Block 3:

SUMMARY

READ CHAPTER 3 Directing your C++ Program Flow

YOU CAN INPUT WHOLE WORDS, WHOLE LINES, OR JUST SINGLE CHARACTERS. MOST

TIMES CHARACTERS ARE THE MOST FLEXIBLE WAY FOR A PROGRAM TO READ INPUT.

"CONTROL" MEANS, LETTING THE PROGRAM DECIDE WHAT TO DO BASED ON INPUT.

CONTINUE RUNNING OR STOP

TAKE ONE OF SEVERAL PATHS THROUGH ITS CODE

THERE ARE TWO BASIC KINDS OF CONTROL: DECISIONS AND LOOPS

SOME LOOPS ARE PREDETERMINED: for()) SOME ARE OPEN (while(), do .. while() ;

DECISIONS ARE if() and ? :

DECISIONS USE RELATIONAL OPERATORS LIKE != == > <= AND SO ON.

BEGIN ASSIGNMENT 1

END SUMMARY

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

Control I

BLOCKS are statements surrounded by braces {}.

You can control a single statement OR a BLOCK using

a conditional or repetition feature of C++.

INDENTATION isn't necessary to the compiler, only to the programmer.

But to the programmer it is VERY necessary.

Conditional:

IF

(if, looping, conditionals.)

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

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

An if() only controls one statement. The controlled statement

can either be a single physical statement, or a compound

statement made of several physical statements enclosed in

braces. (A BLOCK).

switch..case

The switch construct is just a clearer way of writing a series of if

statements, it is for you the programmer, it doesn't run any faster or

slower. BUT:

switch (x)

{

case -1 : cout << "-1" << endl ;

case 0 : cout << "0" << endl ;

case 1 : cout << "1" << endl ;

} ;

Bear in mind that above, if x equals -1, this will print:

-1

0

1

If x == 0, the code will print:

0

1

Why?? THE break MESSUP IN switch() STATEMENTS.

Because you must use "break" statements to fix this characteristic of

switches called "cascading":

switch (x)

{

case -1 : cout << "-1" << endl ;

break ;

case 0 : cout << "0" << endl ;

break ;

case 1 : cout << "1" << endl ;

break ;

} ;

LOOPS (Repetitive execution.)

while(some condition)

statement ;

Make sure that something happens in "statement" that will

eventually alter the "condition" or you'll have an

infinite loop.

do

{

something ;

} while (some condition) ;

The do..while LOOKS like the while, but it is different,

since the do..while WILL ALWAYS EXECUTE AT LEAST ONE TIME.

The "while" construction may NEVER execute, if the condition

starts out false.

do..while is used to get a reply from a user, and to keep asking until

the reply makes sense.

The most powerful loop is the for() loop. The for() loop is used when

we know how many times we want to repeat an operation or task, when

you have multiple loop-control counters, and generally complex

situations.

This for() loop is the same in C/C++/Java/PHP/awk/PERL.

NOTE: good programs almost never have "complex situations", because

good programmers keep thinking until they've broken the task down into

simple steps.

for (starting\_initialization ; continuity\_test ; iteration\_operation)

executed-statement

// print numbers 0 - 9:

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

cout << x << endl ;

// print numbers running up from 0 and down from 10:

for (x = 0 , y = 10 ; x < 10 && y > 0 ; x++ , --y)

cout << x << " " << y << endl ;

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

Loops can be prematurely ended,

continue ; ends THIS ITERATION through the loop

break ; ends the loop altogether

goto is used to jump from one part of a function to another.

goto is never necessary, and is only used when it provides

a CLEANER AND CLEARER way to control a program, or the code

needs to run as fast as possible.

Many programmers consider goto a dirty word.

Up to now, we've read in words or lines at a time from the

user's keyboard using "cin >>". "cin >>" is an "overloaded

stream operator" and has a lot of intelligence, so that

it can tell how many characters / keystrokes to read based

on what variable type you're filling.

But how about if we want to read in character-by-character?

