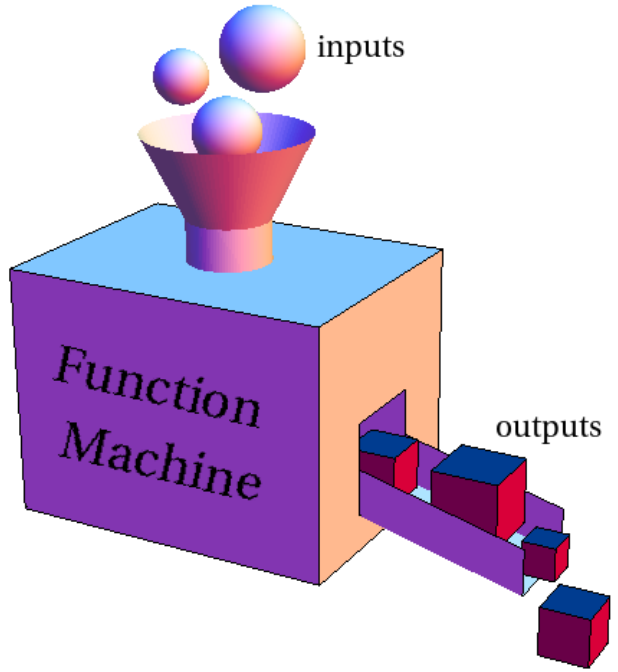
Functions

The Basis of C



iClicker Attendance

Please click on A if you are here:

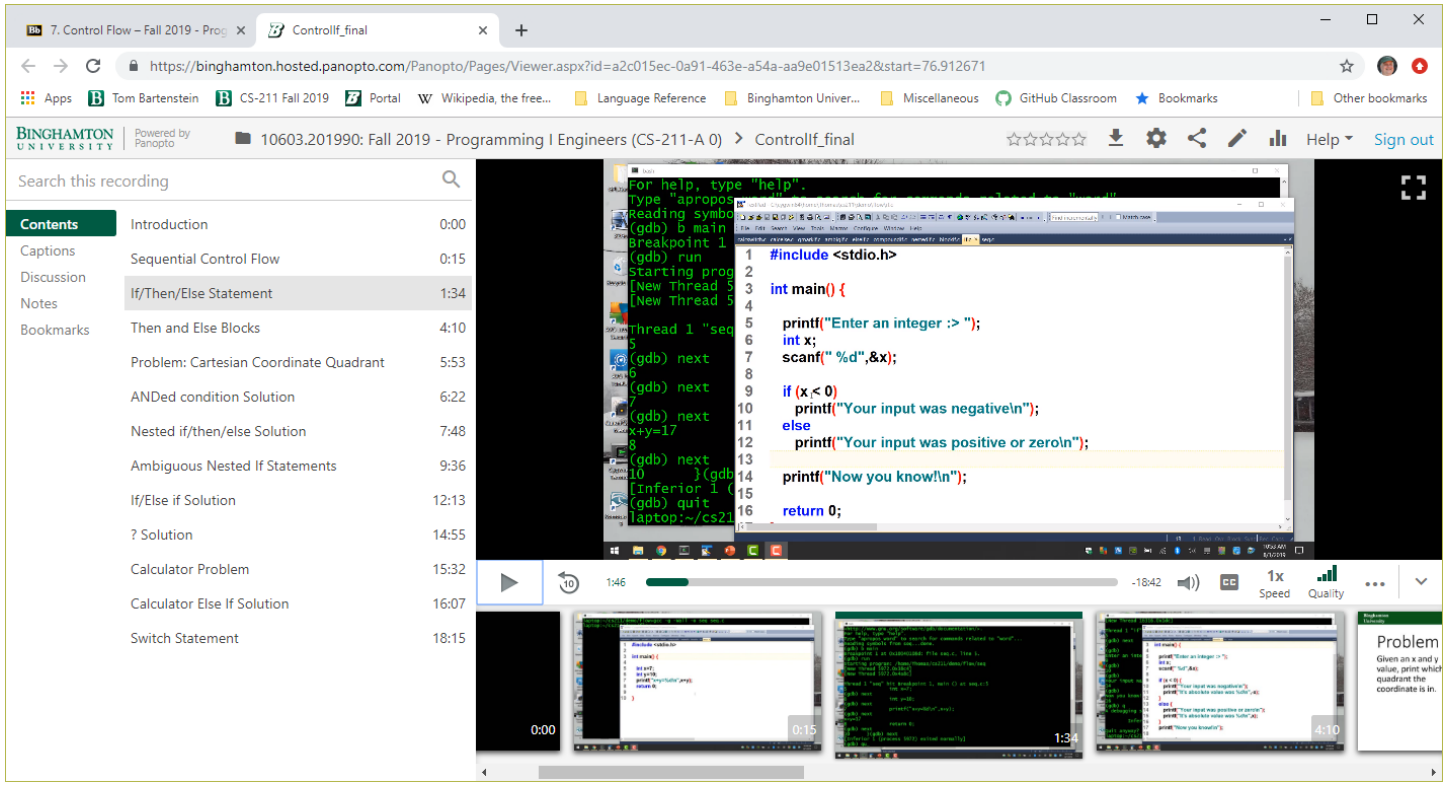
A. I am here today.

Exercise Review: HighLow

Write a computer program that selects a random number between 0 and 100. Then prompt the user to enter a guess of that number. As long as the user has not guessed the number, tell her whether her guess is greater than or less than the target number, then ask for another guess.

