Computer Science 50
Fall 2011
Scribe Notes

Week 2 Monday: September 12, 2011
Andrew Sellergren

Contents

Computer Science 50
Fall 2011
Scribe Notes

Week 2 Monday: September 12, 2011
Andrew Sellergren

1 Announcements and Demos (0:00–1:00, 50:00–60:00)

- If you've already dived into Problem Set 1, you've probably encountered one or more bugs. The very first bug was, in fact, a real bug—a moth that got stuck in the Harvard Mark II and was subsequently preserved inside the log book. This is where the term "bug" actually comes from.
- Don't forget to RSVP if you're interested in CS50 Lunch!
- Please return your Scratch board if you still have one from Problem Set 0.
- Sectioning needs to be done by 5 p.m. today!
- Supersections will be held today and tomorrow at 6 p.m., locations to be announced on the course website. The video from last night's supersection is available online.
- Some interesting statistics:
  - 33% of you own an iPhone, 17% of you own an Android device, 13% of you own a Blackberry, and 33% own a "normal" or non-smartphone.
  - 39% of you have AT&T, 36% of you have Verizon, 15% of you have T-Mobile.
  - Sophomores have the highest representation as a class among you, as does Adams as a house.
  - 56% of you are running Mac OS, 41% of you are running Windows.
  - Interestingly, this year for the first year, those "less comfortable" are the majority. You're definitely not alone!
  - **54% of you have no prior programming experience.**
- Check out the manual for the CS50 Appliance before you ask your question at Office Hours or on help.cs50.net. If you're using a PC and you find the Appliance so slow as to be unusable, follow the instructions here to see if it's because your computer's manufacturer has turned off hardware virtualization by default.
- Our aim is to give you as much qualitative feedback as possible on your work in CS50. Grades will be broken down as follows:
  - scope – how much of the problem set did you actually attempt?
  - correctness – does your code do what it's supposed to?
  - design – is your code efficient and elegant?
  - style – is your code readable and well-commented?

  You'll be given a score of 1 (poor) through 5 (best) for each of these axes. Early in the semester, the expected range is 2-3, not so much 4-5. 3 is actually good, it does not translate to 60%!

- A word on academic honesty, which we take very seriously in this course. Over the past four years, we have sent 37 students to the Ad Board. We'd like to never have to send another one. For your convenience, our policy is spelled out very clearly on the second page of **every** problem set we release. It describes in detail the line between collaboration and plagiarism, which essentially boils down to one guideline: don't talk in real code. If you want to discuss ideas or specific problems, feel free to go so far as to write out pseudocode, but don't go any farther. What you write in C, PHP, SQL, JavaScript, etc., should ultimately be your own. If you are ever in doubt, feel free to contact us with your specific situation.

2 From Last Time (1:00–3:00)

- We discussed Boolean expressions, which might look like the following:

```
if (condition || condition)
{
    // do this
}

if (condition && condition)
{
    // do this
}
```

  The `II` is the "or" operator and the `&&` is the "and" operator.

- Conditions allow us to handle more than one fork in the road:

```
if (condition)
{
    // do this
}
else if (condition)
{
    // do that
}
else
{
    // do this other thing
}
```

  Alternatively, we could use switch statements to achieve the same, but perhaps more elegantly:

```
switch (expression)
```

```
{
    case i:
        // do this
        break;

   case j:
       // do that
       break;

   default:
       // do this other thing
}
```

- When we wanted to complete the same task over and over again, we made use of loops:

```
for (initializations; condition; updates)
{
    // do this again and again
}

while (condition)
{
    // do this again and again
}
```

Functionally, for and while loops are identical, but in certain cases one might be more elegant than the other. There's even one more kind of loop that might come in handy with Problem Set 1:

```
do
{
     // do this again and again
}
while (condition);
```

The do-while construct ensures that the code will always execute at least once. It is particularly useful in prompting the user for input.

