Declaring the public interface for a linked list int_stack (Stacks/mydefs2.h)

```
//typedef enum {false, true} Boolean;
#define Boolean int

class int_stack {
    class node {
    public:
        int data;
        node* next;
    };
    node* new_node (int d, node* n);
    node* top;
public:
    // Note that the stack is implicit
    void init (int Nitems = 100);
    void push (int x);
    int pop ();
    Boolean isempty();
    Boolean isfull();
};
```

*   The public part of the interface looks the same as for the array version. Good information hiding.

Rewrite main (Stack/stackmain2.C)

```
#include <iostream.h>
#include "mydefs2.h"      // for the linked-list definition

main()
{
    int_stack s1,s2;
    int i;

    s1.init(500);
    s2.init();

    for( i = 0; i < 20; i++ ) {// add 21 items to s1
        s1.push(i);
    }

    for( i = 0; i < 20; i++ ) { // remove 21 items
        cout << s1.pop() << ' ';
    }
    cout << endl;

    s2.push(10);
    i = s2.pop();
    assert (s2.isempty());   // confirm empty stack
}
```

* Only one character different in the #include!

* s1 has been tested with 21 pops.
* It correctly detects the empty stack, s2.

A linked list implementation of stack (Stack/stack2.C)

```
#include "mydefs2.h"
#include <assert.h>

// Linked list implementation of stacks.

int_stack :: node* int_stack :: new_node(int d, node* n)
{
    node* t = new node;
    assert( t );        // space found OK
    t -> data = d;
    t -> next = n;
    return( t );        // return pointer to stack
}

void int_stack :: init(int Nitems = 100)
{
    top = 0;        // Note Nitems not used
}
void int_stack :: push(int x)
{
    top = new_node(x, top);
}
int int_stack :: pop()
{
    assert( !isempty() );
    int t = top -> data;
    node* oldtop = top;
    top = top -> next;
    delete oldtop;        // frees item pointed to
    return t;
}
Boolean int_stack :: isempty()
{
    return( top == 0 );
}
Boolean int_stack :: isfull()
{
    return( 0 );          // candidate inline function?
}
```

* free store available checked
* Nitems is completely ignored by init
* delete t works like free(t)

Implicit Inline functions

```
#include <assert.h>
typedef enum {false, true} Boolean;

class int_stack {
   int sz;
   int top;
   int* stack;
public:
   // Note the following are implicit inline
   void init(int Nitems = 100) { sz = Nitems;
                stack = new int[sz]; top = sz; }
   void push(int x) { assert( !isfull() );
                       stack[--top] = x; }
   int pop() { assert( !isempty() );
                return( stack[top++] ); }
   Boolean isempty() { return( top == sz ); }
   Boolean isfull() { return( top == 0 ); }
};

#include "mydefs2a.h"
#include <assert.h>

// Linked list implementation of stacks.

int_stack:: node* int_stack:: new_node(int d, node* n)
{
   node* t = new node;
   assert( t );
   t -> data = d;
   t -> next = n;
   return( t );
}
int int_stack :: pop()
{
   assert( !isempty() );
   int temp = top -> data;
   node* oldtop = top;
   top = top -> next;
   delete oldtop;
   return temp;
```

```
}
```

Overloading Stack functions

Suppose we have a stack of integers and a stack of chars, but we would like to share such operators as push, pop, isempty and isfull. First let's look at the two (char and int) array-stacks for the similarities.

```
#include <assert.h>
typedef enum {false, true} Boolean;

class int_stack {
   int sz;
   int top;
   int* stack;
public:
   void init(int Nitems = 100) { sz = Nitems;
               stack = new int[sz]; top = sz; }
   void push(int x) { assert( !isfull() );
                             stack[--top] = x; }
   int pop() { assert( !isempty() );
                        return( stack[top++] ); }
   Boolean isempty() { return( top == sz ); }
   Boolean isfull() { return( top == 0 ); }
};
class char_stack {    // Changed
   int sz;
   int top;
   char* stack;       // Changed
public:
   // init, push, and pop are changed
   void init(int Nitems = 100) { sz = Nitems;
              stack = new char[sz]; top = sz; }
   void push(char x) { assert( !isfull() );
                             stack[--top] = x; }
   char pop() { assert( !isempty() );
                        return( stack[top++] ); }
   Boolean isempty() { return( top == sz ); }
   Boolean isfull() { return( top == 0 ); }
```

```
};

#include <iostream.h>
#include "mydefs1b.h"

main()
{
   int_stack s1;        // a stack of integers
   char_stack s3;       // a stack of chars
   int i;

   s1.init(500);
   s3.init();           // default size of 100

   for( i = 0; i < 20; i++ ) {
      s1.push(i);       // also push(s1, i);
   }
   for( i = 0; i < 20; i++ ) {
      cout << s1.pop() << ' ';
   }
   cout << endl;

   for( i = 0; i < 20; i++ ) {
      s3.push('a' + i);  // overloading push
   }
   for( i = 0; i < 20; i++ ) {
      cout << s3.pop() << ' ';
   }
   cout << endl;
}
```

