the buffer, so that the end of the array cannot be overwritten.

```
fgets(buffer, sizeof(buffer), stdin);
if((p = strchr(buffer, '\n')) != NULL)
{
    *p = '\0';
}
```

Also, `fgets()` does not delete the trailing `"\n"`, as `gets()`.

So what's the right way to use `fgets()` when reading a file?

```
while(!feof(inp_file_ptr))
{
    fgets(buffer, MAX_LINE_SIZE, inp_file_ptr);
    fputs(buffer, out_file_ptr);
}
```

The code above will copy the last line twice! This is because, in C, end-of-file is only indicated after an input routine has tried to read, and failed. We should just check the return value of the input routine (in this case, `fgets()` will return `NULL` on end-of-file); often, you don't need to use `feof()` at all.

This is the right way to do it

```
while(fgets(buffer, MAX_LINE_SIZE, inp_file_ptr))
{
    fputs(buffer, out_file_ptr);
}
```

215 - How can I have a variable field width with `printf`?

Use something like

```
printf("%*d", width, x);
```

Here is a C program...

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

#define WIDTH 5

int main ( void )
{
    char str1[] = "Good Boy";
    char str2[] = "The earth is round";

    int width = strlen ( str1 ) + WIDTH;
    int prec  = strlen ( str2 ) + WIDTH;
```

```

    printf ( "%*.*s\n", width, prec, str1 );
    return 0;
}

```

216 - How can I specify a variable width in a scanf() format string?

You cant!.

An asterisk in a scanf() format string means to suppress assignment. You may be able to use ANSI stringizing and string concatenation to accomplish about the same thing, or you can construct the scanf format string at run time.

217 - How can I convert numbers to strings (the opposite of atoi)?

Use sprintf()

Note!, since sprintf() is also a variable argument function, it fails badly if passed with wrong arguments or if some of the arguments are missed causing segmentation faults. So be very careful when using sprintf() and pass the right number and type of arguments to it!

218 - Why should one use strncpy() and not strcpy()? What are the problems with strncpy()?

strcpy() is the cause of many bugs. Thats because programmers almost always end up copying more data into a buffer than it can hold. To prevent this problem, strncpy() comes in handy as you can specify the exact number of bytes to be copied.

But there are problems with strncpy() also. strncpy() does not always place a '\0' terminator in the destination string. (Funnily, it pads short strings with multiple \0's, out to the specified length). We must often append a '\0' to the destination string by hand. You can get around the problem by using strncat() instead of strncpy(): if the destination string starts out empty, strncat() does what you probably wanted strncpy() to do. Another possibility is sprintf(dest, "%.*s", n, source). When arbitrary bytes (as opposed to strings) are being copied, memcpy() is usually a more appropriate function to use than strncpy().

219 - How does the function strtok() work?

strtok() is the only standard function available for "tokenizing" strings.

The strtok() function can be used to parse a string into tokens. The first call to strtok() should have the string as its first argument. Subsequent calls should have the first argument set to NULL. Each call returns a pointer to the next token, or NULL when no more tokens are found. If a token ends with a delimiter, this delimiting character is overwritten with a '\0' and a pointer to the next character is saved for the next call to strtok(). The delimiter string delim may be different for each call.

The strtok_r() function works the same as the strtok() function, but instead of using a static buffer it uses a pointer to a user allocated char* pointer. This pointer must be the same while parsing the same string.

An example

```

main()
{
    char str[]="This is a test";
    char *ptr[10];
    char *p;
    int i=1;
    int j;

    p=strtok(str, " ");

    if(p!=NULL)
    {
        ptr[0]=p;
        while(1)
        {
            p=strtok(NULL, " ");

            if(p==NULL) break;
            else
            {
                ptr[i]=p; i++;
            }
        }

        for(j=0; j<i; j++)
        {
            printf("\n%s\n", ptr[i]);
        }
    }
}

```

220 - Why do we get the floating point formats not linked error?

Some compilers leave out certain floating point support if it looks like it will not be needed. In particular, the non-floating-point versions of `printf()` and `scanf()` save space by not including code to handle `%e`, `%f`, and `%g`. It happens that Borland's heuristics for determining whether the program uses floating point are insufficient, and the programmer must sometimes insert a dummy call to a floating-point library function (such as `sqrt()`; any will do) to force loading of floating-point support.

221 - Why do some people put void cast before each call to printf()?

`printf()` returns a value. Some compilers (and specially lint) will warn about return values not used by the program. An explicit cast to `(void)` is a way of telling lint that you have decided to ignore the return value from this specific function call. It's also common to use void casts on calls to `strcpy()` and `strcat()`, since the return value is never used for anything useful.

222 - What is assert() and when would I use it?

An assertion is a macro, defined in `<assert.h>`, for testing assumptions. An assertion is an assumption made by the programmer. If its violated, it would be caught by this macro.

For example

```

int i, j;

for(i=0; i<=10; i++)

```



```
{
    j+=5;
    assert(i<5);
}
```

Runtime error: Abnormal program termination.
assert failed (i<5), <file name>,<line number>

If anytime during the execution, i gets a value of 0, then the program would break into the assertion since the assumption that the programmer made was wrong.

223 - What do memcpy(), memchr(), memcmp(), memset(), strdup(), strncat(), strcmp(), strncmp(), strcpy(), strncpy(), strlen(), strchr(), strchr(), strpbrk(), strspn(), strcspn(), strtok() do?

Here are the prototypes...

