Block 7

SUMMARY

READ CHAPTER Pointers

END SUMMARY

Chapter 8: Pointers and Pointer-based Strings ("CStrings").

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/arrays">arrays</A>

Keep referring back to these next 2 lectures throughout the

course. Take your time with each example, but don't expect

to instinctively understand pointers this week, or this year.

Pointers are ALWAYS a source of argument and confusion among

programmers for the first two years or more of their careers.

I do NOT expect you to "master" them now, just to understand the

theory and to be aware that you need a reference work and

extra time for testing when using them.

Arrays again:

int a[10] , b [10][20], c[500][2][5] ;

Multidimensional arrays can take up a lot of space,

and most programmers in business would rather use

many one-dimensional arrays than a single multi-dimensional

one. Also, using several arrays allows for the arrays

to be of different types, as for instance, char for name,

int for empno, double for hours and payrate, etc.

But multi-dimensional arrays are nothing special. Here is

an example of a 6-dimensional array:

"Column 14 on Line 12 on Page 23 in Book 204 on Shelf 5 in

Library 1."

And here is an example of a 10-dimensional array:

"619-555-0324"

See? You use multi-dimensional arrays all the time.

Why are arrays so useful? Well, suppose you had

two hundred sales people in the field, and you wanted

to print out how many hours each worked. You COULD

create two hundred different variables, "herbs\_hours,

frans\_hours, janes\_hours" and soforth, or you could

just store all the hours in an array called "hours".

( int hours[200] ; )

Then, to print out all the hours, you would do

something like:

char s[1024] ; // output buffer

for (int x = 0 ; x < 200 ; x++)

{

sprintf(s, "Salesperson: %4d worked %6d hours\n", x, hours[x]) ;

cout << s ;

}

Result:

Saleperson: 195 worked 3 hours

Saleperson: 196 worked 40 hours

Saleperson: 197 worked 30 hours

Saleperson: 198 worked 18 hours

Saleperson: 199 worked 10 hours

That is better than 200 variables running

around.

You may want to review

<A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/sprintf\_formatting"> sprintf\_formatting</A> again.

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

<A name=day8>

<A HREF="syllabus.html#Menu">Return to Menu</A>

Block 8

SUMMARY

PRACTICE WITH POINTERS. THINK OF WAYS THEY CAN MAKE CODE

SMALLER AND FASTER.

REMEMBER THAT MOST OF JAVA EXISTS TO PROVIDE "C++ WITHOUT POINTERS"

IF YOU UNDERSTAND THE RESPONSIBLE AND SAFE USE OF GUNS, APPLY

THE SAME THINKING TO POINTERS, AND YOU'LL NOT GO WRONG, THEY

FUNCTION IDENTICALLY, AND ARE MISUSED IN THE SAME WAYS.

END SUMMARY

Intro Pointers:

There is more silliness and opinion about pointers than any other

language feature. Java exists largely because people wanted a C++

without pointers. Many people want a world without guns, too, and for

similar reasons. Both are powerful tools. Both can get people in a lot

of trouble fast. Both punish inattention or laziness. Both are the kind

of thing which, if you are used to using it, you tend to rely on it.

And yet, with the proper training, you can get by without either...it

just takes a different approach. But pointers save a great deal of time

and trouble, and settle problems FAST. There is a moral here: efficient

methods are only for those willing to master them. If you're mentally

lazy, don't decide to become a pilot, stay on the ground.

Simply: a pointer is a constant or variable which contains the

address of another variable, or some other place in memory.

A pointer VARIABLE CONTAINS an address. A pointer CONSTANT

IS an address.

int x , \*ip ; // x is an int, ip is a pointer to an int.

ip = &x ; // ip now contains the address of x

\*ip = 25 ; // x now contains 25

pointers can be used for several tasks:

1. to manipulate simple variables whose addresses they

contain.

2. to allow Pass by Reference so that receiving functions in

C may modify calling-function data items.

Read: <A

HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/pass\_by\_ref\_ptr.cpp.txt">pass\_by\_ref\_ptr.cpp</A>

3. to manipulate elements in one-dimensional arrays using

pointer notation instead of array subscripting.

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

4. to process characters in arrays of char (C strings).

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

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

5. to process strings in string vectors such as \*argv[]

and \*\*env.

Read: <A

HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/stringvector\_ptr.cpp.txt">stringvector\_ptr.cpp</A>

6. to rapidly copy or initialize the contents of arrays or

other data structures.

