1. **Addition of two number without using arithmetic operator.**

int Add(int x, int y)

{

while (y != 0)

{

int carry = x & y;

x = x ^ y;

y = carry << 1;

}

return x;

}

printf("\n Addition value = %d", Add(15, 8));

**OUTPUT :- 23**

int a =10,b=36;

while(b>0)

{

a++;

b--;

}

printf("After addition : %d",a);

**OUTPUT :- 46**

1. **Freeing a dynamically allocated memory without using free system call.**

int \*p = (int \*)malloc(20\*sizeof(int));

\*(p+5) = 188;

printf ("\nSize of allocated memory = %d",sizeof(\*p));

printf ("\nValue at 5th = %d",\*(p+5));

p = (int \*)realloc(p,0);// Here memory are free

//printf ("\nValue at 5th = %d",\*(p+5)); // If you access the memory

, get a runtime error

printf ("\nSize of allocated memory after free = %d",sizeof(\*p));

1. **Program with no structure padding.**

#pragmapack(push) /\* push current alignment to stack \*/

#pragmapack(1) /\* set alignment to 1 byte boundary \*/

structMyPackedData

{

char Data1;

long Data2;

char Data3;

};

#pragma pack(pop)

structMyPackedData a;

printf("Size of Structure %d",sizeof(a));

**OUTPUT :- 6**

1. **How to avoid structure padding using pragma?**

The default compiler alignment is of 4 bytes. We need to change it to 1 byte.

For that, do the following steps:

* Push the current compiler alignment into the stack.
* Set the alignment into 1 byte.
* Declare the structure.
* Restore the default compiler alignment from the stack.

1. **Program with Volatile qualifier.**

constvolatileint local = 10;

int \*ptr = (int\*) &local;

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

\*ptr = 100;

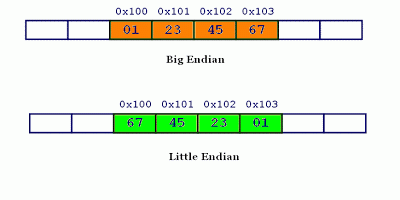
printf("\nModified value of local: %d \n", local);

/\* Here the value will be changed to 100 \*/

**OUTPUT :- 10 , 100**

1. **Big endian and Little endian determination program**

Little and big endian are two ways of storing multibyte data-types ( int, float, etc). In little endian machines, last byte of binary representation of the multibyte data-type is stored first. On the other hand, in big endian machines, first byte of binary representation of the multibyte data-type is stored first.  
  
Suppose integer is stored as 4 bytes (For those who are using DOS based compilers such as C++ 3.0 , integer is 2 bytes) then a variable x with value 0×01234567 will be stored as following.

[](http://4.bp.blogspot.com/_IEmaCFe3y9g/SO3GGEF4UkI/AAAAAAAAAAc/z7waF2Lwg0s/s1600-h/lb.GIF)

Memory representation of integer ox01234567 inside Big and little endian machines

unsignedinti = 1;

char \*c = (char\*)&i;

if (\*c)

printf("Little endian");

else

printf("Big endian");

1. **Programme to reverse a single linked with each k nodes.**
2. **Print leftmost and right most element in a binary tree.**
3. **How will you contruct your own memory management.**
4. **How to sort a stack elements.**

// C program to sort a stack using recursion

#include <stdio.h>

#include <stdlib.h>

// Stack is represented using linked list

struct stack

{

int data;

struct stack \*next;

};

// Utility function to initialize stack

void initStack(struct stack \*\*s)

{

\*s = NULL;

}

// Utility function to chcek if stack is empty

int isEmpty(struct stack \*s)

{

if (s == NULL)

return 1;

return 0;

}

// Utility function to push an item to stack

void push(struct stack \*\*s, int x)

{

struct stack \*p = (struct stack \*)malloc(sizeof(\*p));

if (p == NULL)

{

fprintf(stderr, "Memory allocation failed.\n");

return;

}

p->data = x;

p->next = \*s;

\*s = p;

}

// Utility function to remove an item from stack

int pop(struct stack \*\*s)

