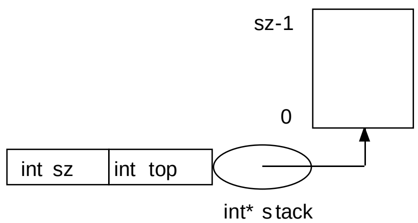
Inline functions

```
#include <assert.h>
class Istack {
      int sz;
      int top;
      int* stack;
public:
      // Note the following are inline functions
      void initialize (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++] );
      }
      bool isempty( )
            { return( top == sz ); }
      bool isfull( )
            { return( top == 0 ); }
};
```



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 highlight the dissimilarities between the two (char and int) array-stacks.

```
#include <assert.h>
class Istack {
    int sz;
    int top;
    int* stack;
public:
    void initialize (int Nitems = 100) {
        sz = Nitems;
        delete [ ] stack;
        stack = new int[sz];
        top = sz;
    }
    void push(int x) {
        assert( !isfull( ) );
        stack[--top] = x;
    }
    int pop( ) {
        assert( !isempty( ) );
        return( stack[top++] );
    }
    bool isempty( )
        { return( top == sz ); }
    bool isfull( )
        { return( top == 0 ); }
// C++ provides a default constructor and a destuctor
};
```



```
class Cstack {    // Changed
    int sz;
    int top;
    char* stack;      // Changed
public:
          // initialize, push, and pop are changed
    void initialize (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++] );
    }
    bool isempty( )
        { return( top == sz ); }
    bool isfull( )
        { return( top == 0 ); }
};
```

The default constructor creates int sz; int top; char* stack; but does not give them values. Here the member (access) function initialize() does the job.


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

int main( )
{
    Istack s1;     // a stack of integers
    Cstack s3;     // a stack of chars
    int i;

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

    for( i = 0; i < 20; i++ ) {
        s1.push(i);     // in C, 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;
    return 0;
}
```

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.

A constructor and destructor makes it possible to say

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

- and not have to do the
  `s1.initialize( );`
- But may want to use
  `s1.initialize(500);`
  to build new stack of different size.
- With a constructor we can do things like
  `Istack array_of_stacks[10];`
- An even

```
    Istack* ps;         // pointer to a stack
    ps = new Istack;
            // allocate space for the stack
    ...                     // use the stack
    delete ps;          // deallocate the stack
```


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

Istack :: Istack

and the destructor would be called

Istack :: ~Istack

```
#include <assert.h>

class Istack {
    int sz;
    int top;
    int* stack;
public:
    Istack(int Nitems = 100)   {   // Two constructors
        sz = Nitems; stack = new int[sz]; top = sz;
    }
    ~Istack( )
        { delete[ ] stack; }
                // delete[ ] "frees" an array
    void push(int x) {
        assert( !isfull( ) ); stack[--top] = x;
    }
    int pop( ) {
        assert( !isempty( ) ); return( stack[top++] );
    }
    bool isempty( )
        { return( top == sz ); }
    bool isfull( )
        { return( top == 0 ); }
};
```


//See Stacks/stack2.cc

```
#include <iostream>       // modern form for <iostream.h>
#include "mydefs1c.h"

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

    s5 = new Istack;        // 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
    return 0;
}
```

- The constructor has two forms,
  `Istack( int Nitems = 100 ),`
  one encompasses the default (not paramters) constructor.
- For the array declaration
  `Istack ... s4[5],`
  the default constructor Istack( ) is called 5 times.

Public interface for a linked list Istack (Stacks/mydefs2.h)

```
class Istack {
    class node {
      public:
        int data;
        node* next;
    };
    node* new_node (int d, node* n);
    node* top;
  public:
    // Note these are all prototypes
    void initialize (int Nitems = 100);
    void push (int x);
    int pop ( );
    bool isempty( );
    bool isfull( );
};
```

- The public part of the interface looks much like the array version. Good information hiding.

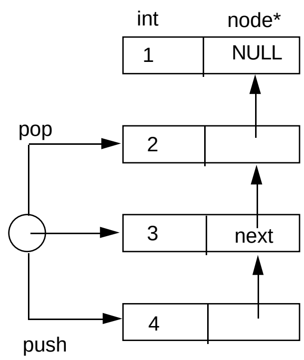
Now we will see

- No need for "new space" available check
- Nitems is completely ignored by initialize
- delete t works like free(t)

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

// Linked list implementation of stacks.

Istack :: node* Istack :: new_node(int d, node* n) {
    node* t = new node;
    // new fails if space not found
    t -> data = d;
    t -> next = n;
    return( t );        // return pointer to stack
}

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



```
#include "mydefs2a.h"
#include <assert.h>
// Linked list implementation of stacks.

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



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

Overloading Stack functions


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

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

    s1.initialize(500);
    s2.initialize( );

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

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

    s2.push(10);
    i = s2.pop( );
    assert (s2.isempty( ));
    // confirm empty stack
    return 0;
}
```

- Only one character different in the #include!
- s1 has been tested with 20 pops.
- It correctly detects the empty stack, s2.

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

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

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

then

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

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

Note the need for explicit type conversion here.

- Similarly,

```
T* ta = new T[n];

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

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

- 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 "throws an exception", so there is no need for the assert() below:

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

delete vs. free( )

delete tp; is equivalent to free(tp);

- `delete[ ] tp;`
  is not defined if 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 (computer does not know what to do)

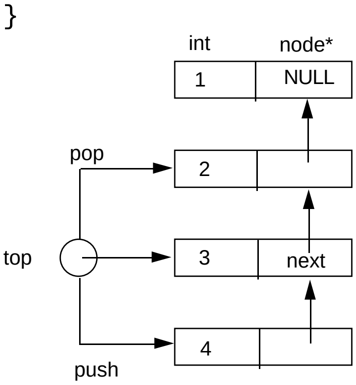
Friends

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 Istack, and see how simple it becomes:

```
/////        this is Stacks/mydefs2b.h     //////
#include <new>      // only if want to catch exception

class node {   // Note node now defined outside Istack
    friend class Istack;
    int data;
    node* next;
    node(int d, node* n);  // constructor  prototype
};
class Istack {
    node* top;
public:
    void initialize(int Nitems = 100)
        { top = NULL; }
    void push(int x)
        { top = new node(x, top); }
    int pop( );         // Not defined here,  later
    bool isempty( )
        { return( top == NULL ); }
    bool isfull( )
        { return( NULL ); }    // Never  Full
};
```



```
//////  this is Stacks/stack2b.cc
#include "mydefs2b.h"
#include <assert.h>
// Linked list implementation of stacks.

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


```
class node {   // Note node now defined outside Istack
    friend class Istack;
    friend ostream& operator << (ostream& , const Istack& ) ;
    int data;
    node* next;
    node(int d, node* n);  // constructor  prototype
};

class Istack {
    node* top;
    friend ostream& operator << (ostream& , const Istack& ) ;
public:
    void initialize(int Nitems = 100) ;
    void push(int x) ;
    int pop( ) ;
    bool isempty( ) ;
    bool isfull( ) ;
};
ostream& operator << (ostream&  out, const Istack& right)
{   node* tmp;
    for (tmp = right.top; tmp != NULL; tmp = tmp->next)
        out << tmp->data  << ' ';
    out << endl;
    return out;
};

int main () {
    Istack s1; int i;
    for (i=0; i < 20; i++)
        s1.push(i);
    cout << s1;        // using new new (overloaded) <<.
    return 0;
}
```