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

7. to design general-purpose utilities such as qsort which

receive pointers-to-functions for use as discriminators.

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/qsort.man.txt">qsort.man.txt</A>

8. to perform lowest-level manipulation of the bits and bytes

that make up data structures.

---

Now, the Silliness: Among programmers it has gotten to be

fashionable to make statements like "an array is a pointer".

Well, no it isn't. But there is a loose interpretation

whereby what they are saying makes sense, and you certainly

can use a pointer as if it was the name of an array, and

"the name of an array is its address", so an array name

is a pointer constant, an address, an address constant,

but not, clearly, a pointer variable.

Just remember this: arrays DO NOT EXIST at the hardware level

of most computer processors. Pointers do, because "indirect

addressing" is built into the CPU's structure..

You'd get tired of hearing this if I was lecturing,

because I repeat it daily:

"THE NAME OF AN OBJECT IS THE ADDRESS OF ITS ORIGIN."

The name of an array, for instance, is a constant address.

Consider:

int a[10] , \*ip ;

To make ip point to the first element, (a[0]) we

could say:

ip = &a[0] ; // ip contains the address of a[0]

but we can accomplish the same thing with:

ip = a ; // (the name of a is its address)

So a MEANS &a[0]

The name is an address.

So, if I wanted to print out the second item in a[], which is a[1]

I could write:

ip = a ; // point to the first int in a[]

ip++ ; // point to the next int

cout << \*ip << endl ; // print it

if I wrote:

int a[] = {9, 8, 7 } ;

int \* ip ;

ip = a ;

ip++ ;

cout << \*ip << endl ;

The code would print:

8

because I would have incrimented where ip points,

so it points to the second array element, THEN

I "dereferenced" ip (fetched the thing it pointed to)

and printed it.

BUT, if I wrote this:

int a[] = {9, 8, 7 } ;

int \* ip ;

ip = a ;

(\*ip)++ ;

cout << \*ip << endl ;

The code would print:

10

Because (parenthesis execute first) I would have

"dereferenced" ip, THEN incrimented the value

I'd fetched (9), producing 10.

See why I made such a fuss over knowing where the

ORDER OF EXECUTION AND ASSOCIATIVITY table is in

the book?

So...what would this print, and why?

int a[] = {9, 8, 7 } ;

int \* ip ;

ip = a ;

++\*ip ;

cout << \*ip << endl ;

If all the pizzas that have been won over

disagreements regarding pointer behavior were all

spread out, we could cover the flight deck of the

USS Harry S. Truman.

What I want you to know about pointers:

"Be careful with that thing, it's LOADED!"

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/pointers1">pointers1</A>

In addition to pointers, C++ instituted a weaker kind of

pointer called a "reference". It must be initialized when

created, and may not be changed, unlike a pointer variable.

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/reference1">reference1</A>

References allow us to do things with functions where we

are able to let the function change data passed to it,

but it looks like an ordinary Pass-by-Value function. An

example is swap().

vector <int> v(100) ; // behaves like int v[100] ;

The main methods of a vector:

vector::assign

vector::at

vector::back

vector::begin

vector::capacity

vector::clear

vector::empty

vector::end

vector::erase

vector::front

vector::get\_allocator

vector::insert

vector::max\_size

vector::operator=

vector::operator[]

vector::pop\_back

vector::push\_back

vector::rbegin

vector::rend

vector::reserve

vector::resize

vector::size

vector::swap

Chapter 6: POINTERS

ARRAYS,

(and for contrast, the "smart" array: VECTORS.

Pointers are the way computers "really" work: indirect

addressing is how arrays are implimented, how instructions

are executed, in short pointers affirm the real internal

structure of the CPU. C simply makes this form of

addressing available to the applications programmer, the

systems programmers, using Assembly Language, always had

this option.

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

\* 00ptr0.cpp - first look at pointer variables

steps to use pointers:

make it

aim it (maybe allocate space THEN aim it)

use it

(maybe deallocate the allocated space)

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

#include <iostream>

#include <cstdio>

#include <stdlib.h>

using namespace std ;

int main ()

{

int i , \*ip ;

ip = &i ; // make ip "point" to i.