```
void *memchr(const void *s, int c, size_t n);
int memcmp(const void *s1, const void *s2, size_t n);
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
;
void *memmove(void *s1, const void *s2, size_t n);
void *memset(void *s, int c, size_t n);
char *strcat(char *restrict s1, const char *restrict s2);
char *strchr(const char *s, int c);
int strcmp(const char *s1, const char *s2);
char *strcpy(char *restrict s1, const char *restrict s2);
size_t strcspn(const char *s1, const char *s2);
char *strdup(const char *s1);
size_t strlen(const char *s);
char *strncat(char *restrict s1, const char *restrict s2, size_t n);
int strncmp(const char *s1, const char *s2, size_t n);
char *strncpy(char *restrict s1, const char *restrict s2, size_t n);
char *strpbrk(const char *s1, const char *s2);
char *strrchr(const char *s, int c);
size_t strspn(const char *s1, const char *s2);
char *strstr(const char *s1, const char *s2);
char *strtok(char *restrict s1, const char *restrict s2);
```

The memchr() function shall locate the first occurrence of c (converted to an unsigned char) in the initial n bytes (each interpreted as unsigned char) of the object pointed to by s. The memchr() function shall return a pointer to the located byte, or a null pointer if the byte does not occur in the object.

The memcmp() function shall compare the first n bytes (each interpreted as unsigned char) of the object pointed to by s1 to the first n bytes of the object pointed to by s2. The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type unsigned char) that differ in the objects being compared. The memcmp() function shall return an integer greater than, equal to, or less than 0, if the object pointed to by s1 is greater than, equal to, or less than

the object pointed to by s2, respectively.

The `memcpy()` function shall copy `n` bytes from the object pointed to by `s2` into the object pointed to by `s1`. If copying takes place between objects that overlap, the behavior is undefined. The `memcpy()` function returns `s1`; no value is reserved to indicate an error. The `memmove()` function shall copy `n` bytes from the object pointed to by `s2` into the object pointed to by `s1`. Copying takes place as if the `n` bytes from the object pointed to by `s2` are first copied into a temporary of `n` bytes that does not overlap the objects pointed to by `s1` and `s2`, and then the `n` bytes from the temporary array are copied into the object pointed to by `s1`. The `memmove()` function returns `s1`; no value is reserved to indicate an error.

The `memset()` function shall copy `c` (converted to an unsigned char) into each of the first `n` bytes of the object pointed to by `s`. The `memset()` function returns `s`; no value is reserved to indicate an error.

The `strcat()` function shall append a copy of the string pointed to by `s2` (including the terminating null byte) to the end of the string pointed to by `s1`. The initial byte of `s2` overwrites the null byte at the end of `s1`. If copying takes place between objects that overlap, the behavior is undefined. The `strcat()` function shall return `s1`; no return value is reserved to indicate an error.

The `strchr()` function shall locate the first occurrence of `c` (converted to a char) in the string pointed to by `s`. The terminating null byte is considered to be part of the string. Upon completion, `strchr()` shall return a pointer to the byte, or a null pointer if the byte was not found.

The `strcmp()` function shall compare the string pointed to by `s1` to the string pointed to by `s2`. The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type unsigned char) that differ in the strings being compared. Upon completion, `strcmp()` shall return an integer greater than, equal to, or less than 0, if the string pointed to by `s1` is greater than, equal to, or less than the string pointed to by `s2`, respectively.

The `strcpy()` function shall copy the string pointed to by `s2` (including the terminating null byte) into the array pointed to by `s1`. If copying takes place between objects that overlap, the behavior is undefined. The `strcpy()` function returns `s1`; no value is reserved to indicate an error.

The `strcspn()` function shall compute the length (in bytes) of the maximum initial segment of the string pointed to by `s1` which consists entirely of bytes not from the string pointed to by `s2`. The `strcspn()` function shall return the length of the computed segment of the string pointed to by `s1`; no return value is reserved to indicate an error.

The `strdup()` function shall return a pointer to a new string, which is a duplicate of the string pointed to by `s1`, if memory can be successfully allocated for the new string. The returned pointer can be passed to `free()`. A null pointer is returned if the new string cannot be created. The function may set `errno` to `ENOMEM` if the allocation failed.

The `strlen()` function shall compute the number of bytes in the string to which `s` points, not including the terminating null byte. The `strlen()` function shall return the length of `s`; no return value shall be reserved to indicate an error.

The `strncat()` function shall append not more than `n` bytes (a null byte and bytes that follow it are not appended) from the array pointed to by `s2` to the end of the string pointed to by `s1`. The initial byte of `s2` overwrites the null byte at the end of `s1`. A terminating null byte is always appended to the result. If copying takes place between objects that overlap, the behavior is undefined. The `strncat()` function shall return `s1`; no return value shall be reserved to indicate an error.

The `strncmp()` function shall compare not more than `n` bytes (bytes that follow a null byte are not compared) from the array pointed to by `s1` to the array pointed to by `s2`. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type unsigned char) that differ in the strings being compared. Upon successful completion, `strncmp()` shall return an integer greater than, equal to, or less than 0, if the possibly null-terminated array pointed to by `s1` is greater than, equal to, or less than the possibly null-terminated array pointed to by `s2` respectively.

The `strncpy()` function shall copy not more than `n` bytes (bytes that follow a null byte are not copied) from the array pointed to by `s2` to the array pointed to by `s1`. If copying takes place between objects that overlap, the behavior is undefined. If the array pointed to by `s2` is a string that is shorter than `n` bytes, null bytes shall be appended to the copy in the array pointed to by `s1`, until `n` bytes in all are written. The `strncpy()` function shall return `s1`; no return value is reserved to indicate an error.

The `strpbrk()` function shall locate the first occurrence in the string pointed to by `s1` of any byte from the string pointed to by `s2`. Upon successful completion, `strpbrk()` shall return a pointer to the byte or a null pointer if no byte from `s2` occurs in `s1`.

The `strrchr()` function shall locate the last occurrence of `c` (converted to a char) in the string pointed to by `s`. The terminating null byte is considered to be part of the string. Upon successful completion, `strrchr()` shall return a pointer to the byte or a null pointer if `c` does not occur in the string.

The `strspn()` function shall compute the length (in bytes) of the maximum initial segment of the string pointed to by `s1` which consists entirely of bytes from the string pointed to by `s2`. The `strspn()` function shall return the computed length; no return value is reserved to indicate an error.

The `strstr()` function shall locate the first occurrence in the string pointed to by `s1` of the sequence of bytes (excluding the terminating null byte) in the string pointed to by `s2`. Upon successful completion, `strstr()` shall return a pointer to the located string or a null pointer if the string is not found. If `s2` points to a string with zero length, the function shall return `s1`.

A sequence of calls to `strtok()` breaks the string pointed to by `s1` into a sequence of tokens, each of which is delimited by a byte from the string pointed to by `s2`. The first

call in the sequence has s1 as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by s2 may be different from call to call. The first call in the sequence searches the string pointed to by s1 for the first byte that is not contained in the current separator string pointed to by s2. If no such byte is found, then there are no tokens in the string pointed to by s1 and strtok() shall return a null pointer. If such a byte is found, it is the start of the first token. The strtok() function then searches from there for a byte that is contained in the current separator string. If no such byte is found, the current token extends to the end of the string pointed to by s1, and subsequent searches for a token shall return a null pointer. If such a byte is found, it is overwritten by a null byte, which terminates the current token. The strtok() function saves a pointer to the following byte, from which the next search for a token shall start. Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above. Upon successful completion, strtok() shall return a pointer to the first byte of a token. Otherwise, if there is no token, strtok() shall return a null pointer.

224 - What does alloca() do?

alloca() allocates memory which is automatically freed when the function which called alloca() returns. However, note that alloca() is not portable and its usage is not recommended.

225 - Can you compare two strings like string1==string2? Why do we need strcmp()? Do you think this will work?

```
if(string1 == string2)
{
}
}
```

No!, strings in C cannot be compared like that!.

The == operator will end up comparing two pointers (that is, if they have the same address). It won't compare the contents of those locations. In C, strings are represented as arrays of characters, and the language never manipulates (assigns, compares, etc.) arrays as a whole.

The correct way to compare strings is to use strcmp()

```
if(strcmp(string1, string2) == 0)
{
}
}
```

226 - What does printf() return?

Upon a successful return, the printf() function returns the number of characters printed (not including the trailing '\0' used to end output to strings). If the output was truncated due to this limit then the return value is the number of characters (not including the trailing '\0') which would have been written to the final string if enough space had been available. Thus, a return value of size or more means that the output was truncated. If an

output error is encountered, a negative value is returned.

There is a very funny yet interesting interview question around this....

Look at the code below.