Video Review: If Statements

- Sequential Control Flow
- Simple “if”
- if/then/else
- Compound Conditions
- Nested if/then/else
  - else ambiguity
- if /else if construct
- switch statements



iClicker Question

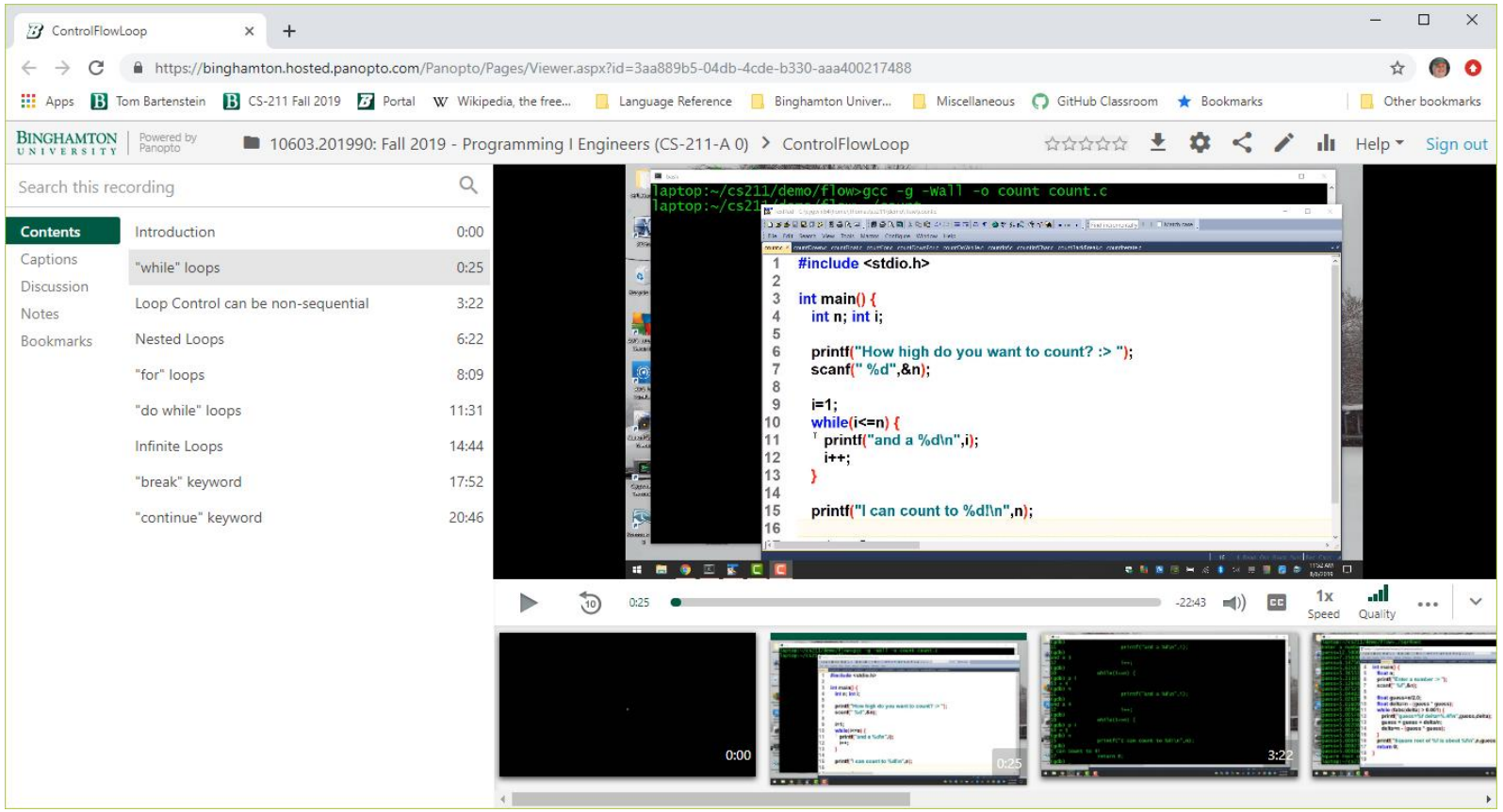
- What gets printed?

```
int a=1; int b=3;
if (++a>b)
if (a < b && a++==b) printf("a equals b ");
else printf("a not equal to b? ");
printf("a is %d, b is %d\n",a,b);
```

A. a is 3, b is 3
B. a equals b a is 3, b is 3
C. a not equal to b? a is 3, b is 3
D. a is 2, b is 3

Video Review: Control Flow: Loops

- while loops
  - nested while loops
- for loops
- do/while loops
- infinite loops
- break and continue



Class Exercise: Identify Duplicates

Given an array of grades, check to see if there are any duplicate grades. If so, write a message that specifies that index i is a duplicate of index j, where i<j. The array is declared as “int grades[100]”, so valid indexes are 0 to 99, and each element of the array can be accessed as “grades[i]”, where 0<=i<100.

What is a function?