The C++ compiler resolves which push and pop operators to use, based on the number and type of the parameters.

Constructors and Destructors

If your abstract data types are going to be really "abstract" they need special support. When you declare an array, or a structure, or an integer in a function, C++ allocates the memory for you when the function is called. Similarly, when the function returns, C++ deallocates the memory. You too can allocate and deallocate memory.

Ideally we want to be able to say

```
{
   int_stack s1;    // allocate space for stack s1
   ...              // use the stack s1
}
// storage for s1 has been deallocated
```

- and not have to do the
  ```
  s1.init();
  ```

- But still want to be able to say
  ```
  s1.init(500);
  ```

- which allocates a stack of non-default size.

- We would also like to be able do things like
  ```
  int_stack array_of_stacks[10];
  ```

- Or even
  ```
  int_stack* ps;      // pointer to a stack
  ps = new int_stack;
        // allocate space for the stack
  ...                 // use the stack
  dispose ps;         // deallocate the stack
  ```

C++ uses special member functions based on the name of the class. For `int_stack` the constructor would be called

`int_stack :: int_stack`

and the destructor would be called

`int_stack :: ~int_stack`

See ~tony/web_docs/C201/ Stacks/stack1.C

```
#include <assert.h>
//typedef enum {false, true} Boolean;
#define Boolean int

class int_stack {
    int sz;
    int top;
    int* stack;
public:
    int_stack(int Nitems = 100)     // The creators
        { sz = Nitems; stack = new int[sz]; top = sz; }
    ~int_stack() { delete[] stack; }
                    // delete[] deletes arrays
    void push(int x) { assert( !isfull() );
                            stack[--top] = x; }
    int pop() { assert( !isempty() );
                    return( stack[top++] ); }
    Boolean isempty() { return( top == sz ); }
    Boolean isfull() { return( top == 0 ); }
};
```

See Stacks/stack2.C

```
#include <iostream.h>
#include "mydefs1c.h"

main()
{               // creator called 5 times for s4
    int_stack s1(500), s2, s4[5], *s5;
    int i;

    s5 = new int_stack;        // get stack space
    for( i = 0; i < 20; i++ ) {
        s1.push(i);            // 500 element stack
    }

    for( i = 0; i < 20; i++ ) {
        cout << s1.pop() << ' ';
    }
    cout << endl;

    s2.push(10);            // default stack
    cout <<  s2.pop() << ' ';
    s4[4].push(11);         // 4th stack in the array
    cout << s4[4].pop() << ' ';
    s5->push(12);           // pointing to a stack
    cout << s5->pop() << endl;
    delete s5;              // explicit removal
}
```

*   Using just one constructor as before say,

`int_stack( int Nitems = 100 ),`

works for everything, but for the array declaration

`int_stack ... s4[5].`

the default constructor `int_stack()` is called 5 times.
See Stacks/stackmain1c.C for more complete example

*   There are three equivalent ways of declaring a stack of non-default size:

```
int_stack s1(500);      // or
int_stack s1 = 500;        // or
```

```
int_stack s1 = {500};
```

* Here are some other equivalent declarations:

```
char s[] = "hello";       // or
char s[] = {"hello"}; // or
char* s = "hello";    // or
char* s = {"hello"};

int i = 5;        // or
int i(5);         // OK, logical, but g++ says not
int i = {5};
```

new and delete vs. malloc() and free()

If T is some type in your program (char, int, double, or some struct or class) and

```
T* tp, *ta;
```

then

```
tp = new T;          // equivalent to:

tp = (T*) malloc (sizeof (T));
```

Note the need for explicit type conversion here.

