Block 4:

SUMMARY

FUNCTIONS, PROTOTYPES, DECLARATIONS, ARGUMENTS, AND RETURN VALUES.

READ CHAPTER 4 Dividing your Work with Functions

UNDERSTAND LOGICAL OPERATORS

LOGIC CAN GET VERY COMPLEX.

BEGINNERS' PROGRAMS ARE OFTEN UNNECESSARILY COMPLICATED.

PROGRAMS ARE BUILT FROM FUNCTIONS. FUNCTIONS HAVE

TYPES LIKE VARIABLES, (SINCE THEY RETURN VALUES, AND

VALUES HAVE TYPES), AND ARE OFTEN USED JUST AS

VARIABLES ARE.

VARIABLES AND FUNCTIONS HAVE "SCOPE". SCOPE IS THE RANGE OF

LOCATIONS IN THE PROGRAM WHERE THE VARIABLE OR FUNCTION CAN

BE SEEN AND REFERENCED.

WHEN SEVERAL FUNCTIONS HAVE THE SAME NAME, BUT DIFFERENT

ARGUMENT LISTS, THEY ARE "OVERLOADED". THAT MEANS,

SEVERAL FUNCTIONS HAVE THE SAME NAME.

WHEN A FUNCTION CALLS ITSELF WE SAY IT IS "RECURSIVE".

READ CHAPTER Functions and an Introduction to Recursion /\*\* TODO validate chapter title

END SUMMARY

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

Block 4:

Read Chapter 4. Dividing your Work with Functions

This is a continuation of the previous lecture.

for(;;), while(), do .. while() ;

if .. else , switch() { case () : ... , case () :} ;

break; continue ;

logical AND && , logical OR ||

Equality == vs Assignment =

You can construct puzzles using if(), the Relational Operators

>, >=, !=, ==, <, <=, which nobody can solve. The idea is

\_not\_ to do so.

Which is easier to understand:

if ( x == 4 && y > x)

cout << x << endl ;

(or:)

if (x == 4)

{

if (y > x)

{

cout << x << endl ;

}

}

Generally, keeping "conditional operations" short,

and breaking them up on separate lines makes it

easier to follow the logic. In most cases,

BIG CONDITIONAL OPERATIONS indicate that the programmer

hasn't thought about the problem long enough before

starting to write the program.

General Rules:

1. Handle the commonest, or most likely case first, then

go back and handle the less-likely exceptions.

2. If you have a condition which can only go one of two

ways, test for one, and if that test is false, ASSUME

THE OTHER, DON'T TEST FOR BOTH OPTIONS, SINCE IF IT'S NOT

ONE, IT MUST BE THE OTHER.

(Here's an example:)

Wrong way:

if (x == 0)

cout << "X is 0!\n" ;

else if (x != 0) // unnecessary.

cout << "X is non-0!\n" ;

Right way:

if (x == 0)

cout << "X is 0!\n" ;

else

cout << "X is non-0!\n" ;

If it's not true, it must be false.

(Note in the examples above I used "\n" not std::endl.

They both evaluate to the same thing, a newline.)

===========

Some background for logic like that:

'. . [The] test looked simple: One lever, two pushbuttons, two colored

lights, two little gates. Once he mastered the instructions it would be

as easy as flying a kite, and a durn sight simpler than flying a copter!

Matt had had his copter license since he was twelve. He got to work.

First, he told himself, there seems to be just two ways to make a score,

one with the red light on and one with both lights out and one gate open.

'Now for the other instructions Let's see, if the left hand gate is not

closed...no, if the left hand gate is closed...he stopped and read them

over again.

'Some minutes later he had sixteen possible combinations of gates and

conditions of lights listed. He checked them against the instructions,

seeking scoring combinations. When he was through he stared at the result,

then checked everything over again.

'After rechecking, he stared at the paper, whistled tunelessly, and

scratched his head. Then he picked up the paper, left the booth, and

went to the examiner.

'That official looked up. "No questions, please."

'"I don't have a question," Matt said. "I want to report

something. There's something wrong with that test. Maybe the wrong

instruction sheet was put in there. In any case, there is no possible

way to make a score under the instructions that are in there."'

The officer checked his watch, and made a mark on the sheet. "Very well.

That's all, you can go."

"But what about my score on the test?" Matt asked.

"Your score has been recorded. Proceed to the next test."

R. A. Heinlein, "Space Cadet", pp. 19-20

===========

The technique Matt Dodson used to analyze that test is exactly the same

kind of thinking a logical mind uses to evaluate any system:

1. Describe it exhaustively using a description of all the ways

it can "be". We call this "describing system states".

