Block 12

Book IV Chapter 1: Working with arrays, pointers and references.

SUMMARY

Operator Overloading: String and Array Objects.

Operator Overloading makes C++ much easier to use than C, and much harder

to understand when reading someone else's code.

If you have:

int age, pounds, inches ;

and you want to enter values into them, all you need do is:

cout << " Enter age, pounds, and inches, separate with space: " ;

cin >> age >> pounds << inches ;

The ">>" operator knows how to work with cin, and it looks at

the variable after it, and decides how many keystrokes to process for

each variable, and how.

You could create a struct:

struct prs

{

int num ;

char name[30] ;

int age ;

} agent ;

And, with a little programming, you could do this:

cout << "Enter number, agent name in quotes, and age: " ;

cin >> agent.num >> agent.name >> agent.age ;

Even though "struct prs" isn't known to the language, you can write

code that lets C++ use ">>" just like you would for ints, doubles, or

chars. This "overloads" the ">>" operator. Overloaded operators

have different blocks of code to call depending on the type of the

data they're given.

String class objects

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/cpp\_strings.txt">C++ Strings</A>

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/strings\_string\_objects">strings\_string\_objects</A>

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/overloading\_operators.cpp.txt">overloading\_operators</A>

Overloading is a sensitive subject. It has been abused

by the first generations of C++ programmers.

For instance, in C, we do almost everything with FUNCTION CALLS:

To tack a string constant on the end of an array of char

from another:

char s[] = "Another Challenge for the Green Hornet, " ;

char b[] = "his aid Kato, and his rolling arsenal the Black Beauty!" ;

char d[1024] ;

strncpy(d, s, sizeof(d)) ; // copy first Cstring to destination

strncat(d, b, sizeof(d)) ; // append the second Cstring

But in C++, the "=" and "+" operators, usually never used

with Cstring data, have been "overloaded" to indicate, when

placed between string objects, string assignment or

concatenation.

string s = "Another Challenge for the Green Hornet, " ;

string b = "his aid Kato, and his rolling arsenal the Black Beauty!" ;

string d = s + d ; // looks like arithmetic :(.

Generally, this just makes code harder to read, especially

in really large programs. Use it with care.

Generally, we overload the << operator when we want

to be able to print fields from a struct or class object

with a single line of code. In that context, it saves

us trouble.

(in the class):

friend ostream &operator<<( ostream &, prs\* ) ;

(outside the class):

ostream &operator<<( ostream &output, prs \*h)

{

output << (void \*) h->getFrom() << " "

<< (void \* )h << " "

<< (void \*) h->getNext() << "\t"

<< h->name << "\t"

<< h->num << endl ;

return output ;

}

Note that in the above, I want to print out three addresses

as "%p" ("pointers"), so I have to CAST them to (void \*)

addresses. Failure to do this will produce a recursive call,

where "<<" is trying to print out an object which it has

been overloaded to print in a certain way, which contains

an object for which it has been overloaded, which contains....

Think this one over for a sec. If you don't believe me,

write a program to exercise it both with and without

the (void \*) casts.

You'll get a screen full of endless junk. Try it.

(I'm not going to beat on overloading operators any more. I use them

only rarely, and I've programmed in C++ for years. They really are

mostly a way to save keystrokes, and saving keystrokes is NOT a priority.

In the C culture, more code generally means greater clarity. Some of

the most opaque programs only occupy one line.)

--

Converting between types.

Most of the code you will encounter will be like the example above:

we coerce one type to be another by enclosing 'another' in parens and

preceeding the object to be cast with the result.

double d = 3.14159 ;

int x ;

x = (int) d ;

C++ is a lot smarter than C, so it is rare that we need

to cast, compared to how often we needed to in legacy C.

On the other hand, if you're really "taking control" of

the computer, C++'s assumptions about what you want to do

can be very annoying. C is intended for systems work, C++

for applications. Systems programs need to "cheat", which

makes C dangerous for the unskilled.

Just learn the commoner string member functions like length(), insert()

begin(), end(), and erase().

Ignore "explicit" constructors. Wait till you've written

some hairy-er C++, then it'll be more relevant.

Read the examples on Operator Overloading, then forget

about it. In general operator overloading is overused

and treated as a toy.

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/overloading\_operators.cpp.txt">overloading\_operators</A>

Pointers make it possible to write assembler-style code, short and powerful.

If you have two arrays:

int a [500] , b [500] ;

.

.

.

//and you want to copy b to a:

for (int i = 0 ; i < 500 ; ++i)

b[i] = a[i] ;

Slow and inefficient.

But if you leave "Algebra Land" and directly control the computer hardware,

you can see some real efficiency:

memcpy (a, b, sizeof(int) \* 500) ;

Instead of a loop with all that overhead, a single C instruction,

which translates to a single machine instruction, that does the same

thing, copying raw bytes from one memory location to another.

