Week 1 Monday — Computer Science 50
September 22, 2008 — Fall 2008
Scribe Notes — Anjuli Kannan

Announcements (0:00-13:00)

- More survey results
    - Comfort levels: about 3/8 among those less comfortable, 1/8 among those more comfortable, and the remainder in between
    - 250 of you have not taken AP Computer Science
    - Over 90% have a laptop
    - Most have “normal” cell phones, a few iPhones, Blackberrys
- Sections
    - You should now know your section. Let Cansu know if you have been assigned a section you can’t make. Section calendar on website.
- Office Hours
    - Now on website.
    - You’ll notice there are also virtual office hours. You can attend them by entering the “Virtual Terminal Room” on the website, which is essentially a “glorified chat room.”
    - It has some additional features, such as desktop sharing. Make sure you read the instructions on the website before logging in.
    - The terminal room now features the “Express Lane”, which is intended for extremely quick questions (probably of the yes/no variety) and will be prioritized relative to the “Regular Lane”.
- Problem Sets
    - If you did the hacker edition of Problem Set 0, please turn in your scratch board at Wednesday lecture. May also be returned at next Monday’s lecture.
    - This week’s problem set introduces you to input validation, greedy algorithms
    - From now on, we will hold problem set walkthroughs on Mondays. These should orient you on the weeks’ problem set, giving you a sense of what you need to do and how to approach the tricky parts. These videos (and *not* office hours) should be the first place you go if you don’t know where to begin on a particular problem set.

Introduction to Cryptography (13:00-15:00)

- Here is an encrypted message: Or fher gb qevax Ibhe binygvar!
- It has been encoded using a cipher. A cipher is an algorithm that takes some input text (plaing text) and produces output text (cipher text). Its purpose is to hide the message.
- This message has been encoded by rotating every letter 13 places (A→N, B→O, etc.)
- What does it say? Be sure to drink your ovaltine.

- This is just like using one of those decoder rings that children use to send secret messages to their friends. (In other words, it's not a very good way to conceal data.)

From C to Other Languages (15:00-17:00)

- Many of the things we learn in C are concepts that can be carried over to other languages.
- For instance, if we wanted to write good old "hello, world" in C++, it would look like this:

```
#include <iostream>

using  namespace std;

int
main(int argc, char * argv[])
{
    cout << "hello, world!\n" << endl;
}
```

- Check your sourcecode handout for the same program written in Java, Perl, and other languages
- After a course like this, you'll be able to do a lot of programming, but you'll find that often other languages are required for a particular task.
- Hopefully this course will give you the skills you need to teach yourself new languages.

Bugs (17:00-19:30)

- According to legend, Grace Hopper discovered the first "bug" in a program when she pulled a dead moth out of a computer.
- Today, we use the term "bug" to refer to any sort of problem that causes our programs to run in a way other than intended
- Take a look at buggy1.c. This program is supposed to print 10 asterisks. What is the problem?

```
int
main(int argc, char * argv[])
{
    int i;

    for (i = 0; i <= 10; i++)
        printf("*");
}
```

- It prints 11 asterisks because i runs from 0 to 10, which is 11 integers. Conventionally, we would change this to for(i=0; i<10; i++)
- Take a look at buggy2.c. This program is supposed to print 10 asterisks one per line.

```
int
main(int argc, char * argv[])
{
    int i;

    for (i = 0; i <= 10; i++)
        printf("*");
        printf("\n");
}
```

- Instead, we get all the asterisks on the same line. Why?
- We are missing curly braces. If you want to associate multiple statements with a for loop, you have to put them inside curly braces. Only if there is a single line can you omit them.

Integer and Character Casting (19:30-30:30)

- Since all the computer can store are 1's and 0's, how are we going to represent characters? With numbers.
- ASCII is a mapping between integers and characters on the keyboard. You can find the ASCII encodings of letters at asciitable.com
- What if we want to know which integer is associated with a particular integer (without looking up the ASCII table)?
- We can use a cast.
- Last week, we used a cast to make an int into a float to force floating point division.
- Again, we will use casting to change a type, but this time between the types int and char.
- So we can say
  int i = (int) 'A';
  char c = (char) 65;
- Let's take a look at ascii1.c

```
int
main(int argc, char * argv[])
{
    int i;

    /* display mapping for uppercase letters */
    for (i = 65; i < 65 + 26; i++)
        printf("%c: %d\n", (char) i, i);

    /* separate uppercase from lowercase */
    printf("\n");
    /* display mapping for lowercase letters */
    for (i = 97; i < 97 + 26; i++)
        printf("%c: %d\n", (char) i, i);
}
```

