

Modular Programming

Encapsulation means that the clients of a class don't ever see the insides of a class. The technique I describe now insures that clients can easily find the class **interface** without having to separate it from the class **implementation**. Nothing forces you to program like this, but it's a good habit to learn.

To fix our ideas, suppose that we want to introduce a new data type called **istack** (stack of integers) A stack of integers is an entity onto which you can push integers values.), and off of which you can pop values.

We will use it like this:

```
istack myStack();           // Creates a new stack
myStack.pop(int i);        // To add a value
int j = myStack.pop();     // Top retrieve the last value.
myStack.print();          // Prints the stack to cout.
if (myStack.isEmpty())    // Test for an empty stack
```

We have various files that need to be written:

File structure:

Client code: example.cc
Header files: istack.hh (only declarations!)
Source files: istack.cc (only definitions)
Inline files : istack.icc (definitions for inline functions, if any)

Now usually, one builds a project out of more than one class, so under realistic circumstances stack would be one of several classes that the client code relies upon to perform its task (e.g, text editor, income tax program, track reconstruction, missile defense software...). These classes are put together into a library called a **class library**. The class library consists of a collection of header files and an archive (.a file) of objects files.

Usually, (on unix):

Type	extension	location
Header file	.hh, .h	/someplace/include
Source file	.cc, .C	/someplace/src (if available)
Archive	.a, .so	/someplace/lib
Client code	.cc, .C	anywhere

To tell the compiler where to find the **archives**, use the directive:

`-L/someplace/lib`

at the compile command. Several such commands may be given on the same line to specify a **library search path**.

To tell the compiler where to find the header files, use the directive:

`-I/someplace/include`

at the compile command. Several such commands may be given on the same line to specify a **library search path**.

The entire directory containing all the different parts of the project is called the **project directory**.

I have created a directory for you, with the skeleton of a C++ program in the proper places and a Makefile that will let you build the project. You are now ready to work on implementing your first class, for which the design has already been done.

The first exercise is a stack class.

Excercises. Follow these instructions to create a project directory containing source files, header files, client code, and a Makefile.

- * cd to your home directory.
- * copy the file `~student0/project.tar` into that directory.
- * type `tar xvf project.tar`
- * cd project

The project contains the header file `istack.hh`, the source code `istack.cc`, and a test program `example.cc` which uses the `stack` class. The `example.cc` program tests the `stack` class. However, all of the `stack` classes' member functions are empty. It's your job to implement them.

To build the project anyway (despite the fact that the unimplemented `stack` class won't work):

- * cd project
- * make
- * test/example

- * examine the file `istack.hh`

- * examine the file `example.cc`

- * examine the file `istack.cc`. This file will explain what each function is and what it should do.

- * Implement the constructor. Make it print a message so you know when it is called.

- * Implement the `push`, `pop`, `isEmpty`, and `print` functions.

- * Test the stack.

```

//-----//
//
// Header file for class istack
//
//-----//
#ifndef _istack_hh_
#include "bool.hh"

class istack {

public:

    // The constructor
    istack();

    // The push function
    void push (int value);

    // The pop function
    int pop();

    // Query for an empty stack
    bool isEmpty() const;

    // The print function
    void print() const;

private:

    const int _ISTACK_MAX=10000; // Maximum length of istack
    int      _count;           // Actual length of istack
    int      _data[_ISTACK_MAX]; // An array to store data

};

#endif

```

```

#include "istack.hh"
#include <iostream.h>
// The constructor is called every time an object
// is created. This insures that the object starts
// from a well defined state...without requiring
// client code to call istack:init().
istack::istack() {
    // ==> insert code here
}

// The push function increases the length of the istack
// by one element, adding the value to the end of the
// istack
void istack::push(int value) {
    // ==> insert code here
}

// The pop function decreases the length of the istack
// by one element, and returns the value that is popped
// off.
int istack::pop() {
    // ==> insert code here
}

// The isEmpty() query function returns true or false,
// depending on whether the stack is empty. This function
// should not change the state of the istack. It is
// therefore declared const;
bool istack::isEmpty() const {
    // ==> insert code here
}

// The print function dumps the entire istack to cout,
// in a readable format. This function should not change
// the state of the istack. It is therefore declared const.
void istack::print() const {
    // ==> insert code here
}

```

```

#include <iostream.h>
#include "istack.hh"
// Program to test stack. This will work (in principal)
// as soon as students have implemented the stack class..
// which means filling in the function hooks in
// src/istack.cc.
main() {

    // create a new stack called a.
    istack a;

    // fill it with powers of 2 through 128.
    a.push(1);
    a.push(2);
    a.push(4);
    a.push(8);
    a.push(32);
    a.push(64);
    a.push(128);

    // now print it out.
    a.print();

    // pop of a couple of values from the top of the stack.
    cout << "Now popping off..." << a.pop() << endl;
    cout << "Now popping off..." << a.pop() << endl;

    // what does the stack look like now??
    a.print();

    // now create another stack.
    istack b;
    while (!a.isEmpty()) {
        b.push(a.pop());
    }

    // now a should be empty....
    a.print();

    // and b should contain a, except in reverse order, right?
    b.print();

    // now let's just assign the stack b to a:
    a=b;

    // so finally, a should have the same contents as b.
    a.print();

    // Cool, huh?
}

```