```
void main()
{
    if(X)
    {
        printf("Hello");
    }
    else
    {
        printf(" World");
    }
}
```

What should X be replaced with inorder to get the output as "Hello World"?

And here comes the answer....

```
#include <stdio.h>
int main()
{
    if(!printf("Hello"))
    {
        printf("Hello");
    }
    else
    {
        printf(" World");
    }
}
```

Kind of stupid isn't it? But they do ask these type of questions. Believe me!

227 - What do setjmp() and longjmp() functions do?

It is not possible to jump from one function to another by means of a goto and a label, since labels have only function scope. However, the macro setjmp and function longjmp provide an alternative, known as a non-local goto, or a non-local jump. The header file <setjmp.h> declares something called a jmp_buf, which is used by the cooperating macro and function to store the information necessary to make the jump. The declarations are as follows:

```
#include <setjmp.h>
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

The setjmp macro is used to initialise the jmp_buf and returns zero on its initial call. Then, it returns again, later, with a non-zero value, when the corresponding longjmp call is made! The non-zero value is whatever value was supplied to the call of longjmp.

This is best explained by way of an example:

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

void func(void);
jmp_buf place;

main()
{
    int retval;

    /*
     * First call returns 0,
     * a later longjmp will return non-zero.
     */

    if(setjmp(place)) != 0)
    {
        printf("Returned using longjmp\n");
        exit(EXIT_SUCCESS);
    }

    /*
     * This call will never return - it
     * 'jumps' back above.
     */

    func();
    printf("What! func returned!\n");
}

void func(void)
{
    /*
     * Return to main.
     * Looks like a second return from setjmp,
     * returning 4!
     */

    longjmp(place, 4);
    printf("What! longjmp returned!\n");
}
```

The val argument to longjmp is the value seen in the second and subsequent ?returns? from setjmp. It should normally be something other than 0; if you attempt to return 0 via longjmp, it will be changed to 1. It is therefore possible to tell whether the setjmp was called directly, or whether it was reached by calling longjmp. If there has been no call to setjmp before calling longjmp, the effect of longjmp is undefined, almost certainly causing the program to crash. The longjmp function is never expected to return, in the normal sense, to the instructions immediately following the call. All accessible objects on ?return? from setjmp have the values that they had when longjmp was called, except for objects of automatic storage class that do not have volatile type; if they have been changed between the setjmp and longjmp calls, their values are indeterminate.

The longjmp function executes correctly in the contexts of interrupts, signals and any of their associated functions. If longjmp is invoked from a function called as a result of a signal arriving while handling another signal, the behaviour is undefined.

It's a serious error to longjmp to a function which is no longer active (i.e. it has already returned or another longjmp call has transferred to a setjmp occurring earlier in a set of nested calls).

The Standard insists that, apart from appearing as the only expression in an expression statement, setjmp may only be used as the entire controlling expression in an if, switch, do, while, or for statement. A slight extension to that rule is that as long as it is the whole controlling expression (as above) the setjmp call may be the subject of the ! operator, or may be directly compared with an integral constant expression using one of the relational or equality operators. No more complex expressions may be employed.

Examples are:

```
setjmp(place);                /* expression statement */
if(setjmp(place)) ...         /* whole controlling expression */
if(!setjmp(place)) ...        /* whole controlling expression */
if(setjmp(place) < 4) ...      /* whole controlling expression */
if(setjmp(place)<;4 && 1!=2) ... /* forbidden */
```

228 - Whats the prototype of main()? Can main() return a structure?

The right declaration of main() is

```
int main(void)
```

or

```
int main(int argc, char *argv[])
```

or

```
int main(int argc, char *argv[], char *env[]) //Compiler dependent, non
-standard C.
```

In C, main() cannot return anything other than an int. Something like

```
void main()
```

is illegal. There are only three valid return values from main() - 0, EXIT_SUCCESS, and EXIT_FAILURE, the latter two are defined in <stdlib.h>.

Something like this can cause unpredicted behavior

```

struct mystruct {
    int value;
    struct mystruct *next;
}

main(argc, argv)
{ ... }

```

Here the missing semicolon after the structure declaration causes main to be misdeclared.

229 - Is exit(status) equivalent to returning the same status from main()?

No.

The two forms are not equivalent in a recursive call to main().

230 - Can main() be called recursively?

Yes

```

main()
{
    main();
}

```

But this will go on till a point where you get a

Runtime error : Stack overflow.

231 - How to print the arguments recieved by main()?

Here is some code

```

main(int argc, char*argv[])
{
    int i;
    for(i=0; i<argc; i++)
    {
        printf("\n[%s]\n", argv[i]);
    }
}

```

OS Concepts

232 - What do the system calls fork(), vfork(), exec(), wait(), waitpid() do? Whats a Zombie process? Whats the difference between fork() and vfork()?

The system call fork() is used to create new processes. It does not take any arguments and returns a process ID. The purpose of fork() is to create a new process, which becomes the child process of the caller (which is called the parent). After a new child process is created, both processes will execute the next instruction following the fork() system call. We can distinguish the parent from the child by testing the returned value of fork():

If fork() returns a negative value, the creation of a child process was unsuccessful. A call to fork() returns a zero to the newly created child process and the same call to fork() returns a positive value (the process ID of the child process) to the parent. The returned process ID is of type pid_t defined in sys/types.h. Normally, the process ID is an integer. Moreover, a process can use function getpid() to retrieve the process ID assigned to this process. Unix will make an exact copy of the parent's address space and give it to the

child. Therefore, the parent and child processes will have separate address spaces. Both processes start their execution right after the system call `fork()`. Since both processes have identical but separate address spaces, those variables initialized before the `fork()` call have the same values in both address spaces. Since every process has its own address space, any modifications will be independent of the others. In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space. Other address spaces created by `fork()` calls will not be affected even though they have identical variable names.

Trick question!

How many processes are forked by the below program