3 Bugs (3:00–15:00)

3.1 `buggy1.c`

- Can you figure out why the program below doesn't print 10 asterisks as it's supposed to?

```
/**************************************************************************
 * buggy1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Should print 10 asterisks but doesn't!
 * Can you find the bug?
 **************************************************************************/

#include <stdio.h>

int
main(void)
{
    for (int i = 0; i <= 10; i++)
        printf("*");
}
```

Well, it prints 11 asterisks instead of 10 because the termination condition is `i <= 10` rather than `i < 10`.

3.2 buggy2.c

- How about `buggy2.c`, which is also supposed to print 10 asterisks, one per line, but doesn't?

```
/**************************************************************************
 * buggy2.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Should print 10 asterisks, one per line, but doesn't!
 * Can you find the bug?
 **************************************************************************/

#include <stdio.h>

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

The second `printf` statement is not executed within the scope of the loop because no curly braces are placed around it and the first `printf` statement! Once we add the curly braces and recompile, we'll get one asterisk per line as we intended. We could also print out the asterisk and the newline character in a single call to `printf` to make this cleaner.

3.3 `math1.c`

- The following program, though syntactically correct, will not compile:

```
/**************************************************************************
 * math1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Computes a total but does nothing with it.
 *
 * Demonstrates use of variables.
 **************************************************************************/

#include <stdio.h>

int
main(void)
{
    int x = 1;
    int y = 2;
    int z = x + y;
}
```

When we try to compile this program, we get the following error:

```
math1.c: In function ‘main’:
math1.c:19:9: error: unused variable ‘z’ [-Werror=unused-variable]
cc1: all warnings being treated as errors
```

Although we correctly computed the sum of `x` and `y`, we never actually do anything with it. Hence the "unused variable" error. We fix this error by simply calling `printf` with the value of `z` in `math2.c`.

- Note that the compiler also tells us where in the program the error is: `math1.c:19:9`. This refers to line 19, character 9.

3.4 `math3.c`

- `math3.c` demonstrates a problem with precision:

```
/**************************************************************************
 * math3.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Computes and prints a floating-point total.
 *
 * Demonstrates loss of precision.
 **************************************************************************/

#include <stdio.h>

int
main(void)
{
    float answer = 17 / 13;
    printf("%.2f\n", answer);
}
```

17/13 will evaluate to a little over 1, so surely the 32 bits of a `float` will be enough to represent this value. When we print it out, we use the formatting string `%.2f` to tell `printf` to display two digits after the decimal point.

- When we compile and run `math3.c`, we get the incorrect value 1.00 printed out. What's going on? 17 and 13 are both integers, so when we divide them, C defaults to integer division and provides an integer result. The digits after the decimal point are *truncated*, or chopped off.
- To solve this problem, we can make use of *typecasting*, which allows us to convert from one variable type to another. We can explicitly cast 13 to a floating point value by writing `(float) 13`. Another way to convince the compiler that 13 should be stored as a floating point value is to write it as 13.0. We do this in `math4.c` and thus get the correct answer to our division problem.
- Don't forget our discussion of floating point imprecision from last week. Imprecision in floating points is what enabled the guys in Office Space to siphon money from Initech.
- Question: why do we cast 13 but not 17? It suffices to cast only one to a `float` in order for C to default to floating point division.
- Question: if the result of dividing 17 by 13 was an integer, how did it get converted to a `float` in the first place? By assigning it to a variable with type `float`, it was implicitly cast to a `float`. Thus, the decimal point was added.

4 More Programming Constructs (15:00–50:00)

4.1 `nonswitch.c` and `switch1.c`

- `nonswitch.c` demonstrates the use of the "and" operator:

```
/**************************************************************************
 * nonswitch.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Assesses the size of user's input.
 *
 * Demonstrates use of Boolean ANDing.
 **************************************************************************/

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

int
main(void)
{
    // ask user for an integer
    printf("Give me an integer between 1 and 10: ");
    int n = GetInt();

    // judge user's input
    if (n >= 1 && n <= 3)
        printf("You picked a small number.\n");
    else if (n >= 4 && n <= 6)
        printf("You picked a medium number.\n");
    else if (n >= 7 && n <= 10)
        printf("You picked a big number.\n");
    else
        printf("You picked an invalid number.\n");
}
```

Pretty straightforward: we're telling the user what kind of number he picked. Certainly it works as it's supposed to, but is there a better way to do this from a style or readability standpoint? You betcha![1] We do so using a switch, as we see in `switch1.c`:

[1] And yes, there's always a better design. Your program will never quite be perfect. Le sigh.

