```
        Node *head;
public:
        LinkedList() // default constructor
        { head = NULL; }
       ~LinkedList(); // prototype
        void appendNode(T); // "
        void insertNode(T); // "
        void insertAtPos(T, X); // "
        void deleteNode(T);   // "
        void display() const; // "
}; //end class

template <class T>
void LinkedList<T>::appendNode(T value)
{   . . .
}
template <class T>
void LinkedList<T>::display()
{   . . .
}
```

Creating objects in main: `LinkedList<double> list1;` (default constructor)

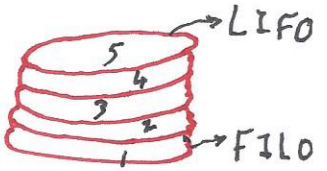
Examples:

```
Circle<int> C1(8);
Circle<double> C2(6.15);
Rectangle<int> Rect1(5, 9);   // 5 -> w, 9 -> L
```

Week 8, Sat.

Stack Data structure: A stack is a data structure that stores and retrieves data in a **LIFO** (Last In First Out) manner.
It is a linear data structure. Like array or linked list, it holds a sequence of elements. Unlike arrays and list, stacks are LIFO structures.

Examples:
- like stack of plates in a party
- stack of books in library



Computer System uses stack while executing programs.

When a func. is called, they save the program's return address on stack, they create local variables on stack. When func. terminates, local variables are removed.

Two Types of stack: 1- Static (using arrays)
2- Dynamic (using Linked Lists)

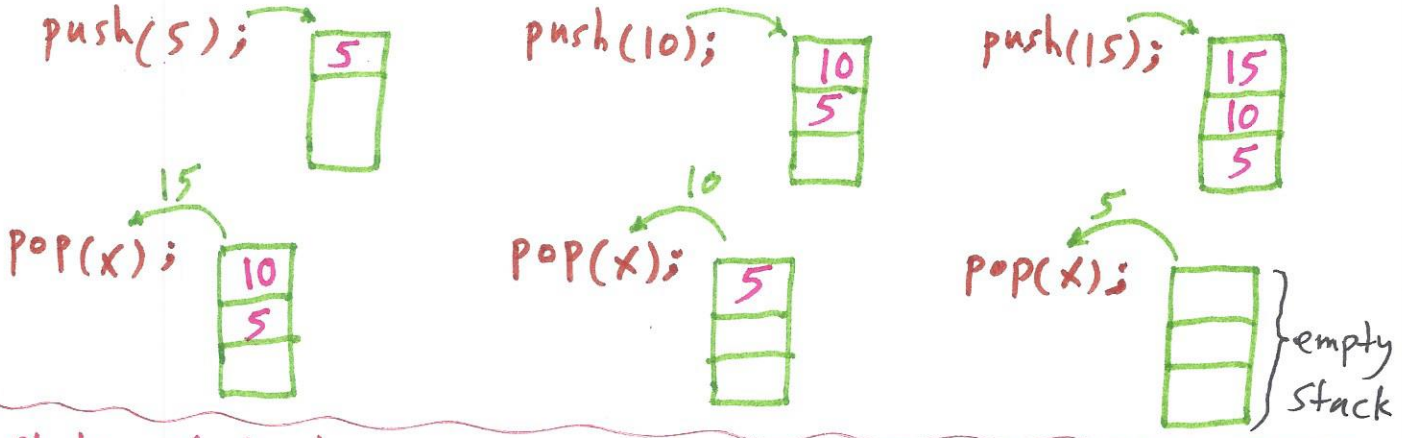
1- Static stacks have fixed size and are implemented as arrays.

2- Dynamic stacks can grow and shrink in size as needed and are implemented as linked list.

Stack Operations: 1- push 2- pop 3- isEmpty 4- isFull
5- display elements

1- push : A value to be stored 2- pop : Remove a value from stack

e.g.



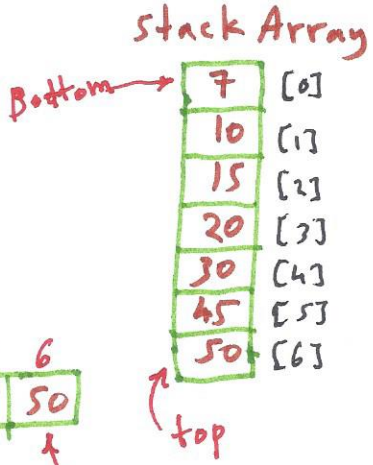
```
// A static stack class
class IntStack
{  private:  int *stackArray;
            int size; // size of stack
            int top; // top of stack or array index
   public:  IntStack(int); // Constructor prototype
            IntStack(const IntStack &); // Copy Constructor prototype
            ~IntStack(); // destructor
            // Operations
            void display(); // prototype
            void push(int); //  "
            void pop(int&); //  "
            bool isFull() const; // "
```

```
        bool isEmpty() const; // "
        int peek() const; // "
};// end class
IntStack::IntStack(int sz) // Constructor
{   stackArray=new int[sz]; // Dynamic Memory Allocation
    size=sz;
    top=-1; // index for array and -1 means stack is empty
}
// push func.
Void IntStack::push(int num)
{   if (isFull())
      { cout << "The stack is full." << endl; }
    else { top++;
           stackArray[top]=num;
         }
}//end push
// display func.
void IntStack::display()
{  int t=top;
   while (t>=0)
   { cout << stackArray[t] << endl;
     t--;
   }
}//end display
// Destructor
IntStack::~IntStack()
{       }
```

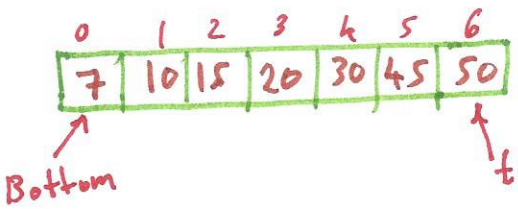
or `stackArray[++top]=num;`

pre-decrement

first increment top by 1 then use it.