```
main()
{
    fork();
    fork();
}
```

Answer is 2 raise to n, when n is the number of calls to fork (2 in this case).

Each process has certain information associated with it including:

The UID (numeric user identity)
The GID (numeric group identity)
A process ID number used to identify the process
A parent process ID
The execution status, e.g. active, runnable, waiting for input etc.
Environment variables and values.
The current directory.
Where the process currently resides in memory
The relative process priority see `nice(1)`
Where it gets standard input from
Where it sends standard output to
Any other files currently open

Certain process resources are unique to the individual process. A few among these are:

- Stack Space: which is where local variables, function calls, etc. are stored.
- Environment Space: which is used for storage of specific environment variables.
- Program Pointer (counter) : PC.
- File Descriptors
- Variables

Under unix, each sub directory under /proc corresponds to a running process (PID #). A ps provide detailed information about processes running

A typical output of ps looks as follows:

COMMAND	PID	TTY	STAT	TIME	C

login)	8811	3	SW	0:00	(
bash)	3466	3	SW	0:00	(
startx)	8777	3	SW	0:00	(
.
s	1262	p7	R	0:00	p

The columns refer to the following:

PID - The process id's (PID).
 TTY - The terminal the process was started from.
 STAT -
 The current status of all the processes. Info about the process status can be broken into more than 1 field. The first of these fields can contain the following entries:

R	- Runnable.
S	- Sleeping.
D	- Un-interruptable sleep.
T	- Stopped or Traced.
Z	- Zombie Process.

The second field can contain the following entry:

W	- If the process has no residual pages.
---	---

And the third field:

N	- If the process has a positive nice value.
TIME	- The CPU time used by the process so far.
COMMAND	- The actual command.

The init process is the very first process run upon startup. It starts additional processes. When it runs it reads a file called /etc/inittab which specifies init how to set up the system, what processes it should start with respect to specific runlevels. One crucial process which it starts is the getty program. A getty process is usually started for each

terminal upon which a user can log into the system. The `getty` program produces the login: prompt on each terminal and then waits for activity. Once a `getty` process detects activity (at a user attempts to log in to the system), the `getty` program passes control over to the `login` program.

There are two command to set a process's priority `nice` and `renice`.

One can start a process using the `system()` function call

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

int main()
{
    printf("Running ls.....\n");
    system("ls -lrt");
    printf("Done.\n");
    exit(0);
}
```

The `exec()` system call

The `exec()` functions replace a current process with another created according to the arguments given.

The syntax of these functions is as follows:

```
#include <unistd.h>

char *env[];

int  execl(const char *path, const char *arg0, ..., (char *)0);
int  execv(const char *path, const char *argv[]);

int  execlp(const char *path, const char *arg0, ..., (char *)0);
int  execvp(const char *path, const char *argv[]);

int  execl(const char *path, const char *arg0, ... , (char *)0, const c
har *env[]);
int  execve(const char *path, const char *argv[], const char *env[]);
```

The program given by the `path` argument is used as the program to execute in place of what is currently running. In the case of the `execl()` the new program is passed arguments `arg0`, `arg1`, `arg2`,... up to a null pointer. By convention, the first argument supplied (i.e. `arg0`) should point to the file name of the file being executed. In the case of the `execv()` programs the arguments can be given in the form of a pointer to an array of strings, i.e.

the argv array. The new program starts with the given arguments appearing in the argv array passed to main. Again, by convention, the first argument listed should point to the file name of the file being executed. The function name suffixed with a p (execlp() and execvp()) differ in that they will search the PATH environment variable to find the new program executable file. If the executable is not on the path, and absolute file name, including directories, will need to be passed to the function as a parameter. The global variable environ is available to pass a value for the new program environment. In addition, an additional argument to the exec() functions execl(), execlp(), execle(), execvp(), and execve() is available for passing an array of strings to be used as the new program environment.

Examples to run the ls command using exec are:

```
const char *argv[] = ("ls", "-lrt", 0);
const char *env[] = {"PATH=/bin:/usr/bin", "TERM=console", 0};

execl("/bin/ls", "ls", "-lrt", 0);
execvp("/bin/ls", argv);

execlp("ls", "ls", "-lrt", 0);
execle("/bin/ls", "ls", "-lrt", 0, env);

execvp("ls", argv);
execve("/bin/ls", argv, env);
```

A simple call to fork() would be something like this

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    pid_t pid;
    pid=fork();
    switch(pid)
    {
        case -1: exit(1); // fork() error.
        case 0: // Child process, can call exec here.
            break;
        default: // Parent.
            break;
    }
    exit(0);
}
```

The call wait() can be used to determine when a child process has completed its job and finished. We can arrange for the parent process to wait until the child finishes before continuing by calling wait(). wait() causes a parent process to pause until one of the child processes dies or is stopped. The call returns the PID of the child process for which status information is available. This will usually be a child process which has terminated. The

status information allows the parent process to determine the exit status of the child process, the value returned from main or passed to exit. If it is not a null pointer the status information will be written to the location pointed to by stat_loc. We can interrogate the status information using macros defined in sys/wait.h.

Macro	Definition

WIFEXITED(stat_val);	Nonzero if the child is terminated normally
WEXITSTATUS(stat_val);	If WIFEXITED is nonzero, this returns child exit code.
WIFSIGNALED(stat_val);	Nonzero if the child is terminated on an uncaught signal.
WTERMSIG(stat_val);	If WIFSIGNALED is nonzero, this returns a signal number.
WIFSTOPPED(stat_val);	Nonzero if the child stopped on a signal.
WSTOPSIG(stat_val);	If WIFSTOPPED is nonzero, this returns a signal number.

An example code which used wait() is shown below

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(void){
    pid_t child_pid;
    int *status=NULL;
    if(fork()){
        /* wait for child, getting PID */
        child_pid=wait(status);
        printf("I'm the parent.\n");
        printf("My child's PID was: %d\n",child_pid);
    } else {
        printf("I'm the child.\n");
    }
    return 0;
}
```

Or a more detailed program

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    pid_t pid;
    int exit_code;
```

```

pid = fork();
switch(pid)
{
    case -1: exit(1);
    case 0:  exit_code = 11; //Set the child exit process
            break;
    default: exit_code = 0;
            break;
}

if (pid)
{
    // This is the parent process
    int status;
    pid_t child_pid;

    child_pid = wait(&status);

    printf("Child process finished with PID [%d]\n", child_pid);
    if (WIFEXITED(status))
    {
        printf("Child exited with code [%d]\n", WEXITSTATUS(status));
    }
    else
    {
        printf("Child terminated abnormally.\n");
    }
}