\*ip = 777 ; // change contents of where ip points

printf("the address (&i) of i is: %p\n", &i) ;

printf("ip contains: %p and \*ip contains: %d\n", ip, \*ip) ;

printf("i contains: %d\n", i ) ;

// you can dynamically allocate storage...

ip = new int [10] ;

if (!ip) exit(-1) ;

// ...and use it like an array:

for (i = 0 ; i < 10 ; ++i)

ip[i] = i ;

// ...as well as what it is: a pointer variable:

for (i = 0 ; i < 10 ; ++i)

printf("ip[%d] = %d\n", i, \*(ip +i)) ;

delete[] ip ; // give it back to the heap.

} // main ends

Distinguish between AIMING A POINTER:

ip = &1 ; // store into THE POINTER

\*ip = 777 ; // store WHERE THE POINTER POINTS

What do you think of the statement: "Arrays are pointers"?

Array names are pointer constants. The system decides where they

point, using its memory management programs.

Pointer variables may point to blocks of memory the program

treats as an array, and may be used just as array names are:

ip = new int[10] ;

ip[4] = 16 ; // you could also say: \*(ip + 4) = 16 ; same thing.

Pointers have TYPE. The compiler must know what the pointer is

pointing to so it can decide how many bytes it should advance when

you use ++.

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

\* ptrmath01.cpp - demo pointer arithmetic

\* Pointers have TYPE. The compiler must know what the pointer is

\* pointing to so it can decide how many bytes it should advance when

\* you use ++.

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

#include <cstdio>

#include <cctype>

#include <iostream>

#include <iomanip>

#include <cstdlib>

using namespace std ;

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

{

int array[20] ;

int \* ip ;

int count = 0 ;

for (ip = array ; ip <= &array[19] ; ++ip)

\*ip = count++ ;

for (ip = array ; ip <= &array[19] ; ++ip)

cout << \*ip << endl ;

}

WHY USE POINTERS?

Low-level control of the hardware demands it. Also, most

pointer applications take less code.

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

\* strcpyprimive.c - copy one CString to another

TEK Thu Jun 26 14:59:46 PDT 2014

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

#include <stdio.h>

#include <ctype.h>

#include <stdlib.h>

#include <string.h> // strlen

int main ()

{

char a[2048] = {"this is used text: delete me"} ;

char b[2048] = {"This is the new and improved text!!"} ;

int i ;

printf("a: %s\n", a) ;

printf("b: %s\n", b) ;

for (i = 0 ; i < strlen(b) ; ++i)

{

a[i] = b[i] ;

}

printf("a: %s\n", a) ;

printf("b: %s\n", b) ;

} // main ends

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

\* strcpyptr.c - copy one CString to another

TEK Thu Jun 26 14:59:46 PDT 2014

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

#include <stdio.h>

#include <ctype.h>

#include <stdlib.h>

#include <string.h> // strlen

int main ()

{

char a[2048] = {"this is used text: delete me"} ;

char b[2048] = {"This is the new and improved text!!"} ;

char \* s , \* d ;

printf("a: %s\n", a) ;

printf("b: %s\n", b) ;

s = b ; d = a ; ;

while (\*d++ = \*s++) ; // this is worth some study.

printf("a: %s\n", a) ;

printf("b: %s\n", b) ;

} // main ends

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

Block 6:

SUMMARY ARGUMENT PASSING TO FUNCTIONS.

READ CHAPTER 7 (ARRAYS AND VECTORS)

ARRAYS ARE COLLECTIONS OF MANY SAME-TYPE VARIABLES WITH THE SAME

NAME BUT DIFFERENT INDEXES. THEY ARE USED TO STORE LOGICALLY

SIMILAR VALUES SO A PROGRAM DOESN'T NEED MANY VARIABLES WITH

DIFFERENT NAMES.

OF ALL THE SKILLS YOU NEED AS A WORKING PROGRAMMER, THE ABILITY

TO USE ARRAYS IS THE MOST IMPORTANT, SINCE THEY'RE THE MOST

COMMON DATA STRUCTURES.

DON'T GO PAST THE END OF YOUR ARRAY. AT BEST, YOU'LL CAUSE

AN ERROR. AT WORST, YOU WON'T. C++ DOESN'T CHECK.

