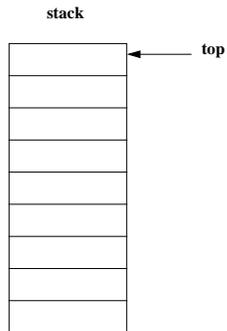


## Building A Simple Stack Abstraction

A stack is an abstraction for a list of “things” of some type. New things are **pushed** onto the top of the stack. Things may only be taken off the stack, by **popping** from the top. So, a stack is an abstraction for a *Last-In, First-Out* (LIFO) data structure. A stack is **empty** when there is nothing to pop from the stack.



What happens when the following stack operations are performed?

```
push x
push y
pop
pop
push z
pop
pop
```

8/5/00

1 of 8

## Stack Class Methods

**Designing the stack interface involves deciding what methods we need, their names, and type signatures. You need to think about how other people besides yourself would use your stack class to create stack objects that will hold integers.**

So, we need methods to push and pop from the stack, and it will also be useful to have methods to test if the stack is empty, and full if we are to implement a fixed size stack.

```
void push(int x);
int pop();
bool empty();
bool full();
```

You should also think about how these will be used. You will want to be able to use a stack object like this:

```
Stack stack;
stack.push(5);
int x = stack.pop();// x should be 5

int y = stack.pop();// should be an error, since stack is empty
...
if (!stack.full())
    stack.push(10);
```

We should also realize that a user may want to create a stack object globally, or on the heap, so we need to keep that in mind when we start to do the implementation.

8/5/00

3 of 8

## A Class Abstraction for a Stack

Assume that we want a stack of integers. How do we implement a class abstraction for a stack?

1. We have to make a design decision about what methods go in the interface, their arguments lists, and their return types. We can defer worrying about the implementation initially.
2. We have to make a design decision about the internal representation for the stack. That is, do we use a fixed size array and implement a fixed size stack or do we use a linked list and implement a variable size stack.
3. We have then write the implementation for each of the methods, and test to make sure they work as advertised!
4. We also want to take into consideration good object-oriented design practice and use C++ correctly, and make our stack abstraction as useful as possible. This means we employ what we know about encapsulation, information hiding, good programming style, and some attention given to efficiency of the implementation.

8/5/00

2 of 8

## Integer Stack Class Interface

```
/* IntStack.h */

class IntStack {
public:
    // constructors and destructors
    IntStack ();
    IntStack (int x);
    IntStack (const IntStack& s);

    ~IntStack ();

    // boolean predicates (truth functions)
    bool empty() const;
    bool full() const;

    // the stack operations
    void push (int x);
    int pop();

    // don't forget the assignment operator!
    IntStack& operator=(const IntStack& s);

    // We might want to allow for input and output too
    friend ostream& operator>> (ostream& os, const IntStack& s);
    friend ostream& operator<< (ostream& os, const IntStack& s);

private:
    ... // we do this next
};
```

8/5/00

4 of 8

## Integer Stack Data Representation

We have to make a design decision about how the stack is to be implemented. To make things simple at first, we can just restrict the `IntStack` to be a fixed size stack, of say 100 integers. This means we can just define a simple array of integers to represent the internal data structure of the `IntStack` class. We also need a "top" member variable that always tells us where the top of the stack is.

```
class IntStack {
public:
    ... // as before

private:
    int list[100];
    int top;
};
```

**So, now what we need to do is implement all the class methods, and decide which ones to make inline member functions and which not.**

## Integer Stack Method Implementation

How do we implement the copy constructor and the assignment operator?

```
/* IntStack.C */

IntStack::IntStack(const IntStack& s)
{
    top = s.top;

    for (int i = 0; i < top; i++)
        list[i] = s.list[i];
}

IntStack&
IntStack::operator= (const IntStack& s)
{
    if (this != &s) {
        top = s.top;

        for (int i = 0; i < top; i++)
            list[i] = s.list[i];
    }

    return *this;
}
```

## Integer Stack Method Implementation

Now that we know what the internal representation will be, we can go back to the method interface and provide an implementation for each of the methods, some inline, others not inline.

We do the constructors and destructors first.

```
class IntStack {
public:
    IntStack () : top (0) { }
    IntStack (int x) : { list[top] = x; top++; }

    IntStack (const IntStack& s);

    ~IntStack () { /* do nothing */ }

    bool empty () const { return (top == 0 ? true : false); }
    bool full () const { return (top < 100 ? false : true); }

    void push (int x) {
        if (!full())
            list[top++] = x;
        else
            cerr << "Error: stack overflow!!" << endl;
    }

    int pop () {
        if (!empty()) {
            top--;
            return list[top];
        }
        else
            /* stack underflow error, what do we do? */
            cerr << "Error: stack underflow!!" << endl;
    }
    ...
};
```

## Integer Stack Friend Function Implementation

All we have left to do is implement the friend input/output operators in `IntStack.C`

```
ostream&
operator<< (ostream& os, const IntStack& s)
{
    if (s.empty())
        os << "Stack is empty" << endl;
    else {
        os << "Stack contains:" << endl;

        for (int i = top; i > 0; i--)
            os << s.list[i] << endl;
    }

    return os;
}

istream
operator>> (istream& is, const IntStack& s)
{
    int x;
    while (is >> x) {
        if (!full())
            s.push(x);
        else {
            cerr << "Error: stack overflow!" << endl;
            break;
        }
    }

    return is;
}
```

The input operator is a bit more complicated because you need to validate the input to make sure that you read a valid integer before inserting into the stack. Assume here that all input is valid (which you cannot assume in practice)