- Have you ever needed to convert from C° to F°?
- Demo: http://www.pronk.com/samples/projects/021$Function_Machine/Function_Machine.HTML

Function in C

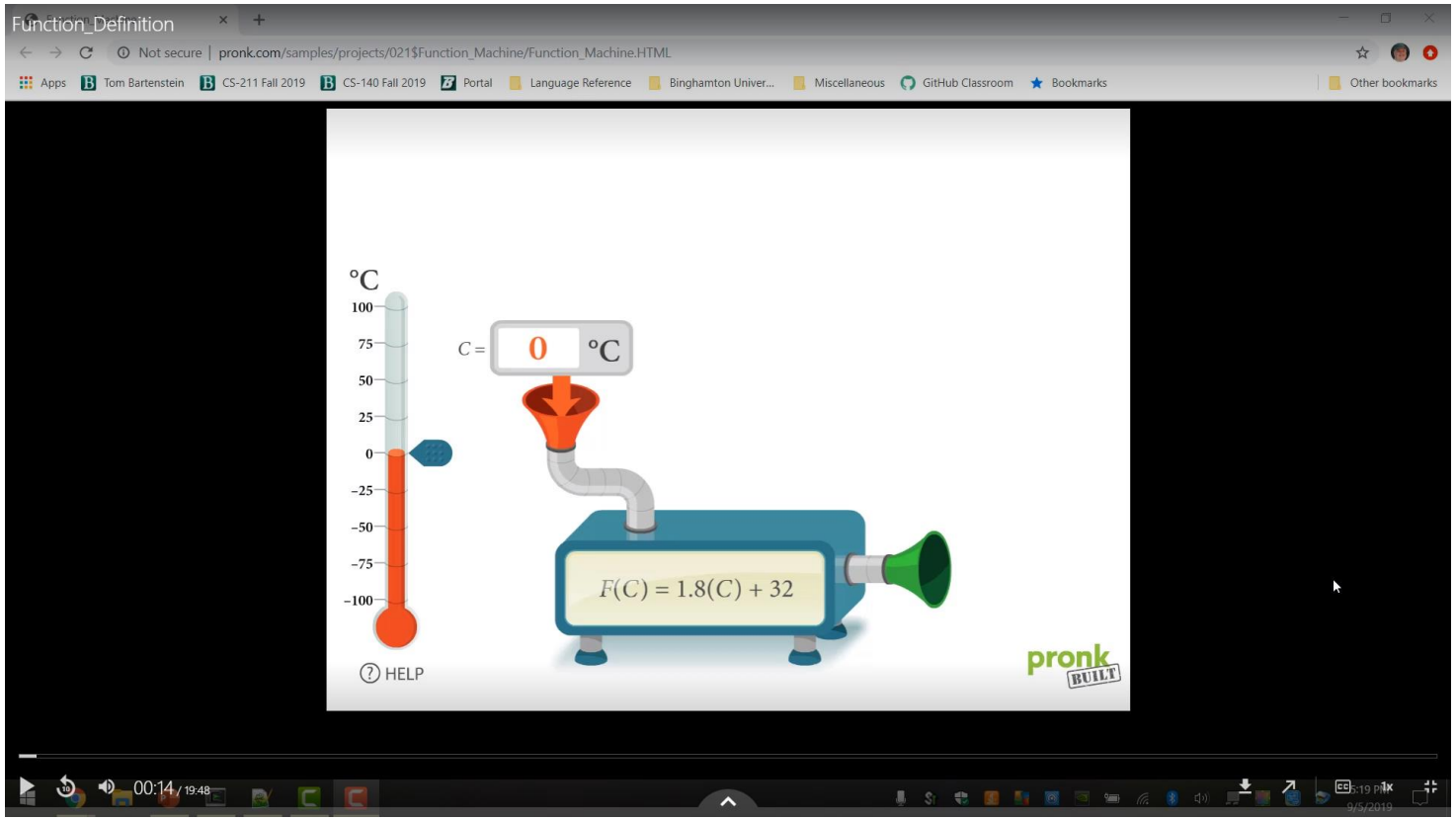
```
float cent_to_far(float t) {
        return (t*9.0)/5.0+32;
}
```

Functions for Abstraction

- Function Prototype: float cent_to_far(float t)
- Function Behavior: Convert Centigrade to Fahrenheit
- Function Embodiment: WHO CARES?
  - As long as it works, we can ignore the embodiment!
  - Code it once, test it, and then use it lots of times!

