**INTRODUCTION TO FILE INPUT**

* We can use cin to read from a file in a manner very similar to reading from the keyboard.
* We are gonna use stream insertion (<<) and stream extraction (>>) operators to read files and to write to files.

player.txt (we are in the same directory) 🡪 100510

Gordon Freeman

#include <iostream>

#include <fstream> //filestream, to be able to use files

#include <string>

using namespace std;

int main()

{

string firstName, lastName;

int score;

fstream inputStream; //inputStream is fstream object which will be my file

inputStream.open(“player.txt”); //inputStream is now connected to player.txt

//You can append these to and you open file for input and output and appending:

//inputStream.open(“player.txt”, fstream::in | fstream::out | fstream::app);

//with this, if file is not exist in the same directory, file will be created because you said //fstream::out

inputStream >> score;

inputStream >> firstName >> lastName;

cout << “Name: “ << firstName << “ “

<< lastName << endl;

cout << “Score: “ << score << endl;

inputStream.close();

return 0;

}

fstream, cout, cin, inputStream are kind of relatives. They come from the same ancestor and that’s why they share lots of genes.

fstream

iostream

istream

ios

ios\_base

ostream

cin is an istream (input stream)

They may be connected to screen or to files.

cout is an ostream (output stream)

cin has getline method:

char name[256];

cin.getline(name, 256); 🡪 read up to 256 characters

getline doesn’t have the string, it has character array.

getline returns an istream (we will talk about this later).

**FUNCTION BASICS**

What we will learn:

* Predefined Functions / Library Functions
  + Those that return a value and those that don’t
* Programmer-defined Functions
  + Defining, Declaring, Calling
  + Recursive Functions
* Scope Rules
  + Local variables
  + Global constants and global variables
  + Blocks, nested scopes

Abstract away the details when you are using functions like sqrt.

Functions are building blocks of programs.

Other terminology in other languages:

* Procedures, subprograms, methods
* In C++: functions

I-P-O

* Input - Process - Output
* Basic subparts to any program
* Use functions for these “pieces”

In C++, we don’t like writing or using global functions. That’s why we don’t like printf, scanf, open function that opens the file.

You have main function which is global because C++ is not a pure OOP language. It is keeping the compatibility with C. To be compatible with C, you have to have main function as a global function.

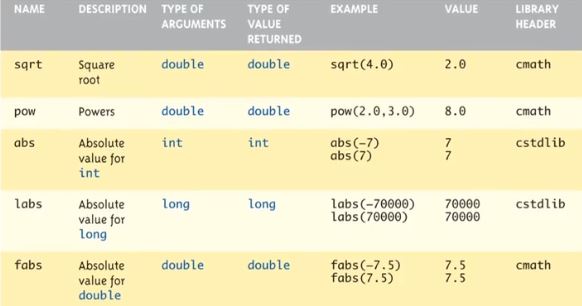
Other than the main, don’t write any global functions. Always use classes. If you are writing a function, that function works on that data, it doesn’t need parameters from the outside world.

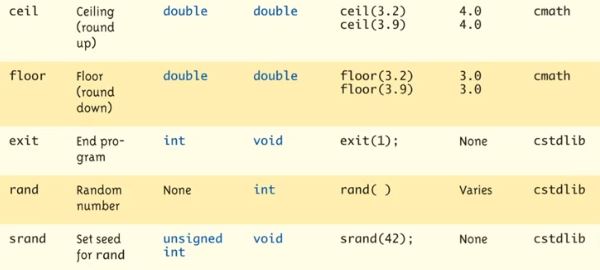
**Predefined Functions**

* Libraries full of functions for our use.
* Two types:
  1. Those that return a value
  2. Those that do not (void)
* Must “#include” appropriate library
  + e.g.,
    - <cmath>, <cstdlib> (Original “C” libraries, all C libraries are available in C++)

to be able to use C libraries in C++, you should add “c” letter before the library name

* + - <iostream> (for cout, cin)
* **Math** functions very plentiful
  + Found in library <cmath.h> (you include <cmath> in C++)
  + Most return a value (the “answer”)
* Example: theRoot = sqrt(9.0);
  + Components:
    - sqrt = name of library function
    - theRoot = variable used to assign “answer” to
    - 9.0 = argument or “starting input” for function
  + In I-P-O:
    - I = 9.0
    - P = “compute the square root”
    - O = 3, which is returned & assigned to theRoot
* pow(x,y)
  + Returns x^y





