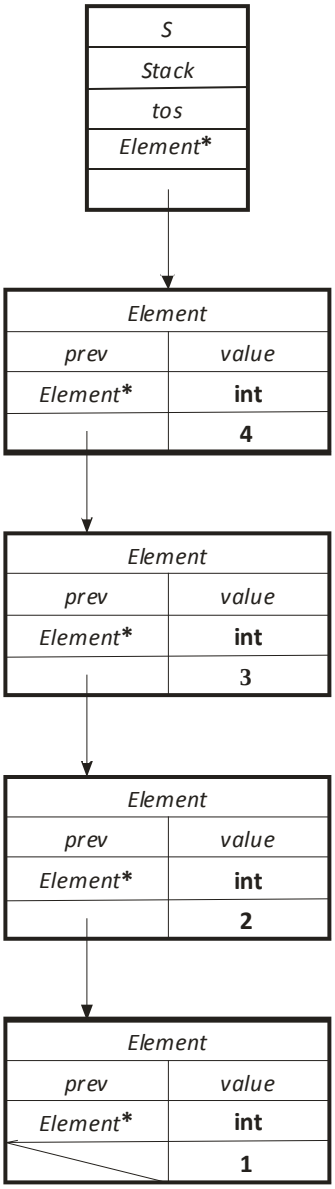


**Figure 1.** Element implementation of a Stack

The stack illustrated in figure 1 is created by dynamically allocating storage for each element. Integers are stored in the elements in figure 1. Integers are stored on the stack as they were pushed on the stack. The integer value 1 was pushed on the stack first followed by integers 2, 3 and 4. Note that element on top of the stack contains the last integer pushed on the stack. A stack is a Last-In-First-Out data structure. The last element inserted on to a stack is the first element to be removed.

**Stack Element:** A stack element contains storage for the data in the stack and a pointer to the previous element.

**class** *Element* is a private implementation for a stack element.

```
class Element {
    Element* prev;
    int value;
public:
    Element(Element* p,int v):prev(p),value(v){}
    Element* Prev(void){return prev;}
    int Value(void){return value;}
};
```

**Figure 2. class** *Element.*

| *Element* | |
|---|---|
| *prev* | *value* |
| *Element**| **int** |
| | |

**Figure 3.** Stack Element

```
struct StackException {
        StackException(char* m)
        {   cout << endl << "I am the Stack and I am " << m << "." << endl;
        }
    };
class Stack {
    struct Element {
        Element* prev;
        int value;
        Element(Element* p,int v):prev(p),value(v){}
    };
    Element* tos;
    void Kill(Element* e);
public:
    Stack();
    ~Stack();
    bool IsFull(void);
    bool IsEmpty(void);
    void Push(int v);
    int Pop(void);
};
```

**Figure 4. class** Stack

1. Member data in **class** *Element*.
   1.1. *Element* ***prev*****;** Member *prev* points the previous element put on the stack. For example, consider the sequence of statements:
   *Stack S***; *S*.*Push*(1); *S*.*Push*(2);**
   Member *prev* in the stack element containing the integer value **2** points to the stack element containing the integer value **1**.
   1.2. **int** *value***;** Member *value* stores integer data placed on the stack.
2. Member functions of **struct** *Element*.
   2.1. *Stack*::*Element*::*Element*(*Element** *p*,**int** *v*);
   Consructor *Element* assigns initial values to private members of **class** *Element*. Parameter *p* is assigned to member *prev* and parameter *v* is assigned to member *value*.
3. Member data in **class** *Stack*.
   3.1. *Element** *tos***;** Member tos points to the element on top of the stack. If the stack is empty, the value of member *tos* is zero (**0**) or null.
4. Member functions of **class** *Stack*.
   4.1. *Stack*::*Stack* **():***tos***(0){}**
   Consructor *Stack* assigns a null-value to private member *tos*. Member *tos* anchors a list of elements that are used to implement the stack.
   4.2. *Stack*::**~***Stack* **(){***Kill*(*tos*);**}**
   Destructor **~***Stack* directs the process of reclaiming storage for stack elements. Member *tos* points to the element on top of the stack. Storage is reclaimed by traversing the list of elements beginning with the element on top of the stack.
   4.3. **void** *Stack*::*Kill* **(***Element** *e***) { … }**
   Function Kill removes elements on the stack and reclaims their storage. The key concept for removal is that an element cannot be removed until its predecessor is removed. Removing predecessors can be removed by walking the list of elements linked by member *prev*.
   4.4. **bool** *Stack*::*IsFull* **(void){…}**
   Member function *IsFull* should return **true** when storage cannot be obtained for a new element. However, that event is very unlikely and the computer system will very likely exhibit other serious difficulties. We assume, therefore, that the stack is never full.
   4.5. **bool** *Stack*::*IsEmpty* **(void){…}**
   Member function *IsEmpty* returns **true** when the stack is empty. The stack is empty when member *tos* points to no elements. Member *tos* has the integer value zero when it is null.
   4.6. **void** *Stack*::*Push* **(int** *v***)**

```
{   if (IsFull()) throw StackException("Full");
    Element* n=new Element(tos,v);
    tos=n;
}
```

   Function Push determines if the Stack is full. If the stack is full an exception is thrown because no more elements can be put on the Stack. If the Stack has space for at least one more element, a new element is created, linked to the previous element, and member *tos* is assigned to point to the newest element.
   4.7. **int** *Stack*::*Pop* **(void){ … }**
   Function *Pop* removes the element on top of the stack and returns the value stored in the element. Local variable *n* is assigned to the value of member *tos*.. Storage for local variable *v* is allocated. The value of *Element*::*value* is assigned to local variable *v*.

Member *tos* is assigned to the penultimate element. Storage for the element that was formerly the element on top of the stack is reclaimed. Storage for local variables *n* and *v* is reclaimed as function *Pop* returns. The value of variable *v* is returned.

```
struct StackException {
    StackException(char* m)
    {   cout << endl << "I am the Stack and I am " << m << "." << endl;
    }
};

class Stack {
    struct Element {
        Element* prev;
        int value;
        Element(Element* p,int v):prev(p),value(v){}
    };
    Element* tos;
    void Kill(Element* e)
    {   Element* e=tos;
        while (e) {
            Element* p=e;
            e=e->prev;
            delete p;
        }
    }
public:
    Stack():tos(0){}
    ~Stack(){Kill(tos);tos=0;}
    bool IsFull(void){return 0;}
    bool IsEmpty(void){return tos==0;}
    void Push(int v)
    {   if (IsFull()) throw StackException("full");
        Element* e=new Element(tos,v);
        tos=e;
    }
    int Pop(void)
    {   If (IsEmpty()) throw StackException("empty");
        Element* e=tos;
        int v=e->value;
        tos=e->prev;
        delete e;
        return v;
    }
};
```

**Figure 5. class** Stack member function implementation