exit(exit_code);
}

```

So now, whats a Zombie process?

When using fork() to create child processes it is important to keep track of these processes. For instance, when a child process terminates, an association with the parent survives until the parent either terminates normally or calls wait(). The child process entry in the process table is not freed up immediately. Although it is no longer active, the child process is still in the system because its exit code needs to be stored in the even the parent process calls wait(). The child process is at that point referred to as a zombie process. Note, if the parent terminates abnormally then the child process gets the process with PID 1, (init) as parent. (such a child process is often referred to as an orphan). The child process is now a zombie. It is no longer running, its original parent process is gone, and it has been inherited by init. It will remain in the process table as a zombie until the next time the table is processed. If the process table is long this may take a while. till init cleans them up. As a general rule, program wisely and try to avoid zombie processes. When zombies accumulate they eat up valuable resources.

The waitpid() system call is another call that can be used to wait for child processes. This system call however can be used to wait for a specific process to terminate.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

The pid argument specifies the PID of the particular child process to wait for. If it is a -1 then waitpid() will return information to the child process. Status information will be written to the location pointed to by status. The options argument enables us to change the behavior of waitpid(). A very useful option is WNOHANG which prevents the call to waitpid() from suspending the execution of the caller. We can use it to find out whether any child process has terminated and, if not, to continue.

Synchronous and asynchronous process execution

In some cases, for example if the child process is a server or "daemon" (a process expected to run all the time in the background to deliver services such as mail forwarding) the parent process would not wait for the child to finish. In other cases, e.g. running an interactive command where it is not good design for the parent's and child's output to be mixed up into the same output stream, the parent process, e.g. a shell program, would normally wait for the child to exit before continuing. If you run a shell command with an ampersand as its last argument, e.g. sleep 60 & the parent shell doesn't wait for this child process to finish.

So what's the difference between fork() and vfork()?

The system call vfork(), is a low overhead version of fork(), as fork() involves copying the entire address space of the process and is therefore quite expensive. The basic difference between the two is that when a new process is created with vfork(), the parent process is temporarily suspended, and the child process might borrow the parent's address space. This strange state of affairs continues until the child process either exits, or calls execve(), at which point the parent process continues. This means that the child process of a vfork() must be careful to avoid unexpectedly modifying variables of the parent process. In particular, the child process must not return from the function containing the vfork() call, and it must not call exit() (if it needs to exit, it should use _exit()); actually, this is also true for the child of a normal fork()).

However, since vfork() was created, the implementation of fork() has improved, most notably with the introduction of 'copy-on-write', where the copying of the process address space is transparently faked by allowing both processes to refer to the same physical memory until either of them modify it. This largely removes the justification for vfork(); indeed, a large proportion of systems now lack the original functionality of vfork() completely. For compatibility, though, there may still be a vfork() call present, that simply calls fork() without attempting to emulate all of the vfork() semantics.

233 - How does freopen() work?

234 - What are threads? What is a lightweight process? What is a heavyweight process? How different is a thread from a process?

235 - How are signals handled?

236 - What is a deadlock?

237 - What are semaphores?

238 - What is meant by context switching in an OS?

239 - What is Belady's anomaly?

Usually, on increasing the number of frames allocated to a process' virtual memory, the process execution is faster, because fewer page faults occur. Sometimes, the reverse happens, i.e., the execution time increases even when more frames are allocated to the process. This is known as the Belady's Anomaly. This is true for certain page reference patterns. This is also called the FIFO anomaly.

General Concepts

240 - What is the difference between statically linked libraries and dynamically linked libraries (dll)?

Static linking means all the referenced code is included in the binary produced. When linking statically, the linker finds the bits that the program modules need, and physically copies them into the executable output file that it generates.

In case of dynamic linking, the binary simply contains a pointer to the needed routine, which will be loaded by the OS as needed. An important benefit is that libraries can be updated to fix bugs or security problems and the binaries that use them are immediately using the fixed code. It also saves disk space, saves memory usage, saves launch time, improve multi-tasking efficiency. But, deployment of dynamically-linked executable generally requires coordination across hosts of shared-object libraries, installation locations, installation environment settings, settings for compile-, link-, and use-time environment variables. Dynamic linking often precludes relocations backward across OS versions.

On Linux, static libraries have names like libname.a, while shared libraries are called libname.so.x.y.z where x.y.z is some form of version number. Shared libraries often also have links pointing to them, which are important, and (on a.out configurations) associated .sa files. The standard libraries come in both shared and static formats. You can find out what shared libraries a program requires by using ldd (List Dynamic Dependencies)

Dynamic linking allows an exe or dll to use required information at run time to call a DLL function. In static linking, the linker gets all the referenced functions from the static link library and places it with your code into your executable. Using DLLs instead of static link libraries makes the size of the executable file smaller. Dynamic linking is faster than static linking.

241 - What are the most common causes of bugs in C?

Murphy's law states that "If something can go wrong, it will. So is the case with programming. Here are a few things that can go wrong when coding in C.

- Typing `if(x=0)` instead of `if(x==0)`. Use `if(0==x)` instead.
- Using `strcpy()` instead of `strncpy()` and copying more memory than the buffer can hold.
- Dereferencing null pointers.
- Improper `malloc()`, `free()` usage. Assuming `malloc`'ed memory contains 0, assuming freed storage persists even after freeing it, freeing something twice, corrupting the `malloc()` data structures.
- Uninitialized local variables.
- Integer overflow.
- Mismatch between `printf()` format and arguments, especially trying to print long ints using `%d`. Problems with wrong arguments to `sprintf()` leading to core dumps.
- Array out of bounds problems, especially of small, temporary buffers.
- Leaving files opened.
- Undefined evaluation order, undefined statements.
- Omitted declaration of external functions, especially those which return something other than `int`, or have "narrow" or variable arguments.
- Floating point problems.
- Missing function prototypes.

Ofcourse, there is the mother of all things that can go wrong - Your logic!.

242 - What is hashing?

Hashing is the process of mapping strings to integers, in a relatively small range.

A hash function maps a string (or some other data structure) to a bounded number (called the hash bucket) which can more easily be used as an index in an array, or for performing repeated comparisons.

A mapping from a potentially huge set of strings to a small set of integers will not be unique. Any algorithm using hashing therefore has to deal with the possibility of collisions.

Here are some common methods used to create hashing functions

Direct method
Subtraction method
Modulo-Division method
Digit-Extraction method
Mid-Square method
Folding method
Pseudo-random method

Here is a simple implementation of a hashing scheme in C

```
#include <stdio.h>
#define HASHSIZE 197

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

typedef struct hashtable
{
    mynode *l1ist;
}HT;

HT myHashTable[HASHSIZE];

int main()
{
    initialize();
    add(2);
    add(2500);
    add(199);

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

int initialize()
{
    int i;
    for(i=0;i<HASHSIZE;i++)
        myHashTable[i].l1ist=NULL;
    return(1);
}

int add(int value)
{
    int hashkey;
    int i;
    mynode *tempnode1, *tempnode2, *tempnode3;

    hashkey=value%HASHSIZE;
    printf("\nHashkey : [%d]\n", hashkey);

    if(myHashTable[hashkey].l1ist==NULL)
    {
        //This hash bucket is empty, add the first element!
        tempnode1 = malloc(sizeof(mynode));
        tempnode1->value=value;
        tempnode1->next=NULL;
        myHashTable[hashkey].l1ist=tempnode1;
    }
}
```

```

else
{
    //This hash bucket already has some items. Add to it at the end.
    //Check if this element is already there?

    for(tempnode1=myHashTable[hashkey].l1ist;
        tempnode1!=NULL;
        tempnode3=tempnode1,tempnode1=tempnode1->next)
    {
        if(tempnode1->value==value)
        {
            printf("\nThis value [%d] already exists in the Hash!\n", v
alue);
            return(1);
        }
    }

    tempnode2 = malloc(sizeof(mynode));
    tempnode2->value = value;
    tempnode2->next=NULL;
    tempnode3->next=tempnode2;
}
return(1);
}

int display_hash()
{
    int i;
    mynode *tempnode;

    for(i=0;i<HASHSIZE;i++)
    {
        if(myHashTable[i].l1ist==NULL)
        {
            printf("\nmyHashTable[%d].l1ist -> (empty)\n",i);
        }
        else
        {
            printf("\nmyHashTable[%d].l1ist -> ",i);
            for(tempnode=myHashTable[i].l1ist;tempnode!=NULL;tempnode=tempn
ode->next)
            {
                printf("[%d] -> ",tempnode->value);
            }
            printf("(end)\n");
        }
        if(i%20==0)getch();
    }
    return(0);
}

```

And here is the output

Hashkey : [2]