* #include <cstdlib>
  + Library contains functions like:
    - abs() //Returns absolute value of an int
    - labs() //Returns absolute value of a long int
    - fabs() //Returns absolute value of a float
  + fabs() is actually in library <cmath>!
    - Can be confusing
    - Remember: libraries were added after C++ was “born,” in incremental phases
    - Refer to appendices/manuals for details
    - in cmath library, functions take double and return double so if a function takes another type it is not belong to cmath library
* Predefined void functions
  + No returned value
  + If a function does something in it and it has an effect, we call it side effect.
  + Performs an action, but sends no “answer”
  + When called, it’s a statement itself
    - exit(1); //No return value, so not assigned
      * This call terminates program
      * void functions can still have arguments
  + All aspects same as functions that “return a value”
    - The just don’t return a value
    - What is returned from main? 🡪 echo $0

**Programmer-Defined Functions**

* Write your own functions
* Building blocks of programs
  + Divide & Conquer
  + Readability
  + Re-use
* Your “definition” can go in either:
  + Same file as main()
  + Separate file so others can use it, too
* Functions are “equals”; no function is ever “part” of another

Function declaration;

also called the function prototype

#include <iostream>

using namespace std;

double totalCost(int numberParameter, double priceParameter);

//Computes the total cost, including 5% sales tax,

//on numberParameter items at a cost of priceParameter each.

int main()

{

double price, bill;

int number;

cout << “Enter the number of items purchased: “;

cin >> number;

cout << “Enter the price per item $”;

cin >> price;

bill = totalCost(number, price);

cout.setf(ios::fixed);

cout.setf(ios::showpoint);

cout.precision(2);

cout << number << “ items at “

<< “$” << price << “ each.\n”

<< “Final bill, including tax, is $” << bill

<< endl;

return 0;

}

double totalCost(int numberParameter, double priceParameter);

FUNCTION HEAD

{

const double TAXRATE = 0.05; //5% sales tax

FUNCTION DEFINITION

double subtotal;

FUNCTION BODY

subtotal = priceParameter \* numberParameter;

return (subtotal + subtotal \* TAXRATE);

}

**Enter the number of items purchased: 2**

**Enter the price per item: $10.10**

**2 items at $10.10 each.**

**Final bill, including tax, is $21.21**

In C, if you didn’t define the function at the top, it compiles bc in C to be able to use a function, main doesn’t have to see it’s prototype or the header or the declaration. When you make function call, C compiler tries to find a function declaration first. If it is available, than assume that it is there. If it is not available, than assume that there is a function that come later and I don’t know how many parameters it takes, I don’t know what kind of parameters it takes and I will assume that it is going to return an integer and it compiles that way. If the linker finds that function everything works.

In C++, if you don’t include declaration, when compiler sees function call, then you’ll get an error message.

*This is a global function, we don’t like global functions in C++ but we haven’t seen the classes yet.*

Common with C way of doing things, we usually keep our functions in a separate file (because functions are our black boxes, we are trying to abstract away some details). For those functions, we write a header file. So that those functions could be used from different files.

file.cpp tc.h tc.cpp

#include <io…>

double tc (int, double);

int main(){…

tc();

}

double tc(int…)…..

I need to include tc.h to file.cpp and compile my code with tc.cpp.

Can you include tc.cpp to file.cpp?

* Yes you could but you shouldn’t.
* tc.cpp is implementation. It shouldn’t be compiled with your customer’s code. Implementation should be kept in different place bc I don’t want to be depending on implementation file. I only care about the function declaration. If you change your implementation, file.cpp doesn’t have to be compiled again.

g++ -c file.cpp 🡪 produce file.o

g++ -c tc.cpp 🡪 produce tc.o This is how you compile

g++ -o myProg file.o tc.o 🡪 linking

* Then if you have a decision that there is some improvement in the implementation of the tc.cpp
* If I change a single line in tc.cpp, I just compile tc.cpp and link them again. I don’t have to compile file.cpp if I include tc.h not tc.cpp
* If you include tc.cpp to file.cpp, then I have to compile file.cpp too bc it is included in there.
* If you change something in tc.h, then you have to compile both file.cpp and tc.cpp but headers are rarely change.
* And also customer doesn’t care how the implementation works.

You can compile everything and link in 1 line:

g++ \*.cpp 🡪 a.out will be our executable program

We don’t prefer this because large projects have hundreds of C++ files. If one of the developers made a change in it, you don’t want to compile everything. You want to compile only 1 file and link it.