Then we use:

int cin.get()

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

\* single-char input and output demo

\*

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

#include &lt stdio.h&gt

#include &lt iostream&gt

using namespace std;

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

{

int x ;

x = cin.get() ; // cin.get() returns a CHAR

cout.put(x) ; // cout.put() accepts a CHAR argument

cout.put('\n') ;

cout << x << endl ; // cout << accepts an object, and decides for itself

// how to format it.

} // main ends

THEREFORE, after: $ a.out

this program prints:

G

G

71

I press G, 'G' is printed by put(), and the integer VALUE

of the CHARACTER 'G' (71) by cout << .

Remember, we generally read in character data into INTEGERS because

the EOF (end of file) mark is really the integer constant -1, and

can't fit into a char, only an int, and we \_do\_ want to know when

we're at the end of the input.

(If you want to confirm this, run sizes.cpp. cout << sizeof(int) ;

produces 2, sizeof(char) produces 1. The digits are the number of bytes.)

Now we wish to write a filter program (think of water pumps)

which reads in a stream of characters, and capitalizes

all letters, then prints them out in their capitalized form.

To capitalize, we will use toupper(). The toupper() function takes an

integer value, and if it represents a lower case letter, returns the

integer value which corresponds to that capital letter. If the

argument you send toupper() IS NOT a lower-case letter, then toupper()

just returns exactly what you send it. This is an example of a "safe

and robust" function that you don't have to waste an if() statement

on.

PSEUDOCODE:

START PROGRAM

DEFINE c //a variable to hold the characters we read in.

BEGIN LOOP

READ A CHARACTER INTO c

PRINT OUT toupper(c)

CONTINUE LOOP WHILE THERE IS INPUT

END PROGRAM

Now we have a blueprint, it's just a matter of

choosing our tools.

We will need an int variable, we've already decided to call it "c".

We will need a loop structure. Since we have no idea how much

or how little our user will type, we can't use for(), so we

probably want to use "while" or "do..while".

We still need a tool to tell when the user is still typing,

in other words, an end-of-input indicator. We have one,

it is called EOF. It is defined in the cstdio header file.

(THIS is the reason we are reading our keystrokes into an

int, instead of a char. A char cannot hold EOF, so our

end-test would not work.)

int c = ' ' ; // start knowing what's in c

do

{

c = cin.get() ;

cout.put(c) ;

} while (c != EOF) ;

The above loop will continue to print what it reads until it hits end

of file. If the user is typing to the program, s/he can type end of

file with ^D (hold down CTRL and press d) in Unix, or ^Z (control z)

in Windows.

This is the traditional C way to find EOF.

Another C method is:

while (!feof(stdin))

{

c = cin.get() ;

cout.put(c) ;

}

A more C++ - style version of detecting EOF is this:

c = cin.get() ;

while ( ! cin.eof())

{

cout.put(c) ;

c = cin.get() ;

} // while ends

(Here we are using the METHOD eof() which is a member of the CLASS OBJECT

cin, which is an INSTANCE of the class "ofstream" .)

This program is intended to "filter" files, so we run it so:

asst3.exe < infile.txt

That way, when the program reaches the end of the file infile.txt,

the SYSTEM generates an EOF for us. The "<" means: "Make infile.txt

impersonate a user typing at the console."

03\_lecture.notes

Understand the Chapter on:

if() { } else { }

while() { }

do { } while () ;

for ( ; ; ) { }

if you wanted to print a series of ascending numbers:

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

\* ascending\_numbers.cpp - simple for()

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

#include <iostream>

#include <iomanip>

using namespace std ;

int main ()

{

int a ;

for (a = 0 ; a < 11 ; ++a)

cout << a << endl ;

}

This prints:

0

1

2

3

4

5

6

7

8

9

10

Now, suppose you wanted to print a table of rows, and each row

a series of ascending numbers:

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

\* ascending\_rows.cpp - nested for() 's

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

#include <iostream>

#include <iomanip>

using namespace std ;

int main ()

{

int a, b ;

for (a = 0 ; a < 10 ; ++a) // prints rows

{

for (b = 0 ; b < 10 ; ++b) // prints individual row

cout << b << '\t' ;

cout.put('\n') ;

}

}

this prints:

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

Now, what would this print?

