

**Tutorial****Stack, Queue, Linked List, Double Liked List**

**-Write a function identical that return true if two stacks are identical and false otherwise.**

```
#include<iostream.h>
#include<stdlib.h>
typedef int ItemType;
const int MAX_ITEMS=10;
class StackType
{
public:
    StackType();
    void push(ItemType newitem);
    ItemType pop();
    bool IsEmpty();
    bool IsFull();
    int topStack();
private:
    int top;
    ItemType items[MAX_ITEMS];
};
StackType::StackType()
{
    top=-1;
}
int StackType::topStack()
{return top;}
bool StackType::IsEmpty()
{ return(top==-1);}

bool StackType::IsFull(void)
{ return(top==MAX_ITEMS-1);}

void StackType::push(ItemType newitem)
{
    if(IsFull())
    { cout<<"empty"; exit(0);}
    top++;
    items[top]=newitem;
}

ItemType StackType::pop()
{
    if(IsEmpty())
        { cout<<"empty"; exit(0);}
    ItemType x=items[top];
    top--;
    return x;
}
```

```

}
bool identical(StackType s1,StackType s2)
{
    bool ident=1;
    if(s1.topStack()!=s2.topStack())
        ident=0;
    else
        while(! s1.IsEmpty())
            if(s1.pop()!=s2.pop())
                { ident=0;
                  break;}
return ident;
}
void main()
{
    StackType s1,s2;
    ItemType symbol;

    for(int i=0;i<5;i++)
        {
            cin>>symbol;
            s1.push(symbol);}
    cout<<"_____";
        for( i=0;i<5;i++)
            { cin>>symbol;
              s2.push(symbol);}
    bool ident;
    ident=identical(s1,s2);
        if(ident==1)
            cout<<"identical";
    else
        cout<<"not identical";
}

```

**Write a program that uses a stack to test for balanced bracket pairs. The input strings all consisting of a single line, less than 80 characters long, will include four types of bracket:**

( ), { }, [ ], < >

**In order for expression to be parenthesized properly, each left bracket must be matched with a right bracket at the same type. For example { A [ B < C > < D > ( E ) F ] ( G ) } is correct, but { A [ B } ] is not**

```

#include<iostream.h>
#include<stdlib.h>
typedef char ItemType;
const int MAX_ITEMS=10;
class StackType
{
public:

```

```
StackType();
void push(ItemType newitem);
ItemType pop();
bool IsEmpty();
bool IsFull();
private:
    int top;
    ItemType items[MAX_ITEMS];
};
StackType::StackType()
{
    top=-1;
}
bool StackType::IsEmpty()
{ return(top==-1);}

bool StackType::IsFull(void)
{ return(top==MAX_ITEMS);}

void StackType::push(ItemType newitem)
{
    if(IsFull())
    { cout<<"empty"; exit(0);}
    top++;
    items[top]=newitem;
}

ItemType StackType::pop()
{
    if(IsEmpty())
        { cout<<"empty"; exit(0);}
    ItemType x=items[top];
    top--;
    return x;
}

bool IsOpen(char symbol)
{ cout<<"*****";
  if( (symbol=='(') || (symbol=='[') || (symbol=='{') )
    return true;
  else
    return false;
}

bool IsClosed(char symbol)
{
    if( (symbol==')' ) || (symbol==']') || (symbol=='}') )
    return true;
    else
    return false;
}
```

```
}

bool matches(char openSymbol,char symbol)
{
return((openSymbol=='(' && symbol==')')
|| (openSymbol=='[' && symbol==']')
|| (openSymbol=='{' && symbol=='}')
);
}

void main()
{ ItemType symbol;
  StackType stack;

  bool balance=true;
  char openSymbol;
  cout<<"enter Expression";
  cin>>symbol;
  while(symbol!='\n' && balance)
  {
    if(IsOpen(symbol))
      stack.push(symbol);

    else if(IsClosed(symbol))
    {
      if(stack.IsEmpty())
        balance=false;
      else
      {
        openSymbol=stack.pop();
        balance=matches(openSymbol,symbol);
      }
    }
    cout<<balance;
  }

  cin>>symbol;

  if(balance)

    cout<<" Expression is Well Formed"<<endl;

  else
    cout<<" Expression is not Well Formed"<<endl;

}

// palinodrome is astring that reads the same forward as backward
```

```
#include<iostream.h>
#include<stdlib.h>
const int size=10;

typedef char ItemType;
const int MAX_ITEMS=10;
class StackType
{
public:
    StackType();
    void push(ItemType newitem);
    ItemType pop();
    bool IsEmpty();
    bool IsFull();
private:
    int top;
    ItemType items[MAX_ITEMS];
};
StackType::StackType()
{
    top=-1;
}
bool StackType::IsEmpty()
{ return(top==-1);}

bool StackType::IsFull(void)
{ return(top==MAX_ITEMS-1);}

void StackType::push(ItemType newitem)
{
    if(IsFull())
    { cout<<"empty"; exit(0);}
    top++;
    items[top]=newitem;
}

ItemType StackType::pop()
{
    if(IsEmpty())
    { cout<<"empty"; exit(0);}
    ItemType x=items[top];
    top--;
    return x;
}
class QueueType
{
public:
    QueueType ();
    void Enqueue(ItemType item);
```

```
    ItemType Dequeue() ;
    bool IsFull();
        bool IsEmpty();

private:
    int f,r;
    ItemType q[size];
};

QueueType::QueueType()
{ r=f=-1;}

bool QueueType::IsFull()
{ return( (r==size-1));}

bool QueueType::IsEmpty()
{ return( (f==-1));}

void QueueType::Enqueue(ItemType item)
{
    if (IsFull())
    {
        cout<<"error.....the Queue is full";
        exit(0);
    }
    else
    {
        if(f==-1)
            f=f+1;

        r=r+1;
        q[r]=item;
    }
}

ItemType QueueType::Dequeue()
{
    int item;
    if( IsEmpty())
    {
        cout<<"Error.....the is empty";
        exit(0);
    }
    else
    {
        item=q[f];
```

```

        f=f+1;
    }
    if(f > r)
    {
        f=-1;
        r=-1;
    }
return item;}

void main()
{
    QueueType queue;
    StackType stack;
    ItemType x,y;
    bool palinodrome=true;
    char str[20];
    cin>>str;
    for(int i=0;str[i]!='\0';i++)
    {
        stack.push(str[i]);
        queue.Enqueue(str[i]);
    }
while(palinodrome && ! stack.IsEmpty())
{
    x=stack.pop();
    y=queue.Dequeue();
    if(x!=y)
        palinodrome=false;
}
if ( ! palinodrome)
cout<<" not palinodrome";
else
cout<<" palinodrome";
}