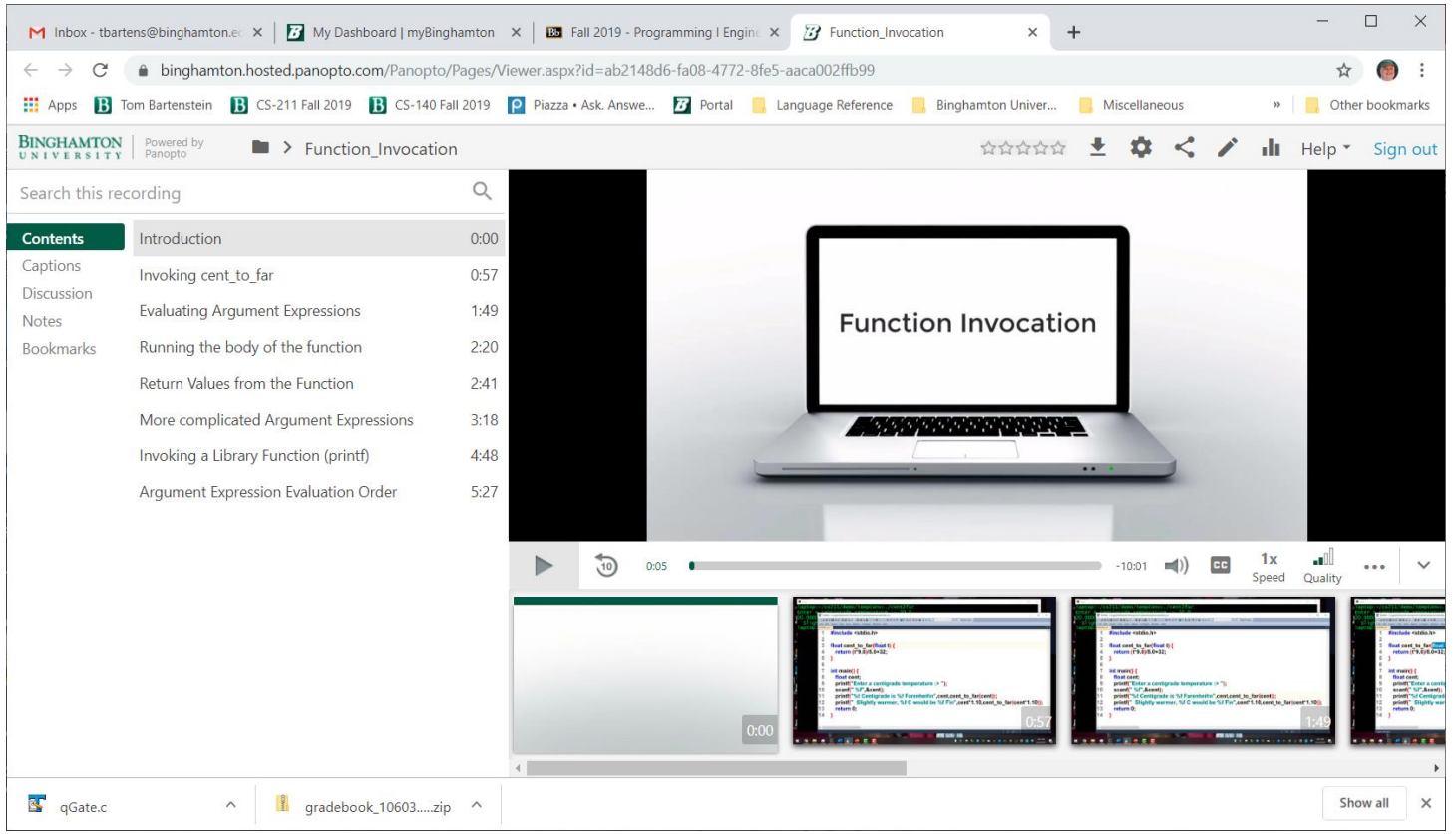
Functions: Function Definition

- Watch the video, available on myCourses
  - Content
    - Videos
      - 8. Functions



Functions: Function Invocation

- Watch the video, available on myCourses
  - Content
    - Videos
      - 8. Functions



Variable Scope

Scope

The places in your code that can read and/or write a variable.

- Scope starts at the location where you declare the variable
  - There may be holes in the scope!
- Scope ends at the end of the block in which the declare occurs
  - Usually, the function in which the variable was declared

Simple Example

```
#include <stdio.h>
int main(int argc, char **argv) {
        int j;
        for(j=0; j<argc; j++) {
                printf("Argument %d = %s\n",j,argv[j]);
        }
        return 0;
}
```

Block in which j is declared

Scope of j

“Block” is Usually a Function

- Scope of variable defined inside a function is “local” to that function
- But… you can define a variable inside a sub-block of a function

Internal Example

```
#include <stdio.h>
int main(int argc, char **argv) {
        int j;
        for(j=0; j<argc; j++) {
                int k=j+1;
                printf("Argument %d = %s\n",k,argv[j]);
        }
        return 0;
}
```

Block in which k is declared

Scope of k

Global Variable

- Declared outside of a block (including functions!)
- Scope is from declaration to the end of the C file!

Global Example

```
#include <stdio.h>
int nc=0;
int myfunc(int n) { nc++; return n; }
int main(int argc, char **argv) {
        int j;
        for(j=0; j<argc; j++) myFunc(2);
        printf("myfunc called %d times\n",nc);
        return 0;
}
```

Block in which nc is declared

Scope of nc

Local Variable Pros & Cons

Advantages

- Simple and intuitive
- When you code a function, you don't need to worry about what you're caller is doing
- Each invocation gets its own version of local variables

Disadvantages

- Not remembered from call to call
- Cannot communicate outside of function

Global Variable Pros & Cons

Advantages

- Simple and intuitive
- Enables functions to communicate data with each other
- Remembers between function calls as well as within function calls

Disadvantages

- Increases the “outside” information a function needs to be aware of (binding)
- Prevents re-use of functions
- Remembers between function calls as well as within function calls

Variable Class

- Automatic
  - Created/Initialized on entry to block
  - Destroyed on exit from block
- Static
  - Created/Initialized when program starts
  - Destroyed when program ends

Default Class

- Function/Block Variables are automatic
  - Created/Initialized on entry to that function/block
  - Deleted when that function/block ends
- Override class of Function/Block Variables by using the “static” keyword
  - Created/Initialized when the program starts
  - Deleted when the program ends
  - Does not change the scope – scope is still inside the function!
- Global Variables are Static
  - Created/Initialized when the program starts
  - Deleted when the program ends
  - “Automatic” and “static” behave exactly the same – the “block” is the program!