Error:

int array[10] ;

for (i = 0 ; i <= 10 ; ++i)

array[i] = 0 ; // memory violation when it tries to use array[10]

TO FIX THE PROBLEMS WITH ARRAYS, C++ INTRODUCED VECTORS AND OTHER

OBJECTS WHICH ARE 'SMART', AND LESS PRONE TO BREEDING BUGS

END SUMMARY

Block 6:

An array gives you many variables of the same type with

one name. Suppose you save customer totals by month:

int months[12] = {0} ;

...and the totals for January and February are 31 and 29.

months[0] = 31 ;

months[1] = 29 ;

Simple.

We process arrays with for() loops:

for (int i = 0 ; i < 12 ; i++)

{

cout << "Month: " << setw(2) << i << " " << months[i] << endl ;

}

Arrays are mostly "one-dimensional" in business,

and 2- or 3-dimensional in game design.

In management science or hard science, they can have many dimensions.

If we wanted to store a page of text, 80 columns by 60 lines:

char page[60][80] ;

<A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/strings\_char\_arrays">CHARACTER ARRAYS character arrays</a>

To process multi-dimensional arrays, we use Nested loops:

One of the most powerful program-flow constructions is the use of one

or more loops inside another.

This is an illustration of what I'm talking about:

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

\* uniqidentifier.cpp - nested loops: example

\*

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

#include <cstdio>

#include <cctype>

#include <iostream>

#include <iomanip>

using namespace std ;

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

{

int month, week, day, hour, min, sec ;

for (month = 0 ; month < 12 ; month++)

for (week = 0 ; week < 52 ; week++)

for (day = 0 ; day < 7 ; day++)

for (hour = 0 ; hour < 52 ; hour++)

for (min = 0 ; min < 52 ; min++)

for (sec = 0 ; sec < 52 ; sec++)

{

printf("%02d:%02d:%02d:%02d:%02d:%02d\n",

month, week, day, hour, min, sec) ;

}

}

That's pretty much the simple part of nested loops. The subject

gets complicated when you start swapping slow and fast loops.

Review:

Understand what a function looks like:

return type name (parameter list)

Do you understand that a "function prototype" DECLARES a

function, it explains what the function is and does, so the

program can USE the function. Later on, the function is

DEFINED, in the place where its code appears, that is where

the function actually "is". Prototypes allow us to DEFINE

functions further down in the program than they are CALLED.

DECLARE stops the compiler from nagging at you.

DEFINE actually sets aside memory for a variable or a function,

because it lists the actual source code which is executed

when the function is called.

(Be advised that some textbook authors use Declare

when they should use Define. The absolute authority

in matters like this is: "The C Programming Language,

2nd Edition, ANSI" by Brian Kernighan and Dennis

Ritchie, Prentice Hall Pub.)

Scope: where the function or variable can be "seen".

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/functions\_and\_parameters">functions\_and\_parameters</A>

Understand:

Pass by value: function gets a copy of the argument.

Pass by reference: function gets a "reference" to

the REAL argument, and can CHANGE IT.

OVERLOADED functions: multiple functions with the

same name, but different argument lists.

I have installed the "mcrypt" command in /usr/local/bin

for your use if you have any digital secrets you want to keep.

Simply put: there is no way you can have

privacy on a computer if someone is

determined to spy on you. They can look

over your shoulder, they can learn the

root password and look inside your files,

they can use "TEMPEST" detection and echo

what's on your monitor screen from a truck

parked outside your house, if they have

the equipment.

Or they can tap your phone, or cable, or cell signal.

The System Administrator ("root" on Unix, "Administrator"

on Windows) can look in any file you save to disk,

regardless of privacy settings.

BUT: if you "crypt" the file, when someone manages

to look at the file contents, they will only see

gibberish. The only way to re-constitute the file

contents once you "crypt" them is to know the password

you used in the first place.

P.S., this is true, not a product warranty.

If you lose the password, it cannot be "recovered".

You could write a program to try all the words in

the dictionary, but....

Try "man crypt" for a tutorial on how to use crypt.

Examples:

encrypt a file hold to hold.nc

crypt hold

encrypt an input stream to an ouput stream

crypt -F < infile > outfile

decrypt an input stream to an output stream

crypt -F -d < infile > outfile

------------------

Read: Chapter Arrays and Vectors. /\*\* TODO: chapter headeing and number

Now we finally get to arrays. Arrays are more important in business than

any other data structure, and only very rarely are they more than one-

or two-dimensional. Rather than multi-dimensional arrays, most business

programmers use several one-dimensional arrays, one each for name, empno,

city, ssan, etc.