- In this program, we iterate from 65 up to 91 and for each number , we tell the computer that we want to know the associated character, (char) i. We also have it print out the integer itself, i.
- The result is a list of the capital letters with their associated numbers (A:65, B: 66, …, Z:90).
- Then we do the same from 97 to 122, which is the lowercase letters.
- What can we do to make this a little cleaner? Let’s take a look at ascii2.c.

```
int
main(int argc, char * argv[])
{
    int i;

    /* display mapping for uppercase letters */
    for (i = 65; i < 65 + 26; i++)
        printf("%c  %d    %3d  %c\n", (char) i, i, i + 32,
               (char) (i + 32));
}
```

- This time, we make use of the fact that the integer corresponding to any given lowercase letter is exactly 32 more than the integer corresponding to the same letter in uppercase. Therefore, we needn’t cycle through 97-122 separately. When we want the uppercase letter, we just add 32 to the letter representing the uppercase letter.
- We also make use of the width aspect of format strings. We know that some numbers will be 2 digits and some 3. To make things line up perfectly, we tell each number to take up three places.
- In ascii3.c, we iterate over the letters themselves.

```
int
main(int argc, char * argv[])
{
    char c;

    /* display mapping for uppercase letters */
    for (c = 'A'; c <= 'Z'; c = (char) ((int) c + 1))
        printf("%c: %d\n", c, (int) c);
}
```

- We initialize char c to A. In the update, we cast c to an int, add 1 to get to the next letter in the alphabet, and then cast it back to a char. We repeat until c is Z.
- Actually, we don’t always need to be this explicit with casting because the compiler can sometimes figure it out without us telling it to (implicitly). For

instance, if we simply added 1 to char c, it would know that we meant to add 1 to the int value and put the result back in c.

- Suppose we wanted to implement the game Battleship. The following code, battleship.c, prints out the gameboard:

```
int
main(int argc, char * argv[])
{
    int i, j;

    /* print top row of numbers */
    printf("\n   ");
    for (i = 1; i <= 10; i++)
        printf("%d  ", i);
    printf("\n");

    /* print rows of holes, with letters in leftmost column */
    for (i = 0; i < 10; i++)
    {
        printf("%c  ", 'A' + i);
        for (j = 1; j <= 10; j++)
            printf("o  ");
        printf("\n");
    }
    printf("\n");
}
```

- The first for loop prints the numbers from 1 to 10 across the top of the screen
- The second for loop prints 10 separate lines (we know because it ends with printf("\n"). On each line, we print a char that is i letters past A (where i goes from 0 to 9) followed by 10 o's.

Functions (30:30-50:45)

- Think about the song "99 Botles of Beer on the Wall."
- If we wanted to print out the lyrics to this song, we would make use of a loop.
- Examine the following code, beer1.c:

```
int
main(int argc, char * argv[])
{
    int i, n;

    /* ask user for number */
    printf("How many bottles will there be? ");
    n = GetInt();

    /* exit upon invalid input */
    if (n < 1)
    {
        printf("Sorry, that makes no sense.\n");
        return 1;
    }

    /* sing the annoying song */
    printf("\n");
    for (i = n; i > 0; i--)
    {
        printf("%d bottle(s) of beer on the wall,\n", i);
        printf("%d bottle(s) of beer,\n", i);
        printf("Take one down, pass it around,\n");
        printf("%d bottle(s) of beer on the wall.\n\n", i - 1);
    }

    /* exit when song is over */
    printf("Wow, that's annoying.\n");
    return 0;
}
```

- First, we get input from the user and, if given a negative number, print an error message and return 1.
- Recall that main has a return value of int. If all goes well, it returns 0. If it exits with a problem it returns 1. 1 is an error code that signals to someone running the program that the program exited abnormally.
- Then we loop from n down to 0, decrementing i by one every time.
- We can improve this program using hierarchical decomposition. This just means breaking down a large problem into smaller problems.
- For example, one major piece of this program is the part that prints out the chorus for a given value of n. We can factor this out and put it into its own function. When we want to print the chorus, we simply call the function.
- Why is hierarchical decomposition good?
    - If you're writing programs as part of a team, breaking the program into functional components allows you to effectively divide up the work, and define how everything will fit together.
    - You can reuse your own code without copying and pasting.
    - Readability.
- Look at beer4.c:

```
int
main(int argc, char * argv[])
{
    int n;

    /* ask user for number */
    printf("How many bottles will there be? ");
    n = GetInt();

    /* exit upon invalid input */
    if (n < 1)
    {
        printf("Sorry, that makes no sense.\n");
        return 1;
    }

    /* sing the annoying song */
    printf("\n");
    while (n)
        chorus(n--);

    /* exit when song is over */
    printf("Wow, that's annoying.\n");
    return 0;
}
```

