

### Announcements (0:00-5:00)

- OHs, VOHs, sections, supersections now listed on website
- Supersection will review the week's material with some different examples
- Website tells you when next office hours are and blinks if in progress

### Functions (5:00-9:00)

- Function - a miniature program that can be summoned to perform a particular task
- Examples: printf(), GetString, GetInt()
- In beer4.c, we call upon the function chorus() when we want to print the chorus for a particular number of bottles
- Functions can take inputs and return outputs
- Example: printf() takes the string you want to print and then optionally 0 or more arguments that are numbers, strings, etc.
- For now, you can write your whole program in main, but eventually it will become too long and unwieldy
- Functions allow you to break your program up into smaller pieces
- Functions also allow you to do the same task multiple times without repeating the code. They make code reusable.
- If you ever find yourself copying and pasting code, you should probably just put the repeated code in a function and call it multiple times.

### Local Variables (9:00-15:00)

- Suppose we want to write a function that switches the values in two variables x and y.
- Take a look at buggy3.c:

```
int
main(int argc, char * argv[])
{
    int x = 1;
    int y = 2;

    printf("x is %d\n", x);
    printf("y is %d\n", y);
    printf("Swapping...\n");
    swap(x, y);
    printf("Swapped!\n");
    printf("x is %d\n", x);
    printf("y is %d\n", y);
}

void
swap(int a, int b)
{
    int tmp;
```

```
    tmp = a;  
    a = b;  
    b = tmp;  
}
```

- This is supposed to print the values of x and then y (1 and then 2), then switch them and print x and y again (2 and then 1)
- But this doesn't work. Why not?
- When swap is executed, it gets passed copies of x and y. That is, it gets the values of x and y, but not the actual memory locations.
- Then swap does its assigned task properly, but only for those copies. As soon as the function is over, those copies disappear and x and y remain unchanged.
- This is called passing by value. Later, we will see how to pass by reference (pointer).
- For now, just know that inside the function is a different scope. These variables are separate from those in main.
- But what if we change the arguments of the function swap to (int x and int y) and then change all the a's and b's in swap() to x's and y's? Does this fix our problem?
- Nope. Even though these variables happen to be named x and y, they still exist only within the function. They are local variables.
- Remember, scope is determined by where variables are declared, not what they are named.
- Later we will learn how to fix this problem properly.

### Global Variables (15:00-28:30)

- Now let's say we want to write a function to increment a variable x.
- Take a look at buggy4.c

```
/* function prototype */  
void increment();  
  
int  
main(int argc, char * argv[])  
{  
    int x = 1;  
    printf("x is now %d\n", x);  
    printf("Incrementing...\n");  
    increment();  
    printf("Incremented!\n");  
    printf("x is now %d\n", x);  
}  
  
void  
increment()  
{  
    x++;  
}
```

- We declare `x` in `main` and then call `increment` to add 1.
- This won't even compile. Why? `gcc` tells us "x undeclared."
- But we declared `x` in `main`. Why doesn't `gcc` understand?
- Because `x` exists only in `main`. It does not exist in `increment()`.
- We can solve this problem using global variables.
- Take a look at `global.c`

```
/* global variable */
int x;

/* function prototype */
void increment();

int
main(int argc, char * argv[])
{
    printf("x is now %d\n", x);
    printf("Initializing...\n");
    x = 1;
    printf("Initialized!\n");
    printf("x is now %d\n", x);
    printf("Incrementing...\n");
    increment();
    printf("Incremented!\n");
    printf("x is now %d\n", x);
}

void
increment()
{
    x++;
}
```

- This time, we declare `int x` at the *top* of the program, outside of all curly braces. This means that it is not scoped. It is accessible to any function written in this file (and even in other files if we use the right syntax).
- Notice also that we put the function prototype `void increment()` at the top of the program
- This is like declaring a variable so the compiler knows what to expect. In C, you must either define functions before `main` or declare them before their first calls.
- **BUT** global variables are generally considered bad design. In most cases, you can avoid using a global variable by passing the variable to a function and then having it return the variable after changing it.
- For instance we could have passed `x` to `increment()`, had `increment` return the incremented `x`, and reassigned the return value to `x` in `main`.
- Now take a look at `buggy5.c`

```
/* global variable */
int x;

/* function prototype */
void increment();

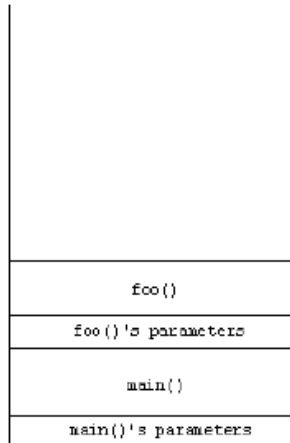
int
main(int argc, char * argv[])
{
    printf("x is now %d\n", x);
    printf("Initializing...\n");
    x = 1;
    printf("Initialized!\n");
    printf("x is now %d\n", x);
    printf("Incrementing...\n");
    increment();
    printf("Incremented!\n");
    printf("x is now %d\n", x);
}

void
increment()
{
    int x = 10;
    x++;
}
```