{

int x;

struct stack \*temp;

x = (\*s)->data;

temp = \*s;

(\*s) = (\*s)->next;

free(temp);

return x;

}

// Function to find top item

int top(struct stack \*s)

{

return (s->data);

}

// Recursive function to insert an item x in sorted way

void sortedInsert(struct stack \*\*s, int x)

{

// Base case: Either stack is empty or newly inserted

// item is greater than top (more than all existing)

if (isEmpty(\*s) || x > top(\*s))

{

push(s, x);

return;

}

// If top is greater, remove the top item and recur

int temp = pop(s);

sortedInsert(s, x);

// Put back the top item removed earlier

push(s, temp);

}

// Function to sort stack

void sortStack(struct stack \*\*s)

{

// If stack is not empty

if (!isEmpty(\*s))

{

// Remove the top item

int x = pop(s);

// Sort remaining stack

sortStack(s);

// Push the top item back in sorted stack

sortedInsert(s, x);

}

}

// Utility function to print contents of stack

void printStack(struct stack \*s)

{

while (s)

{

printf("%d ", s->data);

s = s->next;

}

printf("\n");

}

// Driver Program

int main(void)

{

struct stack \*top;

initStack(&top);

push(&top, 30);

push(&top, -5);

push(&top, 18);

push(&top, 14);

push(&top, -3);

printf("Stack elements before sorting:\n");

printStack(top);

sortStack(&top);

printf("\n\n");

printf("Stack elements after sorting:\n");

printStack(top);

return 0;

}

1. **How to find out the number of 1s in a binary number.**
2. **Swap the left most elements with rightmost elements in a binary tree.**

**C program to reverse an array:**

#include<stdio.h>

/\* Function to reverse arr[] from start to end\*/

void rvereseArray(int arr[], int start, int end)

{

    int temp;

    while (start < end)

    {

        temp = arr[start];

        arr[start] = arr[end];

        arr[end] = temp;

        start++;

        end--;

    }

}

/\* Utility that prints out an array on a line \*/

void printArray(int arr[], int size)

{

  int i;

  for (i=0; i < size; i++)

    printf("%d ", arr[i]);

  printf("\n");

}

/\* Driver function to test above functions \*/

int main()

{

    int arr[] = {1, 2, 3, 4, 5, 6};

    printArray(arr, 6);

    rvereseArray(arr, 0, 5);

    printf("Reversed array is \n");

    printArray(arr, 6);

    return 0;

}

**Find the middle of a given linked list in C:**

**Method 1:**  
Traverse the whole linked list and count the no. of nodes. Now traverse the list again till count/2 and return the node at count/2.

**Method 2:**  
Traverse linked list using two pointers. Move one pointer by one and other pointer by two. When the fast pointer reaches end slow pointer will reach middle of the linked list.

#include<stdio.h>

#include<stdlib.h>

/\* Link list node \*/

struct node

{

    int data;

    struct node\* next;

};

/\* Function to get the middle of the linked list\*/

void printMiddle(struct node \*head)

{

    struct node \*slow\_ptr = head;

    struct node \*fast\_ptr = head;

    if (head!=NULL)

    {

        while (fast\_ptr != NULL && fast\_ptr->next != NULL)

        {

            fast\_ptr = fast\_ptr->next->next;

            slow\_ptr = slow\_ptr->next;

        }

        printf("The middle element is [%d]\n\n", slow\_ptr->data);

    }

}

void push(struct node\*\* head\_ref, int new\_data)

{

    /\* allocate node \*/

    struct node\* new\_node =

        (struct node\*) malloc(sizeof(struct node));

    /\* put in the data  \*/

    new\_node->data  = new\_data;

    /\* link the old list off the new node \*/

    new\_node->next = (\*head\_ref);

    /\* move the head to point to the new node \*/

    (\*head\_ref)    = new\_node;

}

// A utility function to print a given linked list

void printList(struct node \*ptr)

{

    while (ptr != NULL)

    {

        printf("%d->", ptr->data);

        ptr = ptr->next;

    }

    printf("NULL\n");

}

/\* Drier program to test above function\*/

int main()

