Block 00: Tuesday of First Week

IF YOU HAVEN'T TAKEN CISC 181 AND 186, YOU SHOULD READ THIS FILE

CAREFULLY.

These files "block??" supplement the text. It helps if you can see something

from different sides, in different lights.

Most students have used a personal computer. You sit at the terminal

and control the processing. The local computer's Central Processing

Unix does the work. Sometimes you get or put something over the internet,

but your system does the work.

That is a more recent development, and applies to the less-powerful

systems. Traditionally and still, large servers ("Mainframes")

reside in buildings made to house them, and people use the big

systems' power over a network. Airline reservation systems, banks,

hospitals, governments, all big entities use large servers. They

can store more data, and process it faster.

New employees who understand mainframe processing will have a better

chance at promotion. Those who only know Windows or Macintoshes will

spend more time in a cubicle doing scut work.

In this class you have the option of using either or both models.

The programs can be written and tested on your PC using CodeBlocks or

Microsoft Visual C++, or any other C++ compiler which meets ISO standards.

But you can also use a terminal program to access your account on Buffy

(209.129.16.61) a Linux server which behaves like any other Unix server.

I encourage you to try both. This is a place of education, you should

take advantage of it to learn everything you might benefit from knowing.

If you went to work for an airline as, say, a manager, and you were

competing with another manager who'd held a pilot's license for years,

who would YOU expect them to hire? Even if the position didn't involve

flying an airplane, someone who knew FAA rules, airport procedures,

radio and navigation would be a better fit, wouldn't they?

So. Any company of any size will have large Unix servers running their

database systems. If you know that world, you're twice as likely

to get hired, so now is a good time to start.

YOU === PC

vs.

MAINFRAME

||

YOU =====(INTERNET ) ===========||

BOB =====(INTERNET ) ===========||

CAROL =====(INTERNET ) ===========||

TED =====(INTERNET ) ===========||

ALICE =====(INTERNET ) ===========||

HOBORG =====(INTERNET ) ===========||

KARLA =====(INTERNET ) ===========||

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

PROCEDURAL NOTE: I'VE INCLUDED THE EXAMPLE PROGRAMS FROM THE FIRST COUPLE

OF CHAPTERS IN THE LECTURE FILES IN block?? BECAUSE THE REALITY OF THE

SITUATION IS THAT ALL STUDENTS DON'T HAVE THE TEXT WHEN THE SEMESTER

STARTS. Many didn't know for sure they were going to be in the class.

So this is a stopgap. By Chapter 3, everyone should have the text.

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

Computers execute commands.

Stored commands are called programs.

PROGRAMS CAN ONLY:

//INPUT

//STORE

//TEST

//CALCULATE

//JUMP

//OUTPUT

That's all. The whole profession of programming is concerned

with those six.

Don't get lazy, though, our profession may only have 6 rules,

but the entire Universe only has 3. It can still get really

complex really fast.

This course is process of putting some parts together in different ways.

The job of the programmer is to solve problems.

Every computer program can be broken down into these six words.

Every computer program design should start as a list of some or

all of these six ideas in various arrangements, until you can

write the skeleton logic of any program without thought.

"You must study this" -- Miyamoto Musashi.

If you wanted to write a program to produce the 'times table' of a given

number, here's how your mind might approach it:

I.: Write clear English expressing what you need to do. Keep it short:

"Input a number, multiply it by the numbers from 0 to 12, printing out

the results."

II.: Translate that short description into examples of the SIX WORDS:

//INPUT NUMBER

//STORE NUMBER AS X

//STORE 0 AS M

//CALCULATE PRODUCT = X TIMES M

//STORE PRODUCT AS P

//OUTPUT X, " \* " , M , " = " , P

//ADD 1 TO M

//TEST IF M => 13

// IF NOT JUMP BACK 5 LINES

// OTHERWISE QUIT

Refine it. Refine it until each line is almost a valid C++

statement. (This means you must know C++.)

get number //INPUT AND STORE

multiplier = 0 //STORE

product = number \* multiplier //CALCULATE

print "number \* multiplier = product" //OUTPUT

number = number + 1 //CALCULATE AND STORE

if (number < 13) //TEST

goto top of loop //JUMP

III.: Flesh out Step II into a valid C++ program:

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

\* multiply.cpp - a first example of programming

\* T.E. Harrisburg, Army Security Agency

Sat May 31 21:28:12 PDT 2014

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

#include <iostream>

using namespace std; //get ready to INPUT

int main()