- This program is supposed to print x, assign 1 to x, print x, increment x, and print x again
- But what it prints is 0, 1, 1. What's going on?
- Because we've declared another variable x within the scope of increment(), this x is what gets incremented.
- We say that the new x shadows the global variable x, just like an identically named parameter would (as in a previous example).

### The Stack (28:30-33:30)

- When you run a.out, your operating system loads all the 0's and 1's from the executable into RAM.
- If we view it as a rectangle, this all goes in the top.
- Meanwhile, it uses the bottom of the rectangle for the stack.
- Any time you call a function, that function gets its own frame (chunk of memory) in the stack
- So first main gets a chunk of memory for its parameter and then its local variables.
- Then when we call foo(), foo() gets a chunk of memory for its parameters and local variables.
- As more things get called within foo(), they get "stacked" on top of the chunk reserved for foo().



- This opens up the possibility for security problems.
- By the way, foo, bar, baz, qux, etc. are just variables

### Return Values (33:30-36:30)

- A return value is the output given by a function.
- You must specify what the return value of a function is when you declare and define it.
- If a function does not have a return value, you precede it by the word void when you declare and define it.
- Example: This function takes an int and returns an int

```
int cubed(int a) {  
    return a*a*a;  
}
```

### Arrays (36:30-50:00)

- An array is a simple data structure that allow you to allocate space for multiple variables at once
- Suppose we want to make space for a student's grade in a course. We could just declare int x
- But what if we want to make space for all 32 of his courses. We could declare int x1, ..., x32, but that is awfully tedious
- Instead, we just declare int grades[32] which sets aside 32 pieces of memory, each the size of an int
- We can index into this array by saying grades[0], grades[1], etc.
- Since we start at 0 and there are 32 slots, we can only go up to grades[31]
- Take a look at array.c

```
/* number of quizzes per term */  
#define QUIZZES 3  
  
int
```

```
main(int argc, char * argv[])
{
    float grades[QUIZZES];
    int average, i, sum;

    /* ask user for grades */
    printf("\nWhat were your quiz scores?\n\n");
    for (i = 0; i < QUIZZES; i++)
    {
        printf("Quiz #%d of %d: ", i+1, QUIZZES);
        grades[i] = GetFloat();
    }

    /* compute average */
    sum = 0;
    for (i = 0; i < QUIZZES; i++)
        sum += grades[i];
    average = (int) (sum / (float) QUIZZES + 0.5);

    /* report average */
    printf("\nWithout dropping your lowest score, " \
           "your average is: %d\n\n", average);
}
```

- Notice how we define QUIZZES to be 3 at the top of the program. This is how we declare constant values that are not going to change later in the program.
- We do this to make code more readable and in case we want to change the value of QUIZZES later.
- In this program, we use a loop to populate the array as we ask the user for input.
- We know that the array will have precisely QUIZZES slots and so we only index up to QUIZZES-1
- Then we take the sum over the array and calculate the average
- Notice that we cast QUIZZES to a float first in order to ensure floating point division
- We implement rounding by adding 0.5 and then casting to an int. (Why can't we just cast to an int?)
- Now look at buggy6.c

```
#include <cs50.h>
#include <stdio.h>

/* number of quizzes per term */
#define QUIZZES 3

int
main(int argc, char * argv[])
{
    float grades[QUIZZES];
    int i;

    /* ask user for grades */
```

```
printf("\nWhat were your quiz scores?\n\n");  
for (i = 0; i < QUIZZES; i++)  
{  
    printf("Quiz #%d of %d: ", i+1, QUIZZES);  
    grades[i] = GetFloat();  
}  
  
/* print scores */  
for (i = 0; i < 10; i++)  
    printf("%.2f\n", grades[i]);  
}
```

- This is supposed to take a list of grades and print them back out.
- But in addition to the three grades we entered, we get 7 more bogus numbers. Why?
- Because instead of iterating up to QUIZZES, we iterated up to 10.
- When we hard code numbers into our program rather than defining them as variables or constants, we call them magic numbers because, from the reader's point of view, there is no rational explanation for their choosing.
- In this situation, it also got us into a bit of trouble since QUIZZES was only 6.
- Indexing memory locations that have not been declared or assigned previously in the program is very bad and opens the door for security problems.