Example Local Static

```
char * flipflop() {
        static int flip=1;
        if (flip) { flip=0; return “flip”; }
        else { flip=1; return “flop”; }
}
for(i=0;i<8;i++) printf(“%s “,flipflop());
```

flip flop flip flop flip flop flip flop

BAD FORM : Pseudo-Globals

- It is legal in C to nest a function inside another function
- This allows the variables in the outside function to be visible (in scope) for the inside function
- C Coders frown on this practice!
  - Nested functions have other complications
  - Nested functions cannot be re-used
  - It's ugly and confusing

Example Nested Functions

```
int main(int argc, char **argv) {
        char firstArgLetter(int i) {
                return argv[i][0];
        }
        int j;
        for(j=0;j<argc;j++) printf("Arg start: %c\n",
                firstArgLetter(j));
        return 0;
}
```

The “main” function

- Every C program must have a “main” function
- When C program is run, the OS invokes the main function
- When the main function returns, program ends
- Return value “int”
  - Return value 0 indicates program worked OK
  - Return value other than 0 indicates program failed
- main function arguments – stay tuned
- main function may invoke lower level functions
- For now: int main() { … ; return 0; }

Arguments to main

- When the operating system calls the “main” function, it:
- parses the command line, splitting at “white space” (blanks, tabs)
  >./convertTemp 21.3 F C
- Counts the number of blank delimited words: argc=4
- Creates an array of “words” or strings (really an array of arrays)

argv[0]	“./convertTemp”
argv[1]	“21.3”
argv[2]	“F”
argv[3]	“C”

Size=4

Parameters of main

- Typically use argc and argv as the names of the parameters to main

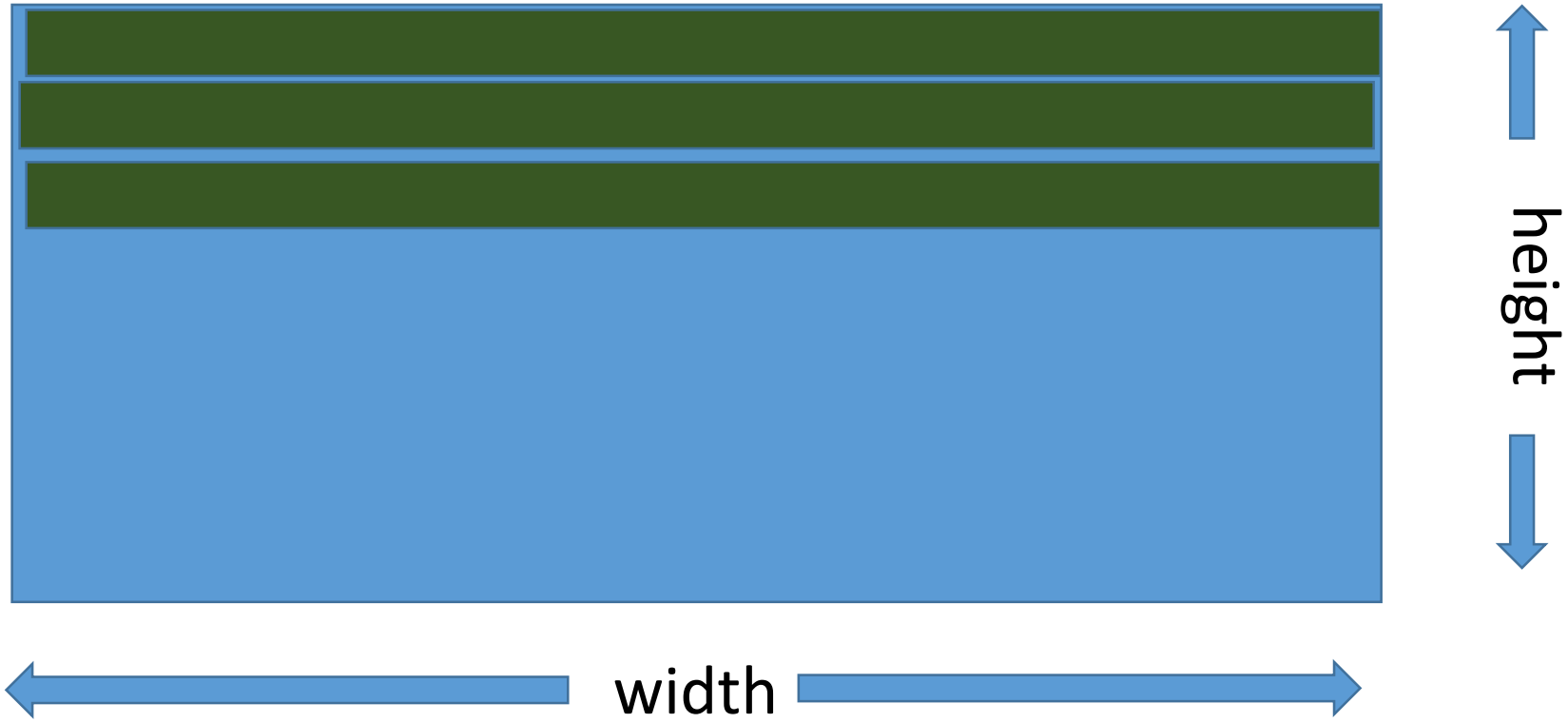
```
int main(int argc, char **argv) { …
```

- argc argument count – the size of the argv array
- argv argument value (or vector) - a list of strings
  - Each string is an array of characters
  - So argv is an array of arrays of characters
  - Almost (char[])[] argv – argv is an array of (array of characters)
    - (note: not char[][] argv – not a two dimensional array of characters!)
  - Can reference each string as argv[i]
  - Can reference individual letters as argv[i][j] really (argv[i])[j]

Demo

- We are working on CAD for house plans. Our customer has asked us to write a function, given the length and width of a wall in inches, figure out how many linear feet of board to the nearest foot will we need to cover the wall if the boards are 8 inches wide?

Big Picture



Need ceil(*height*/8) boards, each *width* wide.

Need ceil(*boards* * *width*/12) linear feet

Function Arguments?