{

    /\* Start with the empty list \*/

    struct node\* head = NULL;

    int i;

    for (i=5; i>0; i--)

    {

        push(&head, i);

        printList(head);

        printMiddle(head);

    }

    return 0;

}

# Pairwise swap elements of a given linked list:

/\* Function to pairwise swap elements of a linked list \*/

void pairWiseSwap(struct node \*head)

{

    struct node \*temp = head;

    /\* Traverse further only if there are at-least two nodes left \*/

    while (temp != NULL && temp->next != NULL)

    {

        /\* Swap data of node with its next node's data \*/

        swap(&temp->data, &temp->next->data);

        /\* Move temp by 2 for the next pair \*/

        temp = temp->next->next;

    }

}

/\* Recursive function to pairwise swap elements of a linked list \*/

void pairWiseSwap(struct node \*head)

{

  /\* There must be at-least two nodes in the list \*/

  if (head != NULL && head->next != NULL)

  {

      /\* Swap the node's data with data of next node \*/

      swap(&head->data, &head->next->data);

      /\* Call pairWiseSwap() for rest of the list \*/

      pairWiseSwap(head->next->next);

  }

}

# How to write your own sizeof operator:

|  |
| --- |
| #define my\_sizeof(type) (char \*)(&type+1)-(char\*)(&type) Merge a linked list into another linked list at alternate positions: // Main function that inserts nodes of linked list q into p at  // alternate positions. Since head of first list never changes  // and head of second list  may change, we need single pointer  // for first list and double pointer for second list.  void merge(struct node \*p, struct node \*\*q)  {       struct node \*p\_curr = p, \*q\_curr = \*q;       struct node \*p\_next, \*q\_next;         // While therre are avialable positions in p       while (p\_curr != NULL && q\_curr != NULL)       {           // Save next pointers           p\_next = p\_curr->next;           q\_next = q\_curr->next;             // Make q\_curr as next of p\_curr           q\_curr->next = p\_next;  // Change next pointer of q\_curr           p\_curr->next = q\_curr;  // Change next pointer of p\_curr             // Update current pointers for next iteration           p\_curr = p\_next;           q\_curr = q\_next;      }        \*q = q\_curr; // Update head pointer of second list  } |

# Given a linked list which is sorted, how will you insert in sorted way:

1) If Linked list is empty then make the node as head and return it.

2) If value of the node to be inserted is smaller than value of head node

then insert the node at start and make it head.

3) In a loop, find the appropriate node after which the input node (let 9) is

to be inserted. To find the appropriate node start from head, keep moving

until you reach a node GN (10 in the below diagram) who's value is

greater than the input node. The node just before GN is the appropriate

node (7).

4) Insert the node (9) after the appropriate node (7) found in step 3.

/\* function to insert a new\_node in a list. Note that this

  function expects a pointer to head\_ref as this can modify the

  head of the input linked list (similar to push())\*/

void sortedInsert(struct node\*\* head\_ref, struct node\* new\_node)

{

    struct node\* current;

    /\* Special case for the head end \*/

    if (\*head\_ref == NULL || (\*head\_ref)->data >= new\_node->data)

    {

        new\_node->next = \*head\_ref;

        \*head\_ref = new\_node;

    }

    else

    {

        /\* Locate the node before the point of insertion \*/

        current = \*head\_ref;

        while (current->next!=NULL &&

               current->next->data < new\_node->data)

        {

            current = current->next;

        }

        new\_node->next = current->next;

        current->next = new\_node;

    }

}

# Why function pointer:-

# What is a Callbak function? In simple terms, a Callback function is one that is not called explicitly by the programmer. Instead, there is some mechanism that continually waits for events to occur, and it will call selected functions in response to particular events. This mechanism is typically used when a operation(function) can take long time for execution and the caller of the function does not want to wait till the operation is complete, but does wish to be intimated of the outcome of the operation. Typically, Callback functions help implement such an *asynchronous mechanism*, wherein the caller registers to get inimated about the result of the time consuming processing and continuous other operations while at a later point of time, the caller gets informed of the result.