

C++

Classes in C++

Declarations

```
class Foo
{
  ...
};
```

It's a lot like **struct**. Don't forget the semicolon at the end.

Protection modes

public	anyone can access
protected	subclasses can access
private	only this class can access
friend	allows named outsiders to get at private parts

Constructors and destructors

Named same as class. Constructor always called when new object created, destructor called when object is destroyed.

```
#include <iostream.h>

class Foo
{
    public:
        Foo( int i ) : _i(i)
        { cout << "In constructor of
          object " << _i << endl; }

        ~Foo()
        { cout << "In destructor of
          object " << _i << endl; }

    private:
        int _i;
}; // don't forget this semicolon!
```

```

Foo f0( 0 );

void main()
{
    cout << "a" << endl;
    Foo f1( 1 );
    cout << "b" << endl;
    Foo f2( 2 );
    cout << "c" << endl;
    {
        cout << "d" << endl;
        Foo f3( 3 );
        cout << "e" << endl;
    }
    cout << "f" << endl;
    Foo f4( 4 );
    cout << "g" << endl;

    Foo *fp;
    cout << "h" << endl;
    fp = new Foo( 5 );
    cout << "i" << endl;
    delete fp;
    cout << "j" << endl;
}

```

Note that even if you don't write a constructor, a "default constructor" will be created for you. The

compiler also create a "default assignment operator" for you. We'll explain those more later.

Member functions

Methods look just like functions declared inside a class. They can access instance variables and do all the other things you'd expect.

Methods without side-effects should be declared **const**. This lets the compiler know you're allowed to call these functions even on a **const** object.

Methods can be defined inline (in the class body) or out of line (outside the class). The out-of-line way requires a little more syntax.

```
class Foo
{
    public:
        // inline definition
        int f() const
        { return 1; }

        int g() const;
};

// out-of-line definition
int Foo::g() const
{
```

```
    return 2;
}
```

Example: `class Stack`

Let's implement a `Stack` class using a linked-list. And let's go all-out OO and make `ListNode` a class, too, because we're gung-ho about C++.

```
#include <assert.h>
#include <bool.h>
#include <iostream.h>

class ListNode
{
public:
    ListNode( int data,
             ListNode* next )
        : _data(data), _next(next)
        {}

    int data() const
    { return _data; }

    ListNode* next() const
    { return _next; }

private:
    int _data;
```

```
};  
    ListNode* _next;
```

```
class Stack
{
    public:
        // constructor
        Stack() : head( NULL ) {}

        // modifier methods
        void push( int x )
        {
            head=new ListNode(x,head);
        }

        int pop()
        // cannot be empty!
        {
            assert( !empty() );
            int x = head->data();
            ListNode* temp = head;
            head = head->next();
            delete temp;
            return x;
        }
}
```

```
// accessor (const) methods
int size() const
{
    int count=0;
    for( ListNode* curr=head;
        curr!=NULL;
        curr=curr->next() )
        count++;
    return count;
}

int top() const
{
    assert( !empty() );
    return head->data();
}

bool empty() const
{
    return head==NULL;
}

private:
    ListNode* head;
};
```

```
void main()
{
    Stack s;
    s.push( 2 );
    s.push( 3 );

    int x( s.pop() );
    cout << x << endl;
    cout << s.size() << endl;

    // Wouldn't it be nice...
    Stack s2( s );
    s.pop();
    cout << s2.top() << endl;
    // Coredump.

    s.pop();
    // This would, too, but
    // friendly-like.
}
```