```

Hashkey : [136]

Hashkey : [2]

Hashkey : [2]

This value [2] already exists in the Hash!

myHashTable[0].l1ist -> (empty)
myHashTable[1].l1ist -> (empty)
myHashTable[2].l1ist -> [2] -> [199] -> (end)
myHashTable[3].l1ist -> (empty)
myHashTable[4].l1ist -> (empty)
myHashTable[5].l1ist -> (empty)
myHashTable[6].l1ist -> (empty)
myHashTable[7].l1ist -> (empty)
myHashTable[8].l1ist -> (empty)
myHashTable[9].l1ist -> (empty)
myHashTable[10].l1ist -> (empty)
myHashTable[11].l1ist -> (empty)
myHashTable[12].l1ist -> (empty)
myHashTable[13].l1ist -> (empty)
myHashTable[14].l1ist -> (empty)
myHashTable[15].l1ist -> (empty)
myHashTable[16].l1ist -> (empty)
myHashTable[17].l1ist -> (empty)
myHashTable[18].l1ist -> (empty)
myHashTable[19].l1ist -> (empty)
myHashTable[20].l1ist -> (empty)
...
myHashTable[133].l1ist -> (empty)
myHashTable[134].l1ist -> (empty)
myHashTable[135].l1ist -> (empty)
myHashTable[136].l1ist -> [2500] -> (end)

```

```
myHashTable[137].l1list -> (empty)
```

...

```
myHashTable[196].l1list -> (empty)
```

243 - What do you mean by Predefined?

It means already written, compiled and linked together with our program at the time of linking.

244 - What is data structure?

A data structure is a way of organizing data that considers not only the items stored, but also their relationship to each other. Advance knowledge about the relationship between data items allows designing of efficient algorithms for the manipulation of data.

245 - What are the time complexities of some famous algorithms?

Binary search	:	$O(\log n)$
Finding max & min for a given set of numbers	:	$O(3n/2-2)$
Merge Sort	:	$O(n \log n)$
Insertion Sort	:	$O(n^2)$
Quick Sort	:	$O(n \log n)$
Selection Sort	:	$O(n^2)$

246 - What is row major and column major form of storage in matrices?

This is row major representation

If this is your matrix

```
a b c d
e f g h
i j k l
```

Convert into 1D array by collecting elements by rows. Within a row elements are collected from left to right.

Rows are collected from top to bottom. So, $x[i][j]$ is mapped to position $i * \text{no_of_columns} + j$.

This is column major representation

On similar lines, in column major representation, we store elements column wise. Here, $x[i][j]$ is mapped to position $i + j * \text{no_of_rows}$.

247 - Explain the BigOh notation.

Time complexity

The Big Oh for the function $f(n)$ states that

$f(n) = O(g(n))$; iff there exist positive constants c and d such that: $f(n) \leq c * g(n)$ for all $n, n \geq d$.

Here are some examples...

1	->Constant
logn	->Logarithmic
n	->Linear
nlogn	->nlogn
n*n	->quadratic
n*n*n	->cubic
2^n	->exponential
n!	->Factorial

We also have a few other notations...

Omega notation

$f(n) = \Omega(g(n))$, iff positive constants c and d exist such as $f(n) \geq cg(n)$ for all n , $n \geq d$.

Theta notation

$f(n) = \Theta(g(n))$, iff positive constants c_1 and c_2 and an d exists such that $c_1g(n) \leq f(n) \leq c_2g(n)$ for all n , $n \geq d$.

Little Oh(o)

$f(n) = o(g(n))$, iff $f(n) = O(g(n))$ and $f(n) \neq \Omega(g(n))$.

Then there is also something called Space complexity...

Fixed part : This is the independent of the instance characteristics. This typically includes the instruction space (i.e. space for the code), space for simple variables and fixed-size component variables, space for constants, and so on.

Variable part: A variable part that consists of the space needed by the component variables whose size depends on the particular problem instance being solved, dynamically allocated space (to the extent that this space depends on the instance characteristics); and the recursion stack space (in so far as this space depends on the instance characteristics).

A complete description of these concepts is out of the scope of this website. There are plenty of books and millions of pages on the Internet. Help yourself!

248 - Give the most important types of algorithms.

There are five important basic designs for algorithms.

They are:

- **Divide and conquer** : For a function to compute on n inputs the divide and conquer strategy suggests the inputs into a k distinct subsets, $1 < k \leq n$, yielding k sub-problems. These sub-problems must be solved and then a method must be found to combine the sub-solutions into a solution of the whole. An example for this approach is "binary search" algorithm. The time complexity of binary search algorithm is $O(\log n)$.
- **The greedy method** : The greedy method suggests that one can devise an algorithm that works in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is an optimal solution. An example for solution using greedy method is "knapsack problem". Greedy algorithms attempt not only to find a solution, but to find the ideal solution to any given problem.
- **Dynamic programming** : Dynamic Programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions. An example for algorithm using dynamic programming is "multistage graphs". This class remembers older results and attempts to use this to speed the process of finding new results.
- **Back-tracking** : Here if we reach a dead-end, we use the stack to pop-off the dead end and try something else we had not tried before. The famous 8-queens problem uses back tracking. Backtracking algorithms test for a solution, if one is found the algorithm has solved, if not it recurs once and tests again, continuing until a solution is found.
- **Branch and bound** : Branch and bound algorithms form a tree of subproblems to the primary problem, following each branch until it is either solved or lumped in with another branch.

249 - What is marshalling and demarshalling?

When objects in memory are to be passed across a network to another host or persisted to storage, their in-memory representation must be converted to a suitable out-of-memory format. This process is called marshalling, and converting back to an in memory representation is called demarshalling. During marshalling, the objects must be respresented with enough information that the destination host can understand the type of object being created. The objects' state data must be converted to the appropriate format. Complex object trees that refer to each other via object references (or pointers) need to refer to each other via some form of ID that is independent of any memory model. During demarshalling, the destination host must reverse all that and must also validate that the objects it receives are consistent with the expected object type (i.e. it validate that it doesn't get a string where it expects a number).

250 - How can we detect and prevent integer overflow and underflow?

254 - How to list all the predefined identifiers?

gcc provides a -dM option which works with -E.

255 - How the compiler make difference between C and C++?

Most compilers recognized the file type by looking at the file extension.

You might also be able to force the compiler to ignore the file type by supplying compiler switch. In MS VC++ 6, for example, the MSCRT defines a macro, `__cplusplus`. If you undefine that macro, then the compiler will treat your code as C code. You don't define the `__cplusplus` macro. It is defined by the compiler when compiling a C++ source. In MSVC6 there's a switch for the compiler, `/Tc`, that forces a C compilation instead of C++.

256 - What are the general steps in compilation?

1. Lexical analysis.
2. Syntactic analysis.
3. Semantic analysis.
4. Pre-optimization of internal representation.
5. Code generation.
6. Post optimization.

257 - What are the different types of linkages?

Linkage is used to determine what makes the same name declared in different scopes refer to the same thing. An object only ever has one name, but in many cases we would like to be able to refer to the same object from different scopes. A typical example is the wish to be able to call `printf()` from several different places in a program, even if those places are not all in the same source file.

The Standard warns that declarations which refer to the same thing must all have compatible type, or the behaviour of the program will be undefined. Except for the use of the storage class specifier, the declarations must be identical.

The three different types of linkage are:

- external linkage
- internal linkage
- no linkage

In an entire program, built up perhaps from a number of source files and libraries, if a name has external linkage, then every instance of a that name refers to the same object throughout the program. For something which has internal linkage, it is only within a given source code file that instances of the same name will refer to the same thing. Finally, names with no linkage refer to separate things.

Linkage and definitions

Every data object or function that is actually used in a program (except as the operand of a `sizeof` operator) must have one and only one corresponding definition. This "exactly one" rule means that for objects with external linkage there must be exactly one

definition in the whole program; for things with internal linkage (confined to one source code file) there must be exactly one definition in the file where it is declared; for things with no linkage, whose declaration is always a definition, there is exactly one definition as well.

The three types of accessibility that you will want of data objects or functions are:

- Throughout the entire program,
- Restricted to one source file,
- Restricted to one function (or perhaps a single compound statement).

For the three cases above, you will want external linkage, internal linkage, and no linkage respectively. The external linkage declarations would be prefixed with `extern`, the internal linkage declarations with `static`.