So are you gonna type all of them one by one to compile? 🡪 No, you have makefiles.

Function declaration is “information” for compiler

Compiler only needs to know:

1. Return type
2. Function name
3. Parameter list

In declaration, parameter names not needed. Still should put in formal parameter names, it improves readability.

Parameter : we use in declaration

Argument : we use when we call function

\*\*Compiler produces an executable code with the linker.

void functions job:

* show something
* write something to the file
* send some data through the network
* etc.

You should include “return;” statement at the end of the function but you don’t have to.

**Main Function**

* OS calls the main.
* You call your main function itself, it makes it recursive.
* Tradition holds it should have return statement
* Value returned to “caller” so to OS
* Should return “int”
* If there is no main function, your linker cannot produce an executable program so there has to be a main function.
* “echo $?” 🡪 get whatever main return with this command

\*\*SHELL is an interface between user and OS. Actually shell is calling your main function.

\*\*Bash is a shell (for unix/command line shell).

\*\*Responsibility of bash is taking commands from user and send it to OS.

\*\*For windows or that kind of graphical OS, shell is different. 🡪 shell is explorer

**Scope Rules**

* Local variables
  + Declared inside body of given block
  + Available only within that block

int g(int k){

{ int i = 0;

cout << i; 1 BLOCK, int i is valid only in this block

}

2 different variables

{ double i = 0.0;

cout << i; 1 BLOCK, double i is valid only in this block

}

return 1;

}

* Can have variables with same names declared in different functions/blocks
  + Scope is local: “that function is it’s scope”
* Local variables preferred
  + Maintain individual control over data
  + Need to know basis
  + Functions should declare whatever local data needed to “do their job”

for (int i = 0; i < SIZE; ++i){

… Scope of i. After for loop, i is gone.

}

**Procedural Abstraction**

Need to know “what” function does, not “how” it does it.

User of function only needs: declaration, does not need function definition 🡪 information hiding

Encapsulation 🡪 abstraction of functions along with the data itself

We don’t hide information because of the security issue, we hide information because it is not useful for the user of that function.

**Global Constants and Global Variables**

* Declared outside function body
  + Global to all functions in that file
* Declared inside function body
  + Local to that function
* Global declarations typical for constants: 🡪 global consts are okay bc nobody can change it
  + const double TAXRATE = 0.05;
  + Declare globally so all functions have scope
* DON’T USE GLOBAL VARIABLES

don’t think about main function is the first function to be called

if this global declaration was:

int size = f(2);

in order to initialize the size, the function f has to be called because i have to initialize that global variable before the main

so f is called before the main

* + They are dangerous

#include …

…

int size = 10; 🡪 initialized before the main function is called

int f(){

size = 7;

…

}

int g(){

…

cout << size;

…

}

Depending on who is called first f or g, the value printed on screen will change.

**Blocks**

* Declare data inside compound statement
  + Called a block
  + Has block-scope
* Note: all function definitions are blocks
  + This provides local “function-scope”
* Loop blocks:

for (int ctr = 0; ctr < 10; ctr++)

{

sum+=ctr;

}

cout << ctr;

//Variable str has scope in loop body block only

if (a==b){

int k = 7; You can do this.

}

1.0 / 3.0 \* 3.0 🡪 this may not return you a perfect 1.0

Never use doubles to represent money!

**PARAMETERS AND OVERLOADING**

* Parameters:
* Call-by-value (“copy” of value is passed) 🡪 C , C++
* Call-by-reference (“address” of actual argument is passed) 🡪 C++
* Simulated call by reference 🡪 C, C++
* Mixed parameter-lists (You could mix call by value and call by reference in a function call)

g is behaving like it is doing call by reference on i

In fact it is doing call by value on some pointer bc when you send address of i, you are not changing address of i. You are changing whatever the pointer points to that location. You are not changing ip, you are playing with whatever ip is pointing.

This is simulated call by reference

ip is copy of address of i

I’m not changing anything about ip, all I’m changing is that I’m following the pointer of ip and I am making modifications to that.