THINK OF A SPREADSHEET AS SEVERAL ONE-DIMENSIONAL ARRAYS:

EACH ARRAY IS A COLUMN.

Ignore material on Vectors (templates) for now. We will concentrate on

the nature of Arrays and using them with loops. Most programming is

maintenance, so programmers work with 20-year old technology most of

their careers. When you get to create some new software, you can use

the new language features...if your client has installed them. Mostly,

they don't.

Defining an array:

type name [numeric constant] ; // "definition": sets aside memory.

Example:

int days[365] ;

Using an array element:

days[304] = 12 ; // store a 12 in Hallowe'en's position.

Passing an array as an argument:

function prototype: int f2( int[23]) ;

or: int f2( int[]) ;

function definition: int f2( int a[23]) ;

or: int f2( int a[]) ; // function doesn't have to be told how many elements in a one-dimensional array.

Funtion Invocation passing array argument:

int a[ 300 ] ;

cout << f2(a) ; // no square brackets or sizes, just name of array.

GOOD PROGRAMMING PRACTICE:

Initialize your variables as the first thing your program does.

int x ; // this leaves any random number in x.

int x = 0 ; // this sets x to zero before it is used.

C language compilers often do NOT INITIALIZE variables to zero, (or

anything else). (Microsoft excepted.)

Why?

C is designed for SPEED. Your program will only do what you tell

it to. Since Windows is an engineering joke, they don't worry about

speed, and all data is initialized in Visual Studio.

So?

So C assumes you know what you're doing, so you'd better.

After you define an array, you must initialize its elements, or

that memory will contain whatever junk was left over from the

last time that memory was used. Use a for() loop to process a whole

array with a minimum of code.

Old way always works:

int a[500] ;

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

{

a[i] = 0 ; // initialize to zero

}

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

{

cout << a[i] << endl ; // print whole array

}

or even:

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

{

cout << a[i] << '\t' ; // print whole array

if (0 == i % 8) // print out a nice square table,

cout << endl ; // instead of a single column of numbers.

}

New way is cleaner:

int a[500] = {0} ;

SORTING (Ordering) ARRAYS:

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/bubble.c.txt">bubble.c</A>

Character Arrays:

char c[1024] ; // set up an input buffer

cout << "Enter your name\n" ;

cin.getline(c, sizeof(c), '\n') ;

// parms are: 1. where to store input,

// 2. max number of characters to store,

// 3. the input character which terminates input.

The above will allow Firstname, Middlename, and Lastname to all be read

into array c[] in one operation.

Char array initialization:

char c[] = "Hello, World!" ; // compiler counts and allocates enough memory.

Be aware that Arrays are even now today the most important data structures

in programming.

<A HREF="http://209.129.16.61/~hhaller/data/cisc192/asst4">asst4.</A>

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/command\_line\_arguments">command\_line\_arguments</A>

Text: (Chapter 13)

C GURU MEDITATION:

To a real guru, it is a Great Truth that there are no "multidimensional

arrays". Rather, there are "arrays of arrays" or rather "arrays of pointers".

So, the definition:

int a[3][10] ;

sets aside 3 arrays of 10 integers, and one array of three pointers

to those arrays, since the addresses of the three arrays are "pointer

constants").

The NAMES of the three arrays are:

a[0]

a[1] and

a[2].

But having said this, you can promptly forget it, just so long as you

remember that you must use:

a[x][y] ;

notation, not some ERROR like:

a[23,54] ;

You can ignore the case studies, template (vector) discussion in

the chapter. It should take rather a short time to get through it

if you read these notes first.

Now think about how you would USE arrays, say, if you had a

lot of data, the number of revenue each of your sales persons

brought in each week, and the number of salespersons changed

each week. How would you write your programs so you didn't

have to re-write them?

cin >> number\_salespersons ;

double a[number\_salespersons] ;

for ( i = 0 ; i < number\_salespersons ; i++)

a[i] = 0 ;

See? VARIABLES are the secret to writing versatle, flexible

and useful programs.

TWO DIMENSIONAL ARRAYS:

If you understand the program code above, you'll be able to manipulate

two dimensional arrays without too much aspirin.