* Similarly,

```
ta = new T[n];

ta = (T*) malloc (n * sizeof(T));
```

are equivalent, except that the default constructor for T,
T :: T()
is called n times if it exists

* the size of the resulting array is saved "somewhere" for use by delete[]

```
*    ta = new T[0];
```
also returns a pointer to an object of type T

* when no storage is available, new returns NULL, so a quick and dirty way of checking is:

```
ta = new T[n];  assert( ta );       //  or even
assert( ta = new T[n] );
```

delete vs. free()

Continuing our example from above, delete tp; is equivalent to free(tp);

* delete[] tp;
is not defined when tp is not allocated with the array syntax

* deleting parts of arrays or other proper subsets of the memory returned by new is not defined

Friends

A function that is a friend of class X has access to all of X's private components. It is particularly useful for the I/O operators >>; and <<; which work on istream&; and ostream&; objects and so cannot be member functions of your class. It is also useful for efficient access. Let's revisit our linked list implementation of int_stack, and see how simple it becomes:

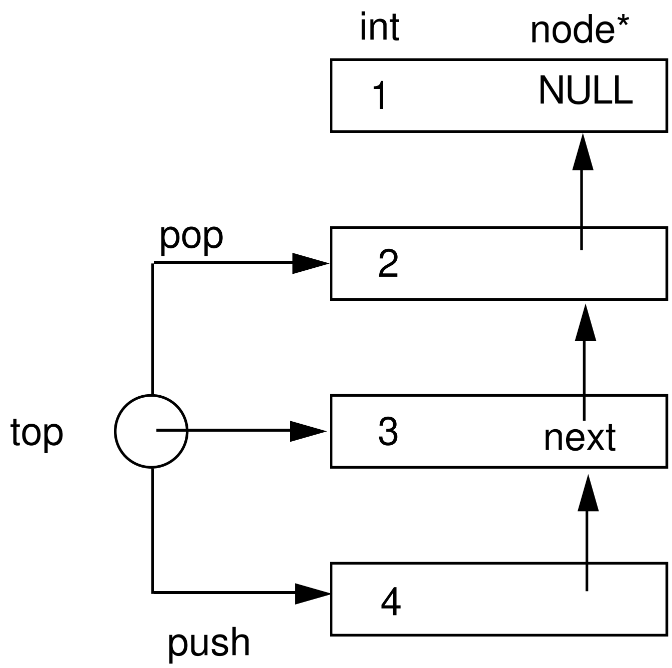
/////   this is Stacks/mydefs2b.h //////

```
//typedef enum {false, true} Boolean;
#define Boolean int
#include <new.h>

class node {
    friend class int_stack;
    int data;
    node* next;
    node(int d, node* n);     // constructor
};

class int_stack {
    node* top;
public:
    void init(int Nitems = 100) { top = NULL; }
    void push(int x) { top = new node(x, top); }
    int pop();          // Not defined here, later
    Boolean isempty() { return( top == NULL ); }
    Boolean isfull() { return( NULL ); }   // Never Full
};
```

////// this is Stacks/stack2b.C

```
#include "mydefs2b.h"
#include <assert.h>
// Linked list implementation of stacks.

node :: node(int d, node* n)
{
    data = d;
    next = n;
}
int int_stack :: pop()
{
    assert( !isempty() );
    int temp = top -> data;       // get data
    node* oldtop = top;     // point to old
    top = top -> next;          // unlink old
    delete oldtop;              // discard
    return temp;
```

```
}
```



Linked Stack showing effect of push() and pop() operations

```
#include "mydefs2b.h"
#include <assert.h>
// Linked list implementation of stacks.

node :: node(int d, node* n)
{
    data = d;
    next = n;
}

int int_stack :: pop()
{
    assert( !isempty() );
    int temp = top -> data;      // get data
    node* oldtop = top;    // point to old
    top = top -> next;        // unlink old
    delete oldtop;            // discard
    return temp;
}

void int_stack :: push(int x) {
    top = new node(x, top);
```

```
}
```