In fact, it is changing the value of i.

f is doing call by value on some integer

Copy of integer is made in f, so k is different than i

Whenever you modify k in f, i will not be modified

i and k are copies of each other

int main (){

int i;

i = 7;

g(&i);

f(i);

…

}

int g(int \* ip){

\*ip = \*ip + 1;

return \*ip;

}

int f(int k){

k++;

return k;

}

**Call-by-Value Parameters**

* Copy of actual argument passed
* Considered “local variable” inside function
* If modified, only “local copy” changes
  + Function has no access to “actual argument” from caller
* This is the default method
  + Used in all examples thus far
* It is expensive because it is copying the same value in memory
* Don’t use your parameter to do some calculations, use another variable for it. Don’t change your parameter.

double fee(int hoursWorked, int minutesWorked){

int quarterHours; 🡪 local variable

int minutesWorked; 🡪 NO!

}

Compiler error results 🡪 “Redefinition error…”

Value arguments are like “local variables” but function gets them automatically

**Call-by-Reference Parameters**

* Used to provide access to caller’s actual argument
* Caller’s data can be modified by called function!
* If you send a reference, there is no copying.
* Typically used for input function
  + To retrieve data for caller
  + Data is then “given” to caller
* Specified by ampersand, &, after type in formal parameter list
* You can use call by reference in any types including user defined types like vectors or strings
* Faster than call by value if your data is large. If your data is integer or double or Boolean, it is the same thing.
* What is happening with the call by reference in assembly is same as passing a pointer and follow those pointers. Compiler is doing the work for you. Compile is doing the call by reference simulation for you.
* Cost of sending a call by reference parameter is the same as sending a pointer. This is not any faster than simulated call by reference but a little bit more convenient.

References are not new variables, they don’t occupy memory.

References are reference to variables.

int a = 4;

int& ref = a;

You just get 1 variable. You can use ref as it was a.

If you do “ref = 2;” , a will be 2.

YOU CAN’T CHANGE REFERENCE. For example if you have another variable b (int b=3;), you can’t do “ref = b;”. You just assign a to 3 with this code.

***PREFER CALL BY REFERENCE***

void getNumbers(int& input1, int& input2);

//Reads 2 integers from the keyboard. Function takes 2 integers by reference.

void swapValues(int& variable1, int& variable2);

//Interchanges the values of variable1 and variable 2

formal parameter

int main(){

int firstNum, secondNum;

swapValues(7, 8);

//Doesn’t make sense, compiler will produce an error message (You cannot get a

//reference of 7 or 8).

getNumbers(firstNum, secondNum); argument

swapValues(firstNum, secondNum);

//if firstNum or secondNum was const int, then these functions would give you error

//messages (**we cannot produce references from constants**)

//If parameters are const int & for swapValues, then you will get error messages in here

}

void getNumbers(int& input1, int& input2){

cout << “Enter two integers: “;

cin >> input1 actually you are modifying firstNum and secondNum

>> input2;

}

void swapValues(int& variable1, int& variable2){

int temp;

temp = variable1;

variable1 = variable2;

variable2 = temp;

}

void getNumbers(int\* i1, int\* i2){

cout << “Give me 2 ints\n”;

cin >> \*i1 >> \*i2;

}

void swapValues(int\* i1, int\* i2){ Simulated call by reference

int t = \*i1;

\*i1 = \*i2;

\*i2 = t;

}

i = f(a); 🡪 You can do in C and C++.

\*f(a) = i; 🡪 You can do in C and C++. If f is returning a pointer, you can dereference it and make this assignment.

f(a) = i; 🡪 You can’t do this in C, but you can do this in C++ bc if f is returning a reference, I can make an assignment to the references (int& f(int);).

int i;

int& ir = i; 🡪 You can define an integer reference but you have to initialize it bc after that point, when you modify ir (ir = 7;) i will change.

Usually we don’t do this that much because I can modify i itself, why do I define a reference for it. References are used generally when you are passing parameters.

**Call-by-Reference Details**

* In assembly, there is no call by reference. Everything is done by call by value.
* What’s going on in assembly is you are sending a hidden pointer. Compiler is handling all those complications like sending the pointer and whenever you refer to that variable, it just take the reference and etc.
* Don’t think that call-by-reference doesn’t have copying codes. It has copying codes but that copying codes are just simple as copying a pointer.
* A “reference” back to caller’s actual argument.
  + Refers to memory location of actual argument
  + Called “address”, which is a unique number referring to distinct place in memory
* Reference means pointer actually. Under the hood, pointers are working but you are not seeing them. So there is less chance to make a mistake, for example, forgetting the dereference or sending the address.

\*\*There is usually a stack pointer that points top of the stack, it knows where it is.