For empty stack:
top -1
size = 7
top 0 After inserting 7



```
// pop member func.
Void IntStack :: pop (int &num)
{   if (isEmpty())
       { cout << "The stack is empty. \n"; }
    else
       { num = stackArray [top];
         top--;
       }
} //end pop
```

post-decrement
post fix

→ or `num = stackArray [top--];`

{ Use value of top first then decrement by 1.

```
// isFull ( ) func.
bool IntStack :: isFull ( ) const
{   if (top == size-1)
          return true;
    else  return false;
} //end pop
```

→ or `return ( top == size-1);`

```
// isEmpty ( ) func.
bool IntStack :: isEmpty ( ) const
{     if (top == -1)
          return true;
      else  return false;
} //end isEmpty()
```

or `return ( top == -1 );`

```
// peek ( ) func. returns the top element without removing it.
int IntStack :: peek ( ) const
{   if ( isEmpty())
        {   cout<<"Stack is empty." <<endl;
            return -1;
        }
    else
        return stackArray [top];
} //end peek()
```

// Copy Constructor, assigns an existing object to a new object.

```
IntStack :: IntStack ( const IntStack &obj )
{   // Create stactArray
    if ( obj.size > 0 )
            stackArray = new int [obj.size];
     else   stackArray = NULL;
    // Copy the size variable
    size = obj.size;
    // Copy the stack Contents
    for ( int count=0; count < size; count++)
    {   stackArray [count] = obj.stackArray [count];   // obj3 = obj1;
    }
     top = obj.top ;
} // end copy constructor
// Driver for IntStack class
int main ( )
{   IntStack st1 ( 5 ); // using constructor
    int x;
    st1.push ( 7);
     st1. push ( 10);
     st1.push (4 );
     Cout << "Top element is: " << st1.peek ( ) <<endl; //4
    Cout << st1. pop(x) << "poped from stack." <<endl;
    st1.display ( );
     IntStack st2 (5);
     st2 = st1; // using Copy Constructor
     st2.display ( );
} //end main
```

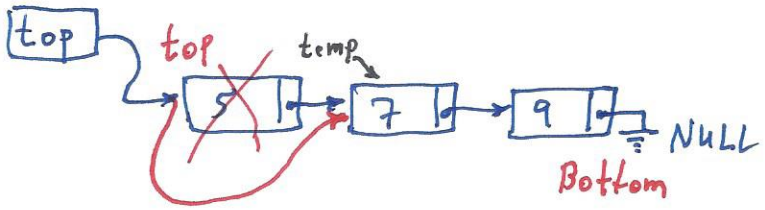
Dynamic Stack: A stack may be implemented as a linked list, can grow and shrink in size, with each push and pop operation.

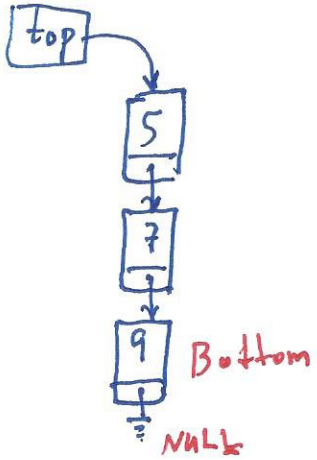
Two Advantages: 1- No need for size

2- Will never be full as long as we have free memory.

```
// Dynamic stack class
Class DynStack
{   private:   Struct Node
               { int value;
                 Node *next;
               };
   Node *top; // top is as same as head pointer in linked list
   public:
          // default constructor
          DynStack ( )
          { top = NULL;} // or using nullptr instead NULL
         ~DynStack( ); // destructor
          // stack operations
          Void push (int); // prototype
          void pop (int&); //  "
          bool isEmpty( ); //  "
          void display ( ); //  "
};//end class DynStack
// pop func.
Void Dynstack:: pop (int &num)
{  Node *temp;
   if ( isEmpty( ))
        { cout<< "The stack is empty.\n"; }
   else
        { num = top-> value; // stores 5 in num
          temp = top->next;
```

```
        delete top;
        top = temp;
      }
  } //end pop func.
```



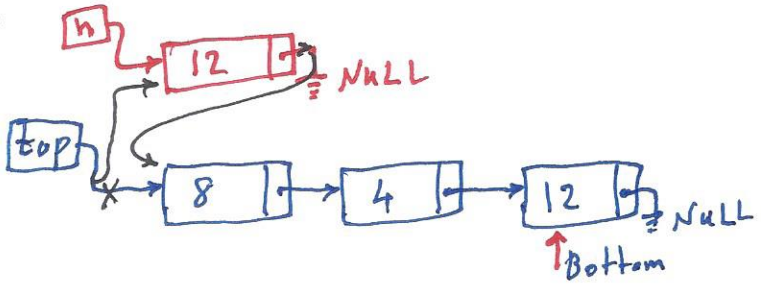


// isEmpty func.

```
bool DynStack:: isEmpty()
{  if (top==NULL)
       return true;
   else return false;
 } //end isEmpty()
```

// push func.

```
void DynStack:: push(int num)        12
{  Node *n = new Node;
   n->value = num;
   n->next = NULL;
   if(isEmpty())
     { top = n; }
   else { n->next = top;
          top = n;
        }
} //end push
```



// display func.

```
void DynStack:: display()
{  Node *curr;
   curr = top;
   while( curr != NULL)
   { cout << curr->value << " ";
     curr = curr->next;
   }
} // end display func.
```