```
#include <stdio.h>

// External linkage.
extern int var1;

// Definitions with external linkage.
extern int var2 = 0;

// Internal linkage:
static int var3;

// Function with external linkage
void f1(int a){}

// Function can only be invoked by name from within this file.
static int f2(int a1, int a2)
{
    return(a1 * a2);
}
```

258 - What do you mean by scope and duration?

The duration of an object describes whether its storage is allocated once only, at program start-up, or is more transient in its nature, being allocated and freed as necessary.

There are only two types of duration of objects: static duration and automatic duration. Static duration means that the object has its storage allocated permanently, automatic means that the storage is allocated and freed as necessary.

It's easy to tell which is which: you only get automatic duration if

- The declaration is inside a function.
- And the declaration does not contain the `static` or `extern` keywords.
- And the declaration is not the declaration of a function.

The scope of the names of objects defines when and where a given name has a particular meaning. The different types of scope are the following:

- function scope
- file scope
- block scope
- function prototype scope

The easiest is function scope. This only applies to labels, whose names are visible throughout the function where they are declared, irrespective of the block structure. No two labels in the same function may have the same name, but because the name only has function scope, the same name can be used for labels in every function. Labels are not objects?they have no storage associated with them and the concepts of linkage and duration have no meaning for them. Any name declared outside a function has file scope, which means that the name is usable at any point from the declaration on to the end of the source code file containing the declaration. Of course it is possible for these names to be temporarily hidden by declarations within compound statements. As we know, function definitions must be outside other functions, so the name introduced by any function definition will always have file scope. A name declared inside a compound statement, or as a formal parameter to a function, has block scope and is usable up to the end of the associated `}` which closes the compound statement. Any declaration of a name within a compound statement hides any outer declaration of the same name until the end of the compound statement. A special and rather trivial example of scope is function prototype scope where a declaration of a name extends only to the end of the function prototype. The scope of a name is completely independent of any storage class specifier that may be used in its declaration.

261 Gotcha! programs

Here are a few Gotcha! programs worth looking at....

•

```
int main()
{
    int const * p=5;
    printf("%d",++(*p));
}
```

Answer: "Cannot modify a constant value". `p` is a pointer to a "constant integer". But we tried to change the value of the "constant integer".

•

```
int main()
{
    char s[ ]="man";
    int i;
    for(i=0;s[ i ];i++)
    {
```

```

        printf("\n%c%c%c%c",s[ i ],*(s+i),*(i+s),i[s]);
    }
}

```

Answer:

mmmm
aaaa
nnnn

Thanks because, `s[i]`, `*(i+s)`, `*(s+i)`, `i[s]` are all different ways of expressing the same idea. Generally array name is the base address for that array. Here `s` is the base address. `i` is the index number/displacement from the base address. So, indirecting it with `*` is same as `s[i]`. `i[s]` may be surprising. But in the case of C it is same as `s[i]`.

•

```

int main()
{
    static int var = 5;
    printf("%d ",var--);
    if(var)
        main();
    return(0);
}

```

Answer: "5 4 3 2 1". When static storage class is given, it is initialized once. The change in the value of a static variable is retained even between the function calls. Main is also treated like any other ordinary function, which can be called recursively.

•

```

int main()
{
    extern int i;
    i=20;
    printf("%d",i);
    return(0);
}

```

Answer: "Linker Error : Undefined symbol '_i'". The extern storage class in the following declaration, `extern int i;` specifies to the compiler that the memory for `i` is allocated in some other program and that address will be given to the current program at the time of linking. But linker finds that no other variable of name `i` is available in any other program with memory space allocated for it. Hence a linker error has occurred .

•

```

int main()
{
    char string[]="Hello World";

```

```

    display(string);
    return(0);
}

void display(char *string)
{
    printf("%s", string);
}

```

Answer: "Compiler Error : Type mismatch in redeclaration of function display". In third line, when the function display is encountered, the compiler doesn't know anything about the function display. It assumes the arguments and return types to be integers, (which is the default type). When it sees the actual function display, the arguments and type contradicts with what it has assumed previously. Hence a compile time error occurs.

•

```

int main()
{
    struct xx
    {
        char name[]="hello";
    };
    struct xx *s;
    printf("%s", s->name);
    return(0);
}

```

Answer: "Compiler Error". You should not initialize variables in structure declaration.

•

```

int main()
{
    printf("%p", main);
}

```

Answer: Some address will be printed. Function names are just addresses (just like array names are addresses).main() is also a function. So the address of function main will be printed. %p in printf specifies that the argument is an address. They are printed as hexadecimal numbers.

•

```

enum colors {BLACK,BLUE, GREEN};
int main()
{
    printf("%d..%d..%d", BLACK,BLUE, GREEN);
    return(0);
}

```

Answer: "0..1..2". The enum tag assigns numbers starting from 0, if not explicitly defined.

•

```
#include <stdio.h>
void fun();

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

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

Answer: "Compiler error: Undefined label 'here' in function main". Labels have functions scope, in other words The scope of the labels is limited to functions . The label 'here' is available in function fun() Hence it is not visible in function main.