2. Look at the states described and find which ones you desire

and which you wish to avoid.

3. If the game is rigged, "stage a revolution." (Go outside

the boundaries of the "game" to seek equity. In programming,

this involves re-writing the logic of the progam so it's no

longer broken. Ref: "Kobayashi Maru," STTWOK.)

The components of a C++ program are generally FUNCTION CALLS. There are

two kinds of functions: "member" functions found in some class, and

"global" functions found in your program or in some library you're

linking to.

Example:

#include <cmath>

double d = sqrt(3.14159) ;

SQRT() is in the "math library" and the compiler doesn't know about it

unless you declare it.

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

Functions with parameters:

int sumprod3 (int a, int b, int c)

{

return (a + b) \* c ) ;

}

WHAT YOU MUST UNDERSTAND: the ORDER of the parameters

is crucial. If you write:

cout << sumprod3(1, 2, 6) ;

then 1 and 2 will be added, and their sum multiplied by 6. The Parameters

are "positional", so the ORDER of the \_arguments\_ (the actual numbers)

determines which argument is stored in which parameter. See?

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

Function Prototypes "declare" functions, and tell the compiler what

TYPE of value the function returns, and what TYPES of values it wants

in its arguments.

C Standard Library Header Files are SOURCE CODE FILES that declare and

define a lot of variables, functions, macros, etc., that are useful

in programming.

Storage Classes are ways of telling the compiler about the life span of

a variable, and how it should be stored in the computer's hardware.

The default is auto for "automatic" in which the system uses a "standard"

form of storage.

Options are: auto, register, extern, volatile, static.

auto (the default) Variables in functions go away when

the function ends.

register A request the compiler doesn't have to honor.

Asks for a super-fast variable with no address. Used to make loops

run faster. You may not use the "address of" (&) operator with a

register variable.

extern Doesn't DEFINE a variable, just tells the

compiler to shut up, the variable is defined outside the current scope,

let the linker find it.

static (as in static int x = 0 ; )

Variables in functions are only initialized the first time, don't go away,

retain their values between function calls.

BEAR IN MIND that generally you won't use any of the above. Most programs

use ordinary auto variables to solve ordinary problems. Only large and

complex code requires messing with storage classes.

const A Named Constant. Once you create and initialize

it, it cannot be changed during the program run.

SCOPE: where a variable or function may be "seen". If a variable

hasn't been "defined" (allocated and named) or "declared" (an external

declaration affirmed locally) in the current "scope", any attempt to

use its name will result in an error.

Read: <A

HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/scoping\_of\_variables">scoping\_of\_variables</A>

Text: Chapter 5

Old languages made all variables "GLOBAL", i.e., visible everywhere.

This caused bugs. Strong programs limit visiblity of variables as much

as possible. Object Oriented Programs limit many variables to inside

a "black box" ("Encapsulation"). If other programmers can't SEE the

variables, they can't MESS UP the variables. It is possible to have

global variables in C++, you define them outside any function.

Usually, it's wrong.

Global variables are a shortcut. The state of mind that takes shortcuts

produces weird, subtle bugs that take weeks to find and never get

fixed right. Clear thought, and clean design come from proceeding

slowly, and really understanding what you are trying to do. One lesson

history teaches us: using shortcuts in any field of engineering gets

people killed. Even in programming. Do a Web search on "Therac-25"

for examples: many human beings killed due to system errors resulting

from flawed design in a machine which "couldn't" hurt anybody.

volatile: One opposite of const.

The other, "automatic" is the default kind of variable. A VARIABLE,

by definition, "varies". Volatile variables are so declared in programs

with multiple "threads".

Trying to explain \_that\_ this early would give us both a headache, but if

I must: In thread-based multiprogramming, it warns the compiler that it

may change even though no code in the current scope touched it. Warns the

compiler not to make any assumptions about it, or to optimize it. If that

makes any sense to you, you should probably be teaching this course:).

Don't worry about it.

Look at the definitions of the STACK and the HEAP.

Stacks are built into the CPU hardware, and are very powerful data

structures. The system stack is where arguments are passed to functions.

We also implement stacks in software, since they are powerful ways to

control programs and data. You will see them a lot in Data Structures and

Assembly Language classes. All you need to know is, if you allocate lots

of array variables, and have many cases where functions call themselves,

the System Stack may crash into the System Heap, you will be out of

memory, and the program will abort.

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

Function Overloading:

When two or more functions have the same name but different parameter

lists, they are said to be "overloaded". The compiler will figure out

which one to call by inspecting the types of the values or variables

you pass to the function when you call it.