**Constant Reference Parameters**

* Reference arguments inherently “dangerous”
  + Caller’s data can be changed
  + Often this is desired, sometimes not
* To “protect” data, & still pass by reference:
  + Use const keyword
    - void sendConstRef( const int &par1,

const int &par2);

* + - Makes arguments “read-only” by function
    - No changes allowed inside function body

**Mixed Parameter List**

* Can combine passing mechanism
* Parameter lists can include pass-by-value and pass-by-reference parameters
* Order of arguments in list is critical:

void mixedCall(int& par1, int par2, double& par3);

* + arg1 must be integer type, is passed by reference
  + arg2 must be integer type, is passed by value
  + arg3 must be double type, is passed by reference
    - mixedCall(1, 2, 3.0); 1 and 3 are constants, can’t get address of them
    - mixedCall(a, 7, d); **OK**

**Choosing Formal Parameter Names**

* Functions as “self-contained modules”
  + Designed separately from rest of program
  + Assigned to teams of programmers
  + All must “understand” proper function use
  + OK if formal parameter names are same as argument names

**OVERLOADING**

You use plus operator in many places:

* 1 + 2 (This is actually same with “operator+(1,2)”)
* 1.2 + 1e-10

Meaning of these two operators in terms of assembly produces are very different.

This + operator is overloaded in C.

In C++ we say that:

* If the core of the language is using overloaded operators, why don’t we use the same idea for any kind of functions not just for operators?
* Use the same function name for different function. You might have more than one function named f which are doing different things.
* Same function name
* Different parameter lists
* Two separate function definitions
* Allows same task performed on different data

In C or C++ every function has a signature. You cannot have 2 functions having the same signature.

Every function should have a different signature.

**In C**, definition of signature is so simple 🡪 name of the function. That’s why you cannot have 2 functions that have same name.

int f(int);

int f(int, int);

**In C++,** signature is not just the name. Signature is:

* name + parameters it takes (type of the parameter + number of parameter + parameter order)

Their signatures are same bc their parameters and names are the same

👍

int f(double, string);

int f(double);

int f(string);

int f(double, double);

string f(double);

**Return type is not in the signature.**

f(int i) and f(const int i) are 2 different functions, their signatures are different.

* Only overload “same-task” functions
  + A mpg() function should always perform same task, in all overloads
  + Otherwise, unpredictable results
* C++ function call resolution:
  + 1st : looks for exact signature
  + 2nd : looks for compatible signature

**Overloading Resolution**

* 1st: Exact match
  + Looks for exact signature
    - Where no argument conversion required
* 2nd: Compatible match
  + Looks for “compatible” signature where automatic type conversion is possible:
    - 1st with promotion (e.g., int🡪double)
      * No loss of data
    - 2nd with demotion (e.g., double🡪int)
      * Possible loss of data

Given following functions:

1. void f(int n, double m);
2. void f(double n, int m);
3. void f(int n, int m);

These calls:

* f(98, 99); 🡪 calls 3
* f(5.3, 4); 🡪 calls 2
* f(4.3, 5.2); 🡪 calls ?

Should I demote 4.3 to 4 to call 1 or 5.2 to 5 to call 2? We don’t know. So avoid such confusing overloading.

If there are more than 1 way to convert your data to overloaded function, there will be a compiler error.

You can force the compiler with this call:

* f<int, double>(4.3, 5.2); 🡪 calls 1

This doesn’t make sense. If you are name your function which one you will call, why are you using overloading?

“operator+<double,double>(1,2)” 🡪 you can do this with the same logic

**Automatic Type Conversion and Overloading**

* Numeric formal parameters typically made “double” type
* Allows for “any” numerical type
  + Any “subordinate” data automatically promoted
    - int 🡪 double
    - float 🡪 double compiler knows these convertions
    - char 🡪 double
* Avoids overloading for different numeric types

sqrt(90); 🡪 sqrt takes double so automatically 90 is converted to a double and I will send a double to it

sqrt(90.); 🡪 don’t send int to sqrt, send double (90. is double)

**Default Arguments**

* Allows omitting some arguments
* Specified in function declaration/prototype

void showVolume( int length,

int width = 1,

int height = 1);

* Last 2 arguments are defaulted.
* In declaration. When you implementing the function, you just write:
  + void showVolume(int length, int width, int height);
  + Possible calls:
    - showVolume(2, 4, 6); //All arguments supplied
    - showVolume(3, 5); //height defaulted to 1
    - showVolume(7); //width and height defaulted to 1

Function doesn’t know if it gets those values by default arguments or real arguments.