•

```
#include <stdio.h>
int main()
{
    static char names[5][20]={"pascal","ada","cobol","fortran","perl"};
    int i;
    char *t;

    t      = names[3];
    names[3] = names[4];
    names[4] = t;
    for (i=0;i<=4;i++)
    {
        printf("%s",names[i]);
    }
}
```

Answer: "Compiler error: Lvalue required in function main". Array names are pointer constants. So it cannot be modified.

-

```
#include <stdio.h>

int main()
{
    int i=1,j=2;
    switch(i)
    {
        case 1: printf("GOOD");
                break;
        case j: printf("BAD");
                break;
    }
}
```

Answer: "Compiler Error: Constant expression required in function main." The case statement can have only constant expressions (this implies that we cannot use variable names directly so an error). Note that enumerated types can be used in case statements.

-

```
int main()
{
    int i;
    printf("%d",scanf("%d",&i)); // value 10 is given as input here
    return(0);
}
```

Answer: "1". Scanf returns number of items successfully read. Here 10 is given as input which should have been scanned successfully. So number of items read is 1.

-

```
#define f(g,g2) g##g2

int main()
{
    int var12=100;
    printf("%d",f(var,12));
    return(0);
}
```

Answer: "100".

-

```
int main()
{
    extern int i;
    i=20;
    printf("%d",sizeof(i));
    return(0);
}
```

Answer: "Linker error: undefined symbol '_i'.". The extern declaration specifies that the variable i is defined somewhere else. The compiler passes the external variable to be resolved by the linker. So compiler doesn't find an error. During linking the linker searches for the definition of i. Since it is not found the linker flags an error.

-

```
#include <stdio.h>
int main()
{
    extern out;
    printf("%d", out);
    return(0);
}
int out=100;
```

Answer: "100". This is the correct way of writing the previous program.

-

```
int i,j;
for(i=0;i<=10;i++)
{
    j+=5;
    assert(i<5);
}
```

Answer: "Runtime error: Abnormal program termination. assert failed (i<5), <file name>,<line number>". The assert() statements are used during debugging to make sure that certain conditions are satisfied. If assertion fails, the program will terminate reporting the same. After debugging, use #undef NDEBUG and this will disable all the assertions from the source code. Assertion is a good debugging tool to make use of.

-

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

Answer: "Runtime error : Stack overflow.". The main function calls itself again and again. Each time the function is called its return address is stored in the call stack. Since there is no condition to terminate the function call, the call stack overflows at runtime. So it terminates the program and results in an error.

-

```

int main()
{
    char *cptr,c;
    void *vptr,v;
    c=10;
    v=0;
    cptr=&c;
    vptr=&v;
    printf("%c%v",c,v);
    return(0);
}

```

Answer: "Compiler error (at line number 4): size of v is Unknown." You can create a variable of type void * but not of type void, since void is an empty type. In the second line you are creating variable vptr of type void * and v of type void hence an error.

•

```

#include <stdio.h>
int i=10;
int main()
{
    extern int i;
    {
        int i=20;
        {
            const volatile unsigned i=30;
            printf("%d",i);
        }
        printf("%d",i);
    }
    printf("%d",i);
    return(0);
}

```

Answer: "30,20,10". The '{' introduces new block and thus new scope. In the innermost block i is declared as, const volatile unsigned which is a valid declaration. i is assumed of type int. So printf prints 30. In the next block, i has value 20 and so printf prints 20. In the outermost block, i is declared as extern, so no storage space is allocated for it. After compilation is over the linker resolves it to global variable i (since it is the only variable visible there). So it prints i's value as 10.

•

```

int main()
{
    const int i=4;
    float j;
    j = ++i;
    printf("%d %f", i,++j);
}

```


Answer: "Compiler error". Here, i is a constant. you cannot change the value of constant

•

```
void aaa()
{
    printf("hi");
}

void bbb()
{
    printf("hello");
}

void ccc()
{
    printf("bye");
}

int main()
{
    int (*ptr[3]) ();
    ptr[0]=aaa;
    ptr[1]=bbb;
    ptr[2]=ccc;
    ptr[2] ();
}
```

Answer: "bye". Here, ptr is array of pointers to functions of return type int. ptr[0] is assigned to address of the function aaa. Similarly ptr[1] and ptr[2] for bbb and ccc respectively. ptr[2]() is in effect of writing ccc(), since ptr[2] points to ccc.

•

```
int main()
{
    int * j;
    void fun(int **);
    fun(&j);
    return(0);
}

void fun(int **k)
{
    int a =0;
    /* add a stmt here*/
}
```

Answer : "&k = &a;". The argument of the function is a pointer to a pointer. In the following pgm add a stmt in the function fun such that the address of 'a' gets stored in 'j'.

•

```
int abc(int a, float b)
{
    /* some code */
}

int abc(a,b)
int a; float b;
{
    /* some code*/
}
```

Answer: The first one is called the "ANSI C notation" and the second one the "Kernighan & Ritchie notation".

•

```
int main()
{
    static int i=5;
    if(--i)
    {
        main();
        printf("%d ",i);
    }
}
```

Answer: "0 0 0 0". The variable "I" is declared as static, hence memory for I will be allocated for only once, as it encounters the statement. The function main() will be called recursively unless I becomes equal to 0, and since main() is recursively called, so the value of static I ie., 0 will be printed every time the control is returned.

•

```
int main()
{
    void *v;
    int integer=2;
    int *i=&integer;
    v=i;
    printf("%d", (int*) *v);
    return(0);
}
```

Answer: "Compiler Error. We cannot apply indirection on type void*.". Void pointer is a generic pointer type. No pointer arithmetic can be done on it. Void pointers are normally used for, 1. Passing generic pointers to functions and returning such pointers. 2. As a intermediate pointer type. 3. Used when the exact pointer type will be known at a later point of time.

•

```
int main()
{
    float f=5,g=10;
    enum{i=10,j=20,k=50};
    printf("%d\n",++k);
    printf("%f\n",f<<2);
    printf("%lf\n",f%g);
    printf("%lf\n",fmod(f,g));
    return(0);
}
```

Answer: "Line no 5: Error: Lvalue required, Line no 6: Cannot apply leftshift to float, Line no 7: Cannot apply mod to float". Enumeration constants cannot be modified, so you cannot apply ++. Bit-wise operators and % operators cannot be applied on float values. fmod() is to find the modulus values for floats as % operator is for ints.

•

```
int main()
{
    while (strcmp(?some?,?some\0?))
    {
        printf(?Strings are not equal\n?);
    }
}
```

Answer: "No output". Ending the string constant with \0 explicitly makes no difference. So ?some? and ?some\0? are equivalent. So, strcmp returns 0 (false) hence breaking out of the while loop.

•

```
int main()
{
    char str1[] = {?s?,?o?,?m?,?e?};
    char str2[] = {?s?,?o?,?m?,?e?,?\0?};
    while (strcmp(str1,str2))
    {
        printf(?Strings are not equal\n?);
    }
    return(0);
}
```

Answer: ?Strings are not equal?, ?Strings are not equal?, ?. If a string constant is initialized explicitly with characters, ?\0? is not appended automatically to the string. Since str1 doesn't have null termination, it treats whatever the values that are in the following positions as part of the string until it randomly reaches a ?\0?. So str1 and str2 are not the same, hence the result.

-

```
int main()
{
    float i=1.5;
    switch(i)
    {
        case 1: printf("1");
        case 2: printf("2");
        default : printf("0");
    }
    return(0);
}
```

Answer: "Compiler Error: switch expression not integral". Switch statements can be applied only to integral types.