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

{

for (b = a ; b < 10 ; ++b)

cout << b << '\t' ;

cout.put('\n') ;

}

Here's the answer. Can you explain why?

0 1 2 3 4 5 6 7 8 9

1 2 3 4 5 6 7 8 9

2 3 4 5 6 7 8 9

3 4 5 6 7 8 9

4 5 6 7 8 9

5 6 7 8 9

6 7 8 9

7 8 9

8 9

9

You can see how powerful for() loops are.

How would you modify the above program to produce this?

0 1 2 3 4 5 6 7 8 9

1 2 3 4 5 6 7 8 9

2 3 4 5 6 7 8 9

3 4 5 6 7 8 9

4 5 6 7 8 9

5 6 7 8 9

6 7 8 9

7 8 9

8 9

9

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

For Samurai, here's a Black Belt Meditation: how would

you write a program "full\_ascending\_rows.cpp" to produce this:

0 1 2 3 4 5 6 7 8 9

1 2 3 4 5 6 7 8 9 0

2 3 4 5 6 7 8 9 0 1

3 4 5 6 7 8 9 0 1 2

4 5 6 7 8 9 0 1 2 3

5 6 7 8 9 0 1 2 3 4

6 7 8 9 0 1 2 3 4 5

7 8 9 0 1 2 3 4 5 6

8 9 0 1 2 3 4 5 6 7

9 0 1 2 3 4 5 6 7 8

(hint, it only takes two extra lines.)

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

Now consider: we spies need to breach security and gain access to

enemy computers. The enemy (THRUSH) has a computer which requires

a two-letter password. Write a program to generate all possible

two-character passwords and print them on individual lines. This is

called a Brute-Force Attack.

(Remember, you can "break" a program when it's running by using

the DEBUG menu, or pressing Control-C at the command line.)

Now, try something a little more realistic: imagine that the THRUSH

computer requires a 6-character password. Modify the code, Get it

running, and let it run for a while. Any feelings about run time?

What do you think of Brute Force attacks now?

The way to calculate how many different password possibilities

is: 26 to the 6th power. It's OK to use a calculator or to

write code to do it.

A generalized program would tell you how many combinations there are

for n letter passwords. Here's what such a program might print:

Length Possible Passwords

2 676

3 17576

4 456976

5 11881376

6 308915776

7 8031810176

8 208827064576

9 5429503678976

10 141167095653376

11 3670344486987776

12 95428956661682176

13 2481152873203736576

14 64509974703297150976

15 1677259342285726023680

16 43608742899428878188544

17 1133827315385150749016064

18 29479510200013920011288576

19 766467265200361851574026240

20 19928148895209410339947937792

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

\* flowcontrol.cpp - show how to modify the execution path.

\* various flavors of control except switch()

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

#include <iostream>

#include <iomanip>

#include <cstdio>

using namespace std ;

int main ()

{

int i, j, one = 1 , twenty = 20 ;

bool match = false ;

printf( "One %s twenty\n", (one == twenty) ? "matches" : "does not match") ;

for (i = 0 ; i < 3 ; ++i)

cout << setw(3) << i << endl ;

do

{

cout << i-- << endl ; // note the '--' post incrimentation

} while (i) ;

match = one == twenty ; // arithmetic, relational, assignment order

cout << "boolean match: " << match << endl ;

cout << "boolean match: " << std::boolalpha << match << noboolalpha << endl ;

i = 999 ;

cout << "Enter a number, stop with 0:" ;

while(i != 0)

{

cin >> i ;

cout << i << endl ;

}

cout << "all done\n" ;

} // main ends

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

\* perfect.cpp - produce all "proper factors" of a number,

Demos like this are crucial for you to understand what computers

are GOOD FOR. Most trained humans are better at solving word

problems than computers are: math requires thought, strategy,

judgement.

Finding Perfect Numbers is something else again.

It's easy to explain Perfect Numbers with an example:

6 is perfect, because 1 + 2 + 3 == 6 == 1 \* 2 \* 3

Its formal factors sum to the number.

10 is Deficient because 1 + 2 + 5 == 8

12 is Abundant because 1 + 2 + 3 + 4 + 6 == 16

Did you know about these? They come from the field of "Number

Theory", which is one of the most-appreciated fields in math,

and the one most non-mathematicians enjoy reading about most.