DANGER!

If you overload showVolume function with this:

void showVolume(int i);

You are in trouble because if you call showVolume like “showVolume(7);“ this, compiler doesn’t know which one to call. So there will be a compiler error.

\*\*Being default doesn’t change the signature.

**TESTING AND DEBUGGING FUNCTIONS**

Many methods:

* Lots of cout statements
  + In calls and definitions
  + Used to “trace” execution

double g(int i, string s){

#ifndef NDEBUG //NDEBUG : no debug

cout << i << s << endl;

#endif

…

#ifndef NDEBUG

cout << d << endl;

#endif

return d;

}

If NDEBUG is defined, selected lines will not be compiled so I won’t see it.

For some reason, if I want to debug my program, I turn off NDEBUG, I don’t define it. So since it is not defined, then selected lines will be compiled and printed.

g++ -D NDEBUG 🡪 defines NDEBUG flag so selected lines will not be compiled.

So you are not making any changes to your program and cancel out all the couts and don’t work on deleting couts.

(g++ -D NDEBUG == #define NDEBUG)

g++ -o3 🡪 o is optimization flag. You are saying to compiler with this line that do whatever optimization you can. In terms of CPU and memory usage etc. With this, compiler automatically defines NDEBUG.

* Compiler Debugger
  + Environment-dependent
* **assert Macro :** macro which is handled by the preprocessor
  + Early termination as needed
  + #define SIZE 10 🡪 is a macro
  + #define SQR(X) X\*X 🡪 is a macro. whenever you see SQRT with a parameter in your

code, will be replaced by x\*x

a = SQR(5);

//preprocessor takes SQR(5) and replace it with 5\*5 🡪 a=5\*5;

BUT if you do this:

b = SQR(x+1); 🡪 b = x+1\*x+1; 🡪 DANGER

You should write like this : #define SQR(X) (X)\*(X)

BUT if you do this with this macro: #define DIFF(X) (X)-(X)

b = DIFF(a+5) \* 2; 🡪 b = (a+5)-(a+5)\*2;

You should write like this : #define DIFF(X) ((X)-(X))

SO BE CAREFUL!

* Assertion: a true or false statement

//precondition of f 🡪 i >= 0

int f(int i){

assert(i >= 0);

//when the program runs, everytime I hit this line, i value will be checked

//and code will make sure that i is greater than or equal to 0

//if it is not, then my program will be terminated and I will get a message

//saying that you have assertion in this line in this file.

}

-Why don’t you use this:

if (i<0){

cerr << “line 371”;

exit(-1);

}

assertions are compiled only if NDEBUG flag (“g++ -D NDEBUG f.cpp”) is not defined

OR you can simply add “#define NDEBUG” before “#include <cassert>”

* Used to document and check correctness
  + Preconditions & Postconditions
    - Typical assert use: confirm their validity
  + Syntax:

assert(<assert\_condition>);

* + - * No return value
      * Evaluates assert\_condition
      * Terminates if false, continues if true
* Predefined in library <cassert>, you should include it
  + Macros used similarly as functions
* Stubs and drivers
  + Incremental development
  + Separate compilation units
    - Each function designed, coded, tested separately
    - Ensures validity of each unit
    - Divide & Conquer
      * Transforms one big task 🡪 smaller, manageable tasks

I’m trying to achieve a compilable code

I’m after if the all system is compiling and linking fine, if I’m making any mistakes about parameters or etc..

int f1(double x, int y){

return 1;

}

* + Write “big-picture” functions first
    - Low-level functions last
    - “Stub-out” functions until implementation
    - Example:

double unitPrice(int diameter, double price)

{

return (9.99); //not valid, but noticeably a “temporary” value

}

Calls to function will still “work”

* + Driver programs are the opposite of stubs. You implement the all function.
    - main function tests your function with while loop as many as you like.
      * (Test again?: y 🡪 takes input and tests… Test again?: …)
    - This is called driver program.
* Unit-testing
  + Whatever you write in your program (your functions, your classes), each unit is tested separately.
  + You write your unit tests for each part of the program. That testing code calls your functions many many times with different parameters.
  + Let’s say you have 10 people in your development team. There are 100 files that you are working on. Somebody made a change in one of the file. After that change, all unit test for all the units of your program run by the robot. Usually it takes a night.