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

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

Function Templates first look

While Overloading via Polymorphism provides a generalized view of

the tools you use, a much more generalized means of programming uses

Templates.

template < class T >

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

\* template02.cpp - show templates and generalized programming

\*

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

#include <cstdio>

#include <cctype>

#include <iostream>

#include <iomanip>

using namespace std ;

template < class T >

T maximumt( T value1, T value2)

{

T maxval = value1 ;

if (value2 > maxval ) // this works for string objects NOT for CStrings.

maxval = value2 ;

return maxval ;

} // end template

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

{

int a = 9 , b = 40 ;

cout << "max of " << a << " and " << b << " is: " << maximumt(a,b) << endl ;

string sa = "Allow" , sb = "Broadway" ;

cout << "max of " << sa << " and " << sb << " is: " << maximumt(sa,sb) << endl ;

} // main ends

The reason the statement:

if (value2 > maxval)

makes sense in the code above, is that '>' already knows how to compare

two numbers of the same type, and it has been OVERLOADED to be able to

compare the contents of two string objects. But if you pass in two arrays

of char (CStrings), '>' has not been overloaded to call strncmp(), and

will instead only compare their addresses, since the name of an array

is the address of its origin.

One template that is useful is the "set":

"The STL container class set is used for the storage and retrieval of

data from a collection in which the values of the elements contained

are unique and serve as the key values according to which the data is

automatically ordered. The value of an element in a set may not be changed

directly. Instead, you must delete old values and insert elements with

new values."

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

Recursion:

This is a very powerful programming technique which often allows a large

program to be re-written as a much smaller one. The idea is simple: a

function is recursive when it calls itself. The older (but still vital)

LISP programming language was designed around heavy use of Recursion.

Using

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/recursion">recursion</A> in C++ is truly 'leet.

04LECTURE.NOTES

FUNCTIONS

Open text to chapter on Functions.

Functions break up a program into a main (the driver routine)

and several sub-programs each of which does one thing only.

Suppose you were running secret agents, and you needed to

issue them numbers to use instead of their names. You decide

to add up the letters in their names to get the numbers,

using four numbers: their health: First Name Number, Middle

Name Number, Last Name Number, and Birth Date number.

That would mean, you'd want to calculate the numbers from

four different inputs. It would be silly to

read first name

calculate total

read middle name

calculate total

read last name

calculate total

read birth date

calculate total

...and DUPLICATE THE CODE to "calculate the total"

four times.

Suppose you found out the "calculatetotal" code

was flawed? YOU'D HAVE TO MAKE THE SAME CORRECTION

IN FOUR PLACES IN YOUR PROGRAM! That's not good

program structure.

INSTEAD: yould write one function to do the calculation,

and CALL it from four different places in your code.

That's why we break a program up into functions:

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

\* calculate\_total.cpp - illustrate functions.

\* we're pretending that we need to find "word totals" in order

\* to develop totals for our Secret Agents.

\* Thu Jan 2 09:31:33 PST 2014

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

#include <cstdio>

#include <cctype>

#include <iostream>

#include <iomanip>

#include <cstdlib>

using namespace std ;

int calculate\_total (string input)

{

int total = 0 ;

for (int i = 0 ; i < input.length() ; ++i)

total += input[i]-'0' ;

return total ;

} // calculate\_total ends

int main ()

{

int fn, mn, ln, bd ;

string buffer ;

cout << "Enter your first name: " ;

cin >> buffer ;

fn = calculate\_total(buffer) ;

cout << fn << endl ;

cout << "Enter your middle name: " ;

cin >> buffer ;

mn = calculate\_total(buffer) ;

cout << mn << endl ;

cout << "Enter your last name: " ;

cin >> buffer ;

ln = calculate\_total(buffer) ;

cout << ln << endl ;

cout << "Enter your birthdate mm/dd/yyyy: " ;

cin >> buffer ;

bd = calculate\_total(buffer) ;

cout << bd << endl ;

cout << "agent number: "

<< fn

<< mn

<< ln

<< bd

<< endl ;

return EXIT\_SUCCESS ;

} // main ends

PASSING ARGUMENTS:

WHEN YOU PASS ARGUMENTS TO A FUNCTION, YOU ONLY PASS COPIES, not the

actual arguments. The function cannot change the values of the

agruments in the calling program, it can only change the copies it

receives.

EXCEPT FOR ARRAYS. If you pass an array, the function can change the

array.

Read the chapter on "pass by value" and "pass by reference". Discuss.

STRING OPERATIONS.

Strings are Class Objects. They are examples of Object Oriented Programming.

You don't have to worry about that, just understand that an Object can

contain both DATA and CODE (functions).

class Mage might contain int hit\_points, string Name, and int damage.