- What values is the user willing to give us?
- What do we need to figure out the answer?
- What is the data type of each argument?
- What name will you use for each argument?
  - Easy to type and remember

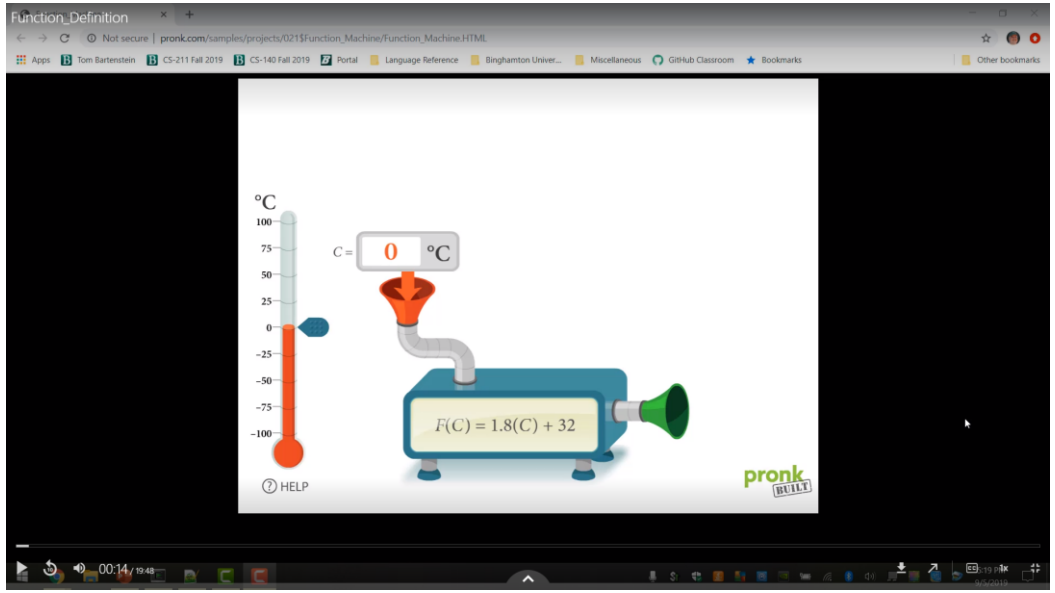
Note: When you implement your function, you may change your mind. That's OK.

Return Type?

- What type of data does your user expect you to return?
- Does the data type reflect the precision of the inputs?

Resources

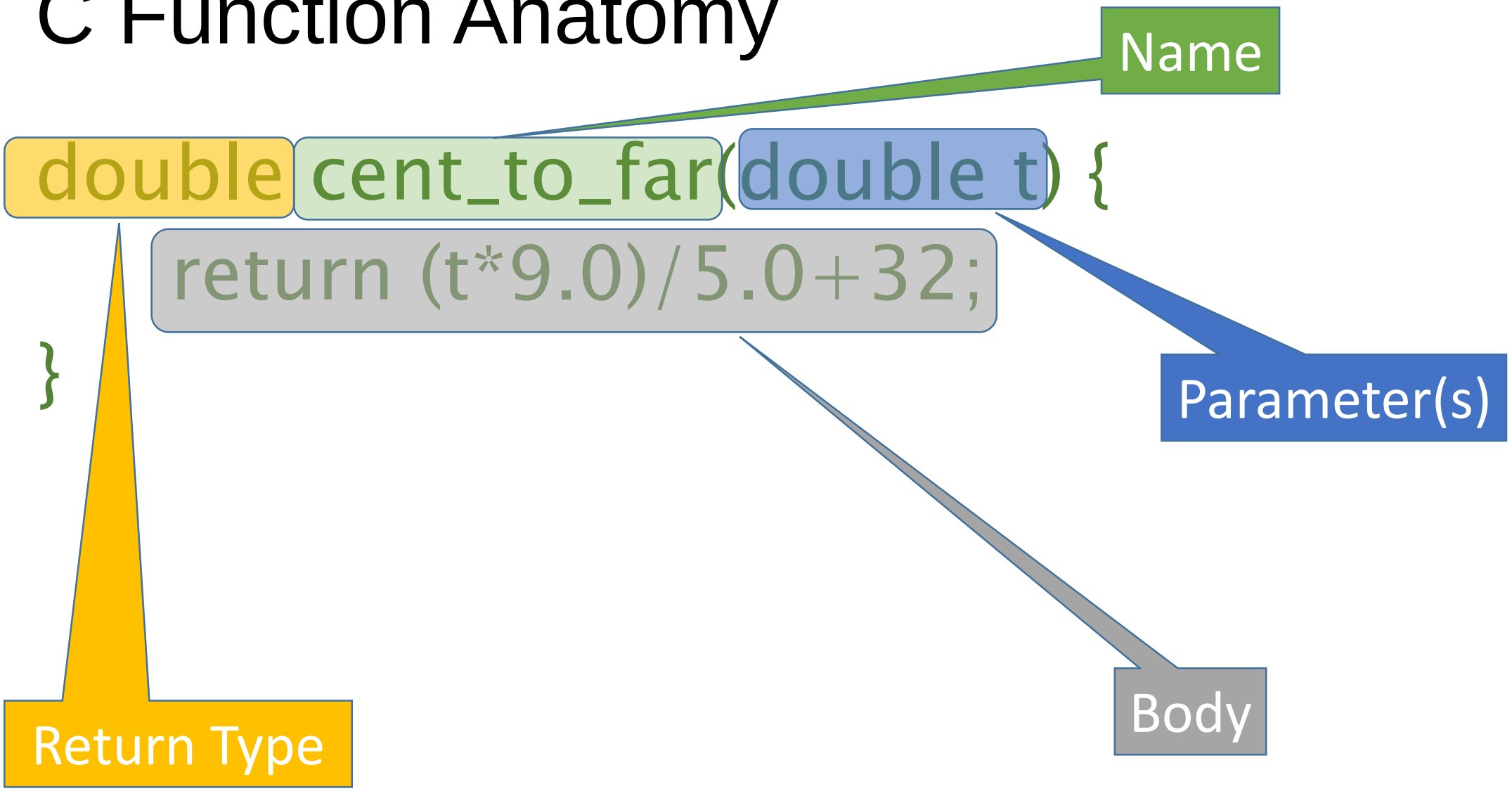
- Programming in C, Chapter 7 up to “Functions Calling Functions Calling …” (p. 130)
- YouTube: Meat-a-Morphis – Introduction to Functions (https://www.youtube.com/watch?v=VUTXsPFx-qQ)
- WikiPedia: Subroutine (https://en.wikipedia.org/wiki/Subroutine)
- WikiPedia: C Standard Library: (https://en.wikipedia.org/wiki/C_standard_library)
- C Library Reference Guide (https://en.cppreference.com/w/c/header)



