Block 13

Book IV Chapter 2: Data Structures

Book IV Chapter 3: Constructors, Destructors, and Exceptions

Chapter 2: Data Structures

Data Structures takes up where Arrays leave off.

Arrays are HOMOgenious. An array can have ints, doubles, chars, any

type. BUT ALL THE ELEMENTS MUST BE THE SAME TYPE.

A struct is HETEROgenious. A struct can have many members, and

they can each be different from the other elements.

struct prs

{

bool deleted ;

char name[30] ;

int empno ;

int weapon ;

int vehicle ;

bool driver\_license ;

bool pilot\_license ;

bool ship\_license ;

bool sub\_license ;

bool english ;

bool russian ;

bool chinese ;

struct prs \* left ;

struct prs \* right ;

} A , \*P ;

This type is called a "struct prs". This code created one, named "A".

We'd use it like this:

A.deleted = false ;

strncpy (&A.name, "Jas Bond", 9) ;

A.empno = 7 ;

The "dot" or "member field" operator separates the struct from its field.

If we create another one:

P = new struct prs ;

if (P == NULL) exit(-1) ;

P->deleted = false ;

strncpy(&P->name, "Modesty Blaise", 14) ;

P->name = 23 ;

The "struct pointer" P "points to a field with the "arrow operator": ->

Structs were the last improvement in data structures before Object Oriented Programming.

They can do most things objects can, it just takes some thought, and pointers.

Legacy code is full of them.

Chapter 3: Constructors, Destructors, and Exceptions.

Recall classes. Classes should make structs seem familiar. The main difference

between classes and structs, is classes can contain functions ("methods").

(In fact, there are many differences, but you'll encounter them in

the coming years. There's not point in drilling you on inheritance,

virtualization, and all the assorted earwax of the New Religion before

you're an expert coder. You won't have enough context to remember it,

and you'll only have to learn it over again.)

When you instantiate an object, it executes its Constructor function.

Constructors are for running "startup housework" code.

class prs

{

bool deleted ;

string name ;

int empno ;

int weapon ;

int vehicle ;

bool driver\_license ;

bool pilot\_license ;

bool ship\_license ;

bool sub\_license ;

bool english ;

bool russian ;

bool chinese ;

prs \* left ;

prs \* right ;

prs () // no-arg constructor

{

deleted = false ;

name = "Vacant" ;

empno = 0 ;

weapon = vehicle = 0 ;

driver\_license =

pilot\_license =

ship\_license =

sub\_license =

english =

russian =

chinese = false ;

left = right = this ;

}

prs (string s, int num, int gun, int car) // no-arg constructor

{

deleted = false ;

name = s ;

empno = num ;

weapon = gun ;

vehicle = car ;

driver\_license =

pilot\_license =

ship\_license =

sub\_license =

english =

russian =

chinese = false ;

left = right = this ;

}

See? I'm using POLYMORPHISM to have two functions with the same

name. If the user programmer creates a prs without any args,

the first one runs. If she creates one with name, num, gun and car in

the argument list, it runs the second.

If you don't write a constructor, the compiler provides one which does

nothing.

Constructors have no type, not even void, and are named after the class.

When an object is garbage-collected, it runs its destructor.

~prs ()

{

cout << "Destroying agent: " << num << endl ;

}

Destructors are named after the class, preceeded by a tilde ~

Usually, they're used to delete or free() dynamically allocated

memory before the object referring to them goes away.

===================

Exceptions.

"Never write a program your user can kill. Nothing should ever knock

you off the air."

If you divide by zero, fail to read from a file, or any of a dozen common

faults, your program can die.

To keep that from happening, HANDLE the error by anticipating it.

try

{

// do the thing that can fail

}

catch (specific or all errors)

{

clean up, fix up, or just exit cleanly.

}

See the "exception programs" in this directory.

============================

BUT WHY??

Objects promise benefits during the programming process. Instead of having

data in one place, and code in another, we can "encapsulate" both into a

class declaration.

Imagine how much data and code there are in an object that contains

a character in an online role-playing game. If everything related to

a character is between two braces, there's less chance of something

getting lost or forgotten.

The other, equally important benefit is INHERITANCE.

If we only need to write the basics of a character once, the

inheritance process allows us to only write what is different in

it's inheritors.

Vehicles have many things in common: position, speed, color, fuel, and functions

like move, start, stop, turn, etc.

One those are done, we can "inherit" them and each derived class has them, so

we can concentrate on things like wheels vs. wings, and soforth.

Having everybody inherit from a standard parent class insures that all

the programmers are approaching similar tasks in similar ways, and the

code becomes more maintainable.

/\*

\* exception02.cpp - demonstrate try and catch

\*/

#include <iostream>

#include <stdexcept>

using namespace std ;

// specify which exception this function can throw:

void function3() throw (runtime\_error)

{

cout << "in function 3" << endl ;

throw runtime\_error( "runtime\_error in function3") ;

} // function 3

void function2() throw (runtime\_error)

{

cout << "function 3 is called inside function2" << endl ;

function3() ;

}

void function1() throw (runtime\_error)

{

cout << "function2 is called inside function1" << endl ;

function2() ;

}

int main(int argc, char \*argv[])

{

try {

cout << "function1 is called inside main" << endl ;

function1() ;

} // end try

catch(runtime\_error & error)

{

// the error object knows what and where it happened

cerr << "Exception occurred: " << error.what() << endl ;

cerr << "Exception handled in main" << endl ;

} // catch B

return 0 ;

} // main ends