{

int number, multiplier, p ; //reserve places to STORE

cout << "Enter a whole number: " ;

cin >> number ; //INPUT and STORE number

multiplier = 0 ; //STORE 0 AS M

looptop:

p = number \* multiplier ; //CALCULATE number \* multiplier STORE as product

cout << number << " \* " //OUTPUT

<< multiplier << " = "

<< p << "\n" ;

multiplier = multiplier + 1 ; //ADD 1 TO multiplier

if (multiplier < 13) //TEST IF multiplier < 13

goto looptop ; //IF NOT JUMP BACK 5 LINES

} //OTHERWISE QUIT

IV.: Compile, run, test the program. Make sure that both program syntax and

design logic are correct.

This is a Computer Program in C++.

All Programs are combinations of input, store, calculate, output, test, and jump.

Read the example over as often as you need to. If you understand this

example, you're ready for this course. Until you understand this example,

you're going nowhere.

If you're curious, you want to try copying that program into Code Blocks

and running it. Nobody's stopping you.

ASSIGNMENT:

If you're in the Online class, log into Blackboard now, and leave a

message in the General Discussion area. You may want to talk about

yourself, why you're taking 192, what your programming experience is,

any patents or copyrights you hold, what systems you've worked on, what

foreign languages you speak, what parts of the world you'd like to visit,

maybe work in. Anything relevant to the profession. Hobbies and similar

interests are often the source of new directions in life, too.

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

Daily routine for CISC 192:

EVERY DAY:

0. Check Syllabus for this week, confirm Block and chapter in text:

1. Go to the current Block (same as the number of

the current week, or the current week times 2.) You should read

each Block as many times as you need to. It points out what's

important in the current chapter of the Textbook.

2. Open the Text to the current chapter. Flip through it, just looking

at the C++ code. Really read the programs, try to see what they're doing.

Enter and run them. Then go back to the start of the Chapter.

3. View any Powerpoint presentations in block??

4. Read the current chapter of the Textbook while sitting

in front of a computer, working the examples, making notes.

5. Work the current assignment.

A more complete step-by-step description of how to learn programming

is in the file in this directory: how.to.learn.programming.faster.txt.

When you have time, come back and look at it.

For The Nerves:

The Mummer's Dance:

https//www.youtube.com/watch?v=yJbkOIXkmX4

Loreena McKennitt is an example to us all: a great talent who could just

coast on that talent, yet also a true scholar whose performance is also

education. She sings of the old times and faith of the Celts. Spells,

magic, miracles. Each culture and generation has its reality, and

its wonders. These machines are ours. Master computers and you are in

control of the central mystery of your world.

How?

You already know, just apply it:

We open our emotions to music, because it is a tool that can directly

change our feelings: it is a form of magic. Our relationship to

complex machines can be magical. The essence of magic is surprise,

and computers are in the surprise business. The most important thing

in understanding the world is to be open to it. Empty your teacup

and observe. Some things may jar you, that's because you are learning

the great truths, among which is that little things are true or false,

great things are true and false. Mastering that gives us reasoning

and judgement. Instead of butting heads like angry primates, we walk

around the conflicts and build the next world.

In the end, there is only one indicator of who is right: they build

something which works. That means they know part of the truth.

But it is always only part. There is always more.

All engineering is art, all art is emotion. No passion, no result.

If you don't want to see the thing work, why waste the time?

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

\* cap.cpp - Create a filter program which capitalizes all

\* letters in an input stream.

\* T. E. Harrisburg

\* Wed Aug 21 05:56:44 PDT 2013

Revision 0

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

#include <cstdio>

#include <iostream>

using namespace std ;

int main ()

{

int c ;

c = cin.get() ; // Read first character

while (!cin.eof()) // Within a loop structure (while())

{

if (isalpha(c)) // if character is alphabetic

{

cout.put(toupper(c)) ; // print out its uppercase equivalent

}

else// otherwise

{

cout.put(c) ; // print out the unchanged character

}

c = cin.get() ; // read a new character

} // if not at End of File, repeat loop

cout.put('\n') ; // else print a line feed and end

} // end program

(From www.cprogramming.com)

5 Ways You can Learn Programming Faster

By Alex Allain

Learning to program isn't something you can do in an afternoon, but it

doesn't have to be a life's work, either. There are lots of things you can

do to make it easier on yourself when you are learning to program. You

already know about The 5 Most Common Problems New Programmers Face--And

How You Can Solve Them. Now, discover how to get the most out of your

learning.