So this program is designed to show you how computers can be used

to search for difficult answers like this. Other examples are:

digits of Pi, primes, Triangular Numbers, Very Large Primes, and

so on.

You start the program one of two ways:

a.out | less

a.out -p | less

The first way shows you everything. The second only shows you

the Prfects.

I want you to play with this for the goosebumps. Starting it the

first way, you'll see thousands of numbers race up the screen, and

it'll go on until you kill the run with ^C.

The second way, it'll print four numbers, then sit thinking about it.

But YOU'll know it's testing and rejecting thousands of numbers.

And that's one of the reasons computers are handy.

If you study this program, there are subtle techniques used in

it to avoid double-computing. They center on sprintf().

Sun Mar 29 18:53:26 PST 2009

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

#include <cstdio>

#include <cctype>

#include <cstring>

#include <iostream>

#include <iomanip>

#include <math.h>

#include <limits.h>

using namespace std ;

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

{

bool PERFECT\_ONLY ;

unsigned long charcount, sum ;

unsigned long i, divisor ;

char buf[4096] , \*op ;

if ( argc > 1 && 0 == strncmp(argv[1], "-p", 2))

PERFECT\_ONLY = true ;

else

PERFECT\_ONLY = false ;

for (i = 1 ; i < ULONG\_MAX ; i++)

{

op = buf ;

sum = 0 ;

charcount = sprintf(op, "%10lu: ", i) ;

op += charcount ;

//for (divisor = 1 ; divisor <= 1 + (int) sqrt((double)i) ; divisor++)

for (divisor = 1 ; divisor < i ; divisor++)

{

if (0 == ( i % divisor) )

{

sum += divisor ;

//charcount = sprintf(op, "%lu ", divisor) ;

op += sprintf(op, "%lu ", divisor) ;

//op += charcount ;

}

}

if (sum == i)

{

charcount = sprintf(op, "perfect: %lu", sum) ;

op += charcount ;

}

sprintf(op, "\n") ;

if (sum == i || !PERFECT\_ONLY)

printf("%s", buf) ;

}

}

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

\* args.cpp0 - show arg processing

\*

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

#include <cstdio>

#include <cctype>

#include <iostream>

#include <iomanip>

#include <cstdlib>

using namespace std ;

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

{

cout << "argc: " << argc << endl ;

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

{

printf("argv[%d]: %s\n", i, argv[i]) ;

} // for

} // main ends

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

\* brute\_force\_8.cpp - breach password security

\*

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

#include <cstdio>

#include <cctype>

#include <iostream>

#include <iomanip>

#include <cstdlib>

using namespace std ;

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

{

int a, b ,c, d, e, f, g, h ;

for (a = 'a' ; a <= 'z' ; ++a)

for (b = 'a' ; b <= 'z' ; ++b)

for (c = 'a' ; c <= 'z' ; ++c)

for (d = 'a' ; d <= 'z' ; ++d)

for (e = 'a' ; e <= 'z' ; ++e)

for (f = 'a' ; f <= 'z' ; ++f)

printf("%c%c%c%c%c%c\n", a,b,c,d,e,f) ;

}

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

\* ascending\_numbers.cpp - simple for()

\*

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

#include <iostream>

#include <iomanip>

using namespace std ;

int main ()

{

int a, b ;

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

{

// start at outer loop value: a, and complete digits to 9

for (b = a ; b < 10 ; ++b)

cout << b << '\t' ;

// start where prior loop stopped and print digits to

// where it began: a.

for (b = 0 ; b < a ; ++b) // complete numbers to o.l.v.

cout << b << '\t' ;

cout.put('\n') ;

}

}

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

\* reverse\_ascending\_numbers.cpp - simple for()

\*

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

#include <iostream>

#include <cstdio>

#include <iomanip>

using namespace std ;

int main ()

{

int a, b ;

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

{

for (b = 0 ; b < a ; ++b)

cout << " \t" ; // print leading whitespace

for (b = a ; b < 10 ; ++b)

cout << b << '\t' ; // print numbers in row

cout.put('\n') ;

}

}