But it also might contain void cast\_lightning\_bolt(), void heal(),

and void teleport( string Destination) ;

It's handy to be able to "package up" data and code, it makes it easier

to write huge programs like MMORG games.

But in general, Objects aren't needed for the small programs we're going

to write this semester, so don't stress.

The exception is strings, since they're so much easier to use than

Arrays of Characters, which is all C had.

string functions (since it's an object, we call the functions Methods...

sure we do.)

string functions we use a lot are:

length() returns an int: length of the string's character array

erase() removes part of a string

replace() changes some of a string

c\_str() returns an array of char containing the string

We use c\_str() a LOT, since we often program using functions that have

been with us since C, and were written to operate on char x[[] style

arrays. They can't process string objects. So we have c\_str() in

the string object which creates an array of char containing its text,

which can be sent to those old functions like strrev(), strtok(), etc.

Function Arguments revisited: main()'s args.

Read the section on the parameters which the Operating System provides

to main() when it starts up:

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

/\*

\* fact.c - demonstrate recursion, with factorials, wherein a function

\* calls itself (Additionally, this program shows how

\* easily overflow can sneak into a C program.

\* Mon May 24 18:33:43 PDT 2004

\* Modified on Mon Mar 5 09:42:07 PST 2007 to show how to insert

\* commas into numbers.

\*/

#include <stdio.h>

#include <iostream>

#include <cstdio>

#include <algorithm>

#include <string>

using namespace std ;

unsigned long long fact(int) ;

char \* commas( char \* total, int groupsize) ;

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

{

if (argc < 2) { puts("fact nn") ; exit(-1) ; }

int i, limit = atoi(argv[1]) ;

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

{

char buffer[1024] ;

sprintf(buffer, "%Lu", fact(i)) ;

//printf("fact of %d : %Lu\n", i, fact(i)) ;

printf("fact of %d : %s\n", i, commas(buffer, 3)) ;

}

}// End main()

unsigned long long fact (int n) {

if (n == 1)

return (unsigned long long) 1 ;

else

return (unsigned long long) (n \* fact(n - 1)) ;

} // End fact()

char \* commas( char \* total, int groupsize)

{

string stotal(total) ; // create a string from a CString.

int i, place = 0 ;

string d, s = stotal ;

reverse(s.begin(), s.end()) ; // reverse the chars in the string

// insert commas into the ouput stream of digits.

for ( i = 0 ; i < s.length() ; i++)

{

d.append(s.substr(i,1)) ;

place++ ;

if (place % groupsize == 0 && i < s.length() -1) // no leading commas

{

d.append(",") ;

}

}

reverse(d.begin() , d.end() ) ; // reverse the string again

// I return a CString so even old-style C code can use this function with change.

return (char \* ) d.c\_str() ; // convert to CString and return

} // commas ends

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

\* overloadfcn001.cpp - functions with same name

\* when two functions have the same name and different

return or parm types, they are OVERLOADED. This is a case

of "polymorphism".

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

#include <cstdio>

#include <iostream>

#include <cstdlib>

using namespace std ;

double quotient(double a, double b) ;

int quotient(int a , int b) ;

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

{

cout << "double: " << quotient ( 22.0 , 7.0 ) << endl ;

cout << "double: " << quotient ( 22 , 7 ) << endl ;

return EXIT\_SUCCESS ;

} // main ends

double quotient(double a, double b)

{

return a / b ;

}

int quotient(int a , int b)

{

return a / b ;

}

RECURSION:

Recursion is a programming technique where a routine calls itself.

Definition: recursion (rek'urzhion) n., see: recursion.

The classic illustration of recursion is programming the Factorial

function. In combinatorics, a factorial is a number multiplied by all

the numbers less than itself and greater than zero.

So 7! = 7 \* 6 \* 5 \* 4 \* 3 \* 2

An easy way to write a factorial is to have the function call itself

until it's playing with 1, then start returning up the stack, and

multiplying....well, just read the code:

/\*

\* recursion.c - demonstrate recursion, wherein a function calls itself

\* (Additionally, this program shows how easily overflow can

\* sneak into a C program.

\* Mon May 24 18:33:43 PDT 2004

\*/

unsigned long long fact(int) ;

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

{

int i ;

char s[1024] ;

for (i = 1 ; i < 35 ; i++)

{

printf("fact of %d : %Lu\n", i, fact(i)) ;

}

}// main ends

unsigned long long fact (int n)

{

if (n == 1)

return (long long) 1 ;

else

return (long long) (n \* fact(n - 1)) ; // fact() calls itself here!!

}