Functions: Function Definition

Summary Notes

C Function Anatomy



Function Arguments or Parameters

- Almost like Local Variable Declarations with no initialization
- Comma separated list of “variables” passed in to the function
- Each entry specifies the **type** and **name** of one parameter
- When a function is invoked, the invoker specifies the initial **values** of the arguments
- The parameter can be referenced by its name in the function body
- Values passed are COPIES of the arguments
  - Changing a parameter does not change the caller’s variable!

Argument vs. Parameter

- Really the same thing… data passed into a function to work on
- The difference is the point of view…
  - Caller's point of view: Arguments

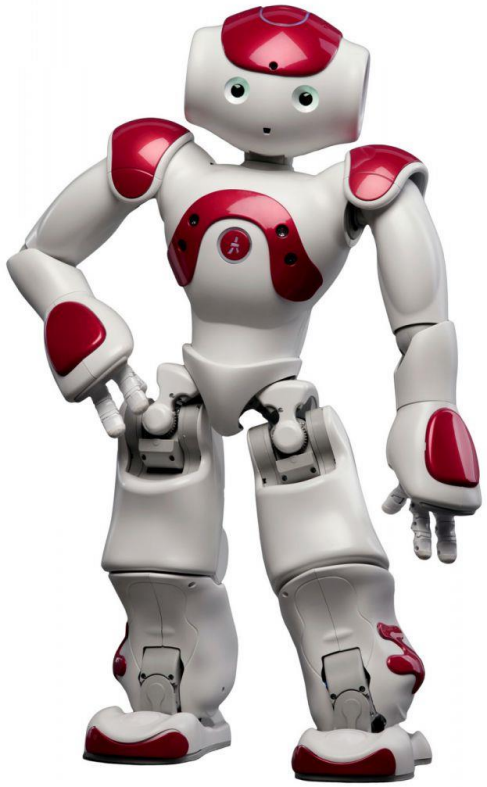
```
float ftemp=cent_to_far(28.3);
```

  - Called function ("callee") point of view: parameter

```
float cent_to_far(float t){
    return (t*9.0)/5.0+32;
```

Function Body

- List of C statements that define how the function works
- Typically works on arguments to produce result
- May have “side effects” (other than producing a result)
  - Write messages to the console
  - Read information from a file
  - Change a global variable value
- Result is specified by a “return” statement

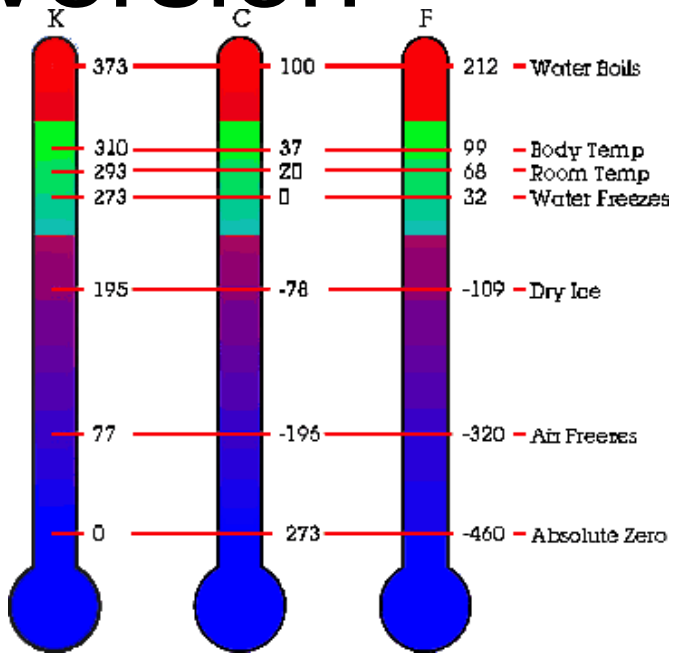


Function Return Value

- Type specified in function definition
  - Must specify some return type
  - May specify “void” to indicate function returns nothing
    - In this case, no return statement is required, or no value: return;
- Value specified in body by return statement

  return *value*;
- where *value* is an expression that evaluates to the return type

Generalizing Temperature Conversion



- What if we want to support conversion from C to F?
- What about C to K (Kelvin)
- Or K to F
- …

- Is there a general function that handles temperature conversion from any scale to any scale?