One common theme across many of these tips is:

don't go too fast; get it right before moving on.

When I was teaching C, there were always a few students who came into

the class knowing a bit about programming. Inevitably, some of these

students did great in the first few weeks only to fall further and

further behind as the course went on. Why? They went too fast through

the introductory part of the course, thinking they knew it all--but they

rarely did. They knew some of the material, but not enough to have a

strong grasp of the fundamentals.

At the same time, you must not stop making progress--you can go too slow

as well as too fast. Don't avoid a topic after you've mastered everything

leading up to it. By facing more challenging ideas, you'll help cement

your grasp of the basics.

1. Look at the Example Code

Reading is usually about the words on the page, but learning to program

is about code. When you're first learning to program, you should make

sure to look at, and try to understand, every example. When I first

learned to program, I would sometimes read the code examples before

the text, and try to figure out what they did. It doesn't always work,

but it did force me to look at the example very carefully, and it often

helped make the writeups clearer.

If you want to see what sample code looks like, you can read this site's

introductory programming tutorial. This tutorial spends a great deal of

time talking about the sample code to help you work through exactly what

the code does.

2. Don't Just Read Example Code--Run It

But when you're reading a programming tutorial (or book), it's easy to

look at the sample code and say "I get it, I get it, that makes sense". Of

course, you might get it, but you might not get it, and you just don't

know it. There's only one way to find out--do something with that code.

If you haven't already, get a compiler like Code::Blocks set up.

Then type the sample code into a compiler--if you type it, instead of

copying and pasting it, you will really force yourself to go through

everything that is there. Typing the code will force you to pay attention

to the details of the syntax of the language--things like those funny

semicolons that seem to go after every line.

Then compile it and run it. Make sure it does what you think it does.

Then change it. Software is the most easily changed machinery on the

planet. You can experiment easily, try new things, see what happens;

the changes will happen almost immediately, and there is no risk of

death or mayhem. The easiest way to learn new language features is to

take some code that works one way, and change it.

3. Write your Own Code as Soon as Possible

Once you understand something about the language--or even if you're

still getting your head around it--start writing sample programs that

use it. Sometimes it's hard to find good ideas for what programs to

write. That's OK, you don't have to come up with every idea at the

beginning.

You can find some programming challenges on this site.

You can also reimplement the examples from the book or tutorial you are

reading. Try to do so without looking back at the sample code; it won't

be as easy as it seems. This technique can work especially well if you

tweak the sample code.

If you can't think of a small program to write, but you have in mind

a larger program you want to implement, like a game, you could start

building small pieces that you can later use for a game. Whether you

use them later or not, you will get the same useful experience.

4. Learn to Use a Debugger

I already talked about the importance of debugging in The 5 Most Common

Problems New Programmers Face--And How You Can Solve Them. But it bears

repeating; the sooner you learn good debugging techniques, easier it

will be to learn to program.

The first step in doing so is to learn how to use a tool called a

debugger, which allows you to step through your code.

A debugger will allow you to step line by line through a piece of code. It

will let you see the values of variables, and whether the code inside

an if statement is executed.

A debugger can help you quickly answer questions about what your code

is doing.

int main()

{

int x;

int y;

if( x > 4 ) // <-- what is the value of x here?

{

y = 5; // <-- did this line of code execute?

}

}

A final word about debuggers: the first time you learn about a debugger,

it will take you longer to fix the problems with your code. After the

tenth or so bug, it will really start to pay off. And believe me, you

will have way more than ten bugs in your programming career.

I often saw students unwilling to use a debugger. These students really

made life hard on themselves, taking ages to find very simple bugs. The

sooner you learn to use a debugger, the sooner it will pay off.

5. Seek out More Sources

If you don't understand something, there's a good possibility the way

it was explained just didn't click.

First, look for alternative explanations. The internet is filled with

information about programming, and some explanations work better for

different people; you might need pictures, someone else might not. There

are also lots of good books with detailed explanations.

But if that doesn't work, the easiest way to figure out where your

misunderstanding lies is to ask someone else. But try to go beyond

saying, "I don't understand. Please explain." You're likely to get a

link back to the same text you didn't understand. Instead, rephrase your

understanding of the text in your words. The more your question reveals

about what you are thinking, the easier it will be for a knowledgeable

expert to answer it. Programmers sometimes have a reputation for being

grumpy about answering questions, but I think the reason is that they

want to make progress in a conversation, and that requires both sides

to put in effort. If you ask a smart, detailed question that shows you

are thinking, you will generally get good results.