- Now, we just call the function chorus() repeatedly as long as n is nonzero.
- In the line while (n) we make use of the fact that, when considered as a Boolean, 0 is false, and all other numbers are true.   This statement will be false only when 0 is reached.
- Notice also that we decrement n within the call to chorus itself. This tells the computer to perform chorus with the current value of n, then reduce it by one.
- This is the same as:

```
while(n)
{
   chorus(n);
   n = n-1;
}
```

- Note that n-- and --n are both valid C syntax. Home exercise: see what happens if you pass --n to chorus().
- Now look at the function void chorus (int b)

```
void
chorus(int b)
{
    string s1, s2;

    /* use proper grammar */
    s1 = (b == 1) ? "bottle" : "bottles";
    s2 = (b == 2) ? "bottle" : "bottles";

    /* sing verses */
    printf("%d %s of beer on the wall,\n", b, s1);
    printf("%d %s of beer,\n", b, s1);
    printf("Take one down, pass it around,\n");
    printf("%d %s of beer on the wall.\n\n", b - 1, s2);
}
```

- This is a void function because it does not return anything. That is, it produces no output that can be assigned to a variable. It only has “side effects” (printing text to the screen)

- Within the function, we now refer to the number of bottles as b. We don't call it n because we might call this function in many different contexts with many different variables.
- The function simply makes a local copy of whatever number it is passed and calls it b. Throughout the function, it refers only to b when it wants to deal with the number of bottles.

Local Variables (50:45-57:00)

- 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)
- First, note that we don't assign the result of swap to anything. This is because swap is a void function, like printf. It does (or should do) some work for us, but doesn't return a value.
- 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. 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.

Scope and the Stack (57:00-62:45)

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

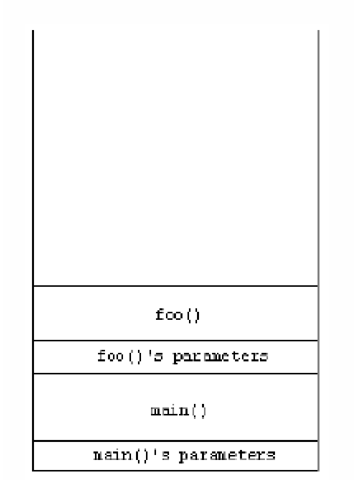
```
/* function prototype */
void increment();

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().
- This is a problem with the *scope* of x.
- Imagine the chunk of RAM allotted to a program as a rectangle, called the stack.
- Each time a function is called, it gets allotted a piece of this rectangle, called a stack frame. Inside the stack frame go all of its local variables.
- When the function is done, its stack frame gets popped off, so its local variables are no longer accessible. They go "out of scope."

- (This is why the local copies of x and y in swap(), which got properly swapped, seemed to disappear after the function was completed. Once swap() was popped off the stack, its variables went out of scop.)
- As the diagram shows, when we first bring up the program, main gets a chunk of memory for its parameter and then its local variables.
- Then when we call foo(), foo() get s 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().



- But each function only has access to what's in its chunk of memory.
- As you can see from the diagram, a variable local to main (such as x in our example) is not accessible from functions called by main.
- This opens up the possibility for security exploits. By guessing at and accessing various memory addresses, a malicious attacker could find the right place to insert his own code such that it will be executed when another function is popped off the stack.

Global Variables (62:45-70:00)

- 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).
- 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.
- 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.
- Sometimes it is useful to put all of your function prototypes, fully commented in a separate .h file. This is what we do in cs50.h, available on cs50.net.
- 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. An identically named parameter would also do this.

Return Values (70:00-71:30)

- In return1.c, we see an example of a function that returns a value
- Since increment() now returns the result of incrementing, we can now see and store the result of that computation
- The value that is added to and returned is still a local copy, but since it gets returned to main, we have the opportunity to store it, and it is not lost as it was before.

The Ternary Operator (71:30-74:00)

- Getting back to something we glossed over before, look at chorus() in beer4.c
- In this function we have fixed the bottle/bottles grammar issue in this line:

```
s1 = (b == 1) ? "bottle" : "bottles";
```

- This line is the combination of an if statement and a variable assignment. It takes the if statement "If b=1, bottle. Else, bottles" and assigns the result to s1.
- ? is called the ternary operator
- Notice that we need two strings, s1 and s2, because the first three lines of the chorus refer to b bottles while the last one refers to b-1 bottles.

Week 1 Monday — Computer Science 50
September 22, 2008 — Fall 2008
Scribe Notes — Anjuli Kannan

Floats and Imprecision (74:00-77:11)

- Floats are inherently imprecise.
- We find that when we give the computer the value .41, it stores something very close to, but not exactly equal to, .41
- To make sure you get the right values in Problem Set 1, you can just work in cents rather than dollars. Then you’ll be working with integers.