Facts about Temperatures

Scale	Freezing Point	Boiling Point
Farenheit	32	212
Centigrade	0	100
Kelvin	273	373
….	…	…

Interpolation: Find distance from freezing, convert distance to new scale, then add freezing point.

Conversion Formulas

$$delta_{scale} = boiling(scale) - freezing(scale)$$

$$fromFreezing_f = t_f - freezing(f)$$

$$fromFreezing_t = fromFreezing_f \times \frac{delta_t}{delta_f}$$

$$t_t = fromFreezing_t + freezing(t)$$

C Declare before Use Rule

- In C, you cannot invoke a function which has not been “declared”
- One way to declare a function is to fully define the function
- This causes “upside down” code
  - Lowest Level functions are first in the file
  - Highest Level function (main) is last in the file

Example of Upside Down Code

```
float freezingPoint(char scale) {…}
float boilingPoint(char scale) { … }
int validScale(char scale) { … }
float convertTemp(…) {
…validScale('C'); boilingPoint('C');
   freezingPoint('C');…  }
int main(int argc, char **argv) {
… convertTemp('C2F',17.3) …
}
```

Function Prototype Declare

- Function Prototype: float freezingPoint(char scale)
- Prototype consists of return type, function name, & parameter list
- We can declare a function by specifying the prototype;
  - Followed by a semi-colon
- Still need full function definition somewhere else
- Enables “Right Side Up” coding
  - Function Prototypes at the top of the file
  - Function definition for top level function (main) next
  - Then function definitions for lower level functions

Example of RightSide Up Code

Declarations

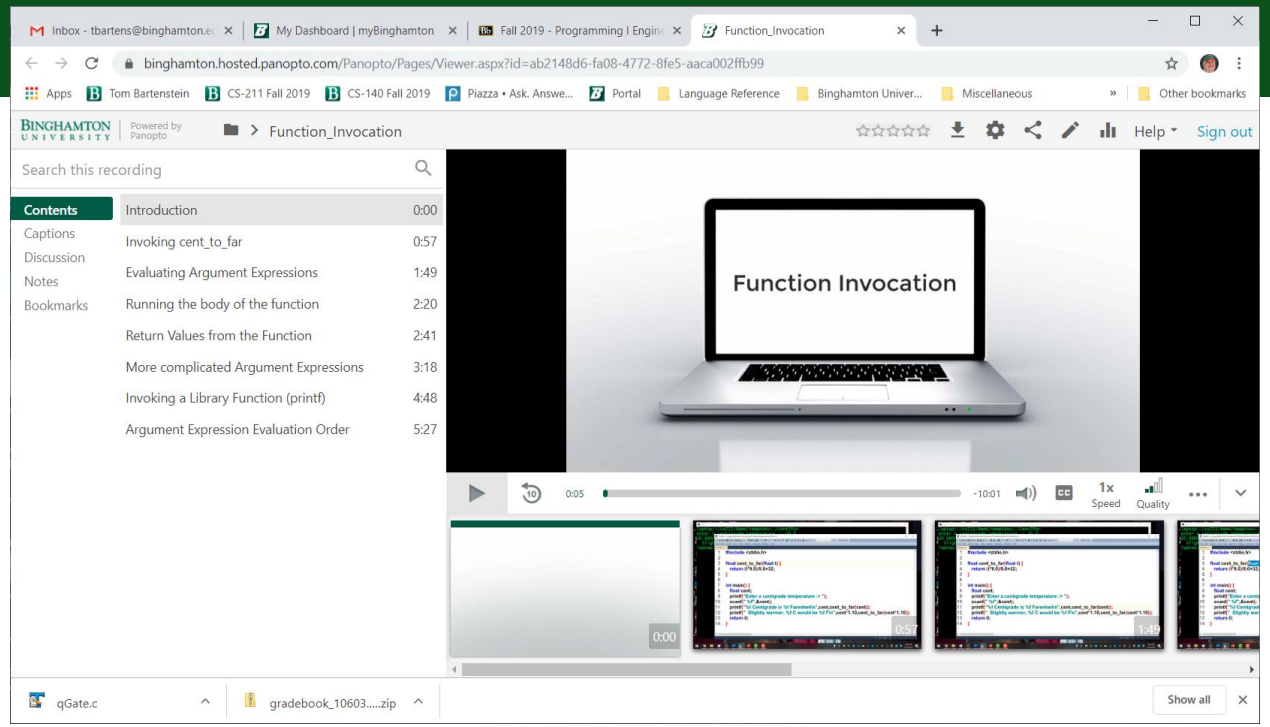
```
float convertTemp(char scale,…);
int validScale(char scale);
float boilingPoint(char scale);
float freezingPoint(char scale);
```

Definitions

```
int main(int argc, char **argv) { … }
float convertTemp(char scale,…) { … }
int validScale(char scale) { … }
float boilingPoint(char scale) { … }
float freezingPoint(char scale) {…}
```

Writing Functions in C

1. Get the big picture… what are we trying to do?
2. Choose a function name
3. Identify the argument list
4. Figure out the return type
5. Write the function prototype
6. Implement the function
7. Test the function



Functions: Function Invocation

Summary Notes

Function Invocation

- Invoke a function in a C expression by specifying:

function_name (*argument_expression_list*)

- *function_name* : name of a previously declared function
- *argument_expression_list* : comma separated list of expressions to be used as arguments
- When a function is invoked, C will….
    1. evaluate argument expressions,
    2. Copy argument values into parameters
    3. invoke the specified function (run it's body)
    4. conceptually replace the function invocation with the returned value

```
float warmer=cent_to_far(ctemp*1.10);
```