REFERENCES:

References are unnecessary, since they're just an attempt to make things

safer. There is no safety. There are two kinds of people: those who

make up their minds to survive, and take responsibility for it, and those

who whine and plead and tantrum to get someone else to take care of them.

They violate the Prime Directive: if you want something done right,

do it yourself.

Programming is the same: if you write clean, efficient code using sharp

tools, YOU must make sure you test and validate the code. If you try

to find a language which is weaker, and makes it hard to write code that

might harm the system, it makes it equally hard to write efficient code.

COBOL can't really do much to the computer system, it's good at record

keeping, but to even do something as hardware-intensive as graphics,

it's weak. C can do anything the computer is capable of doing, but

there is no warning about overflow, null references, etc.

A Reference tries to be a safe pointer:

int b = 44 ;

int &a = b ;

cout a ; // prints 44

a = 0 ; // b is now 0

BUT: YOU CAN NEVER POINT a AT ANY OTHER VARIABLE.

AND WHEN YOU'RE USING REFERENCES, THE CODE LOOKS JUST LIKE YOU'RE

MANIPULATING ORDINARY VARIABLES. THAT IS STUPID, VERY POOR DESIGN.

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

OBJECTS

SUMMARY

READ CHAPTER Classes: A Deeper Look Part I

THINK ABOUT CLASSES

CLASSES HAVE CONSTRUCTOR FUNCTIONS WHICH RUN WHEN THEY ARE CREATED,

AND DESTRUCTOR FUNCTIONS WHICH RUN WHEN THEY ARE DELETED.

THESE ARE FOR HOUSEKEEPING.

CLASSES HAVE METHODS WHICH ARE EXECUTABLE FUNCTIONS.

CLASSES HAVE INSTANCE VARIABLES WHICH ARE "GLOBAL" IN SCOPE

TO ALL CLASS METHODS.

VARIABLES IN CLASS METHODS ARE SCOPED LOCAL TO THAT METHOD, AND CANNOT

BE SEEN BY ANY OTHER CLASSES, METHODS OR FUNCTIONS.

END SUMMARY

Chapter 9: Classes, A Deeper Look Part I

Classes "Encapsulate" both public and private

"attributes/properties" (data) and public and private "methods"

(functions). That means, they wrap them all up in one

single "object".

/\*-----------------------------------------------------------------

\* C1.cpp - exercise class methods

Thu Oct 26 09:54:46 PDT 2006

-----------------------------------------------------------------\*/

#include <stdio.h>

#include <iostream>

using namespace std;

class prs // personnel record for MI-6

{

private:

bool useme ;

char \*name ;

int num ;

public:

prs \* left , \* right ; // pointers to prs objects

// constructor

prs::prs() { useme = true ; num = 0 ; }

// constructor

prs::prs(bool um, char\* n, int s) {

int temp = strlen(n) ;

temp++ ;

useme = um ;

name = new char[temp] ;

strcpy(name, n) ;

num = s ;

left = right = this ; // make struct point to itself

} // constructor ends

//destructor

prs::~prs() {

if (name) delete[] name ; // release memory

} // destructor ends here

// utility methods

void prs::print(void) {

printf("%p %p %p %-25s %4d\n",

left, this, right, name, num) ;

} // print ends

} ; //class prs ends

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

{

prs \*sp ;

prs p(true, "Jas. Bond", 7) ;

p.print() ;

sp = &p ;

sp = sp->right ;

sp->print() ;

} // main ends

To the "methods" (functions) inside a Class, the "instance variables"

(variables declared outside of any of the class' "methods") are

"global" as far as the methods are concerned.

Points to remember:

Data members of a class cannot be INITIALIZED as part of their declaration

because DECLARATIONS ARE NOT EXECUTABLE STATEMENTS: this must be done

in the constructor or later during the run of the program.

(Declarations are commands to the compiler, not to the operating

system, this means that all variables exist inside the program's

load module, and make the program larger where it's stored on disk.

The scope of class members is local to the class. Even if

they are IMPLIMENTED outside the class declaration and made

part of the class with "::", they are still DEFINED in the

class code, so they are scoped local to the class.

VOCABULARY:

The class { variables...method prototypes } ; is

known as the "interface" to the class. If you don't

actually program the methods INSIDE the class, but

place them outside the class with the "class::" scope

resolution operator, that is called the "implementation"

of the class. It is fashionable now to separate them.

Some of the methods are known as "access functions" and are

used to get at "private:" data items. Methods/functions

which are "private:" to the class are known as "utility

functions", since they are used only by other functions

within the class.

Since each class has a "constructor" function, if only

the do-nothing default one, they also have "destructor"

functions. They are called automatically when the object

is deallocated. This is where you would put any code to

"give back" objects you'd dynamically allocated.

so:

Class A1{ }

has constructor:

A1::A1()

and destructor:

A1::~A1()