--------------------

Class Demonstration: sortdemo in

~hhaller/data/cisc192/today/sortdemo Demonstration shows:

multi-file project, the "make" utility, the "Makefile" file

which supervises "make",

Many projects are built from dozens or hundreds of \*.c,

\*.cpp, and \*.h files. All that source code takes a LONG

time to compile. We use a "make" utility to only compile

what needs to be compiled. If we have \*.o (object files,

compiled code) from source files that have not changed since

the last compile, they are NOT re-compiled, that would be

wasted effort. Only source files that are younger than their

corresponding \*.o files are compiled, then all the \*.o files

are linked to build the executable program.

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/make\_and\_Makefiles">make\_and\_Makefiles</A>

New: If you're interested in seeing "WHY" makefiles are used, or you

just want to see a realistic-sized program or two, take a look at:

<A HREF="http://209.129.16.61/~hhaller/data/cisc192/bbs">bbs</A>

or:

<A HREF="http://209.129.16.61/~hhaller/data/cisc192/sortdemo">sortdemo</A>

or

<A HREF="http://209.129.16.61/~hhaller/source/cpp/games/rogue/dist>the source for the "Rogue" game</A>

These are programs which required modularization in order not to become

unwieldy and difficult to maintain.

If you wish to "try out" the tbC BBS, you may log into Buffy as user

"bbs", passwd "bbs". See what you think.

====

Now, here is a puzzle to puzzle over: since the more

arguments you pass to a function, the more "overhead" (waste

stack processing) is involved, how would you pass an array

to a function if:

1. Each time the program runs, the array will have a different size,

so you cannot hard-code the size of the array into the function

2. You want to ONLY pass the array, not the array PLUS another

variable containing the size of the array.

3. You have complete control over all the program code, you can

allocate, fill, and process the array any way you choose.

<A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/array.puzzle">Think about it before you look at the solution!</A>

Well, did you solve it without cribbing? If so, you're an alligator step

closer to 'leetness. W00t!

Vectors:

#include <iostream>

#include <vector>

using namespace std ;

int main()

{

vector <int> nums1(10) ;

nums1.clear() ;

nums1.capacity () ;

nums1.begin() ;

nums1.end() ;

nums1.date() ;

nums1.empty() ;

nums1.erase(1,4) ;

nums1.insert(3,x) ;

nums1.pop\_back() ;

nums1.push\_back() ;

} // main ends

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

\* vectiter.cpp - show vectors and iterators

Mon Oct 1 15:27:57 PDT 2012

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

#include <cstdio>

#include <cctype>

#include <iostream>

#include <iomanip>

#include <vector>

using namespace std ;

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

{

vector <int> a(0) ;

int x ;

for (x =0 ; x < 10 ; x++ )

a.push\_back( x) ;

cout << "a.size: " << a.size() << endl ;

// special iterator for vectors:

vector<int>::iterator i ;

for (i = a.begin() ; i != a.end() ; i++ )

cout << \*i << endl ;

cout << "===========================\n" ; ;

vector<int>::reverse\_iterator r ;

for (r = a.rbegin() ; r != a.rend() ; r++ )

cout << \*r << endl ;

}

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

\* vector1.cpp - show the vector template class

\* Deitel & Deitel p.380

compile with g++ vector1.cpp -std=c++0x

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

#include <cstdio>

#include <cctype>

#include <iostream>

#include <iomanip>

#include <vector>

using namespace std ;

void outputVector( const vector< int > & ) ; // unchanging reference to a vector

void inputVector( vector< int > & ) ; // changable reference to a vector

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

{

vector< int > integers1 ( 10 ) ;

vector< int > integers2 ( 10 ) ;

// each of these makes the vector grow in memory:

integers2.push\_back(2) ;

integers2.push\_back(52) ;

integers2.push\_back(902) ;

integers2.push\_back(1209) ;

integers2.insert(integers2.begin() ,1209) ; // look for the duplicate at the beginning

// print size and contents

cout << "Size of vector integers1 is: " << integers1.size() << "\n"

<< "vector integers1 after initialization: \n" ;

outputVector( integers1 ) ;

cout.put('\n') ; cout.put('\n') ;

cout << "vector integers2: \n" ;

outputVector( integers2 ) ;

cout.put('\n') ;

} // main

void outputVector( const vector< int > &array)

{

//size\_t i ; // control var

//for (i = 0 ; i < array.size() ; i++)

int count = 0 ;

for (int i: array) // range-based for() loop

{

cout << setw( 12 ) << array[i] ;

if( (++count ) % 4 == 0 ) // 4 nums / row

cout << endl ;

} // for

} // outputVector