```

**Implement splitlist as a member function of list which divides list into two list according to the key of item.**

```

#include<iostream.h>
#include<assert.h>
typedef int ItemType;
// class definitions with six member functions
class List {
public:
    List(){start=0;tail=0;current=0;}
    void creatlist(ItemType & elem);
    void Display(void);

```

```
void SplitList(List &l1, List &l2, ItemType elem);

private:
    struct Node;
    typedef Node *Link;
    struct Node{
        ItemType elem;
        Link next;    };
    public:
    Link start;
    ;
};

void List::creatlist(ItemType &elem)
{
    Link addedNode=new Node;
    assert(addedNode);
    addedNode->elem=elem;
    if(start==0)
        start=addedNode;
    else
        tail->next=addedNode;
    tail=addedNode;
    tail->next=0;
}

void List::SplitList( List &l1, List &l2, ItemType elem)
{
    Link p=start;
    Link ptr;
    while(p->elem !=elem)
    {ptr=p;
    p=p->next;}

    l1.start=p;
    l2.start=start;
    ptr->next=0;
}

main()
{
    List l1,l2;
    ItemType x,elem;
    for(int i=0;i<8;i++)
    {
        cin>>x;
        l1.creatlist(x);
    }
    cout<<"enter element you want";
    cin>>elem;
    l1.Display();
}
```

```
cout<<"-----" <<endl;
l.SplitList(l1,l2,elem);
l1.Display();
cout<<"-----" <<endl;
l2.Display();
return 0;
}

#include<iostream.h>
#include<assert.h>
typedef int ListElementType;
class List {
public:
    List();
    void insert(ListElementType elem);
    ListElementType Delete(void);
    ListElementType DeleteAnyPos(int pos);
    bool first(ListElementType elem);
    bool next(ListElementType elem);
    bool previous(ListElementType &elem);
    void printlist(void);
private:
    struct NodeType {
        ListElementType elem;
        NodeType* next;
        NodeType * back;
    };
    NodeType* head;
    NodeType* current;
};
List::List()
{
    head=0;
    current=0;
}
void List::printlist(void)
{
    NodeType *ptr;
    assert(head);
    ptr=head;
    while(ptr !=0)
    {
        cout<<ptr->elem<<" ";

        ptr=ptr->next;
    }
}
bool List::previous(ListElementType &elem)
{

```

```
        assert(current);
        if(current->back==0)
            return false;
        else
        {
            current=current->back;
            elem=current->elem;
            return true;
        }
    }
void List:: insert(ListElementType elem)
{
    NodeType *addedNode= new NodeType;
    assert(addedNode);
    addedNode->elem=elem;
    addedNode->next=head;
    if(head)
        head->back=addedNode;
    addedNode->back=0;
    head=addedNode;
}
ListElementType List::Delete(void)
{
    ListElementType x;
    NodeType *ptr;
    ptr=head;
    assert(head);
    if(head->next==0)
        head=0;
    else
    {
        head=head->next;
        head->back=0;
    }
    x=ptr->elem;
    delete ptr;
    return x;
}
ListElementType List::DeleteAnyPos(int pos)
{
    ListElementType x;
    NodeType *ptr;
    ptr=head;
    assert(head);
    int i=0;
    while(i<pos )
        {ptr=ptr->next;i++;}
    if(ptr->next !=NULL) // if the deleted not at the end of list
    {
```

```
ptr->back->next=ptr->next;
ptr->next->back=ptr->back;
}
else // if deleted node at last of list
{   ptr->back->next=0;
    ptr->back=0;
}
x=ptr->elem;
delete ptr;
return x;
}
void main()
{
List l;
l.insert(1);
l.insert(2);
l.insert(3);
l.insert(4);
l.printlist();cout<<endl;
l.Delete();
l.printlist();cout<<endl;
l.DeleteAnyPos(3);
l.printlist();
}
```