```
/**************************************************************************
 * switch1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Assesses the size of user's input.
 *
 * Demonstrates use of a switch.
 **************************************************************************/

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

int
main(void)
{
    // ask user for an integer
    printf("Give me an integer between 1 and 10: ");
    int n = GetInt();

    // judge user's input
    switch (n)
    {
        case 1:
        case 2:
        case 3:
            printf("You picked a small number.\n");
            break;

        case 4:
        case 5:
        case 6:
            printf("You picked a medium number.\n");
            break;

        case 7:
        case 8:
        case 9:
        case 10:
            printf("You picked a big number.\n");
            break;

        default:
           printf("You picked an invalid number.\n");
    }
```

```
}
```

Functionally, this program is identical to `nonswitch.c`. Arguably, though, it's more readable, albeit longer.

- In the above program, each of the `case` statements is compared with the variable provided to `switch` at the beginning of the block. If the case matches the variable, then its lines of code are executed. Notice that the cases lump together unless we explicitly type `break`. Thus 1, 2, and 3 fall together, 4, 5, and 6 fall together, and 7, 8, 9, and 10 fall together. This is a common source of bugs in programs! Don't forget the `break` statements! If you were to forget them here, all of the statements would print if you picked a number between 1 and 3.
- Note that switches are somewhat limited in that they must hinge on a single variable, so for more complex conditions, if-else if-else might still be the way to go.
- Question: is there a way to handle large ranges of numbers with switch statments? No, you would need to use if-else if-else.

4.2 `positive1.c`

- One type of loop that we haven't seen an example of is the do-while loop. do-while loops have a particular use. The while block comes after the do block, and, as you might expect, executes after it as well. This is useful when we want to guarantee that some block of code be executed **at least** once no matter what. Let's take a look at an example in `positive1.c`:

```
/**************************************************************************
 * positive1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Demands that user provide a positive number.
 *
 * Demonstrates use of do-while.
 **************************************************************************/

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

int
main(void)
{
    // loop until user provides a positive integer
```

```
    int n;
    do
    {
        printf("I demand that you give me a positive integer: ");
        n = GetInt();
    }
    while (n < 1);
    printf("Thanks for the %d!\n", n);
}
```

Obviously, we want to want to ask the user for his input at least once no matter what. Now we'll either continue to execute the loop (which will ask the user for input again) if the user didn't provide the input we were looking for (in this case a positive integer).

- It seems a little ugly that we're declaring the variable `n` outside the do block, but the reason is that variables declared within blocks of code like this only exist within those blocks of code, not outside of them. This is called *scope*. If we didn't declare `n` outside the loop, it wouldn't exist when we wanted to print it out at the end of our program and so the compiler would throw an error saying 'n' undeclared. We could even declare `n` outside of the `main` function so that it would be available to every function in our program. However, this would make it a *global variable*, which can cause unexpected problems and is generally considered to be bad style.

- Question: could we be more specific when we inform the user that he has provided bad input, perhaps by telling him what his input was? Yes, we certainly could, but it would take some restructuring and might even be more easily accomplished without a do-while loop.

- Question: is there a way to destroy a variable within the scope of a program? No, but we will have that feature in PHP.

- Question: does `n` have a default value? In short, it depends. We'll come back to this later, but for now, assume that if you don't explicitly initialize a variable with a value, it will contain a garbage value, i.e. some collection of bits that just happen to be stored in the memory that your program allocated for that variable.

- Question: is there any limit on the number of conditions you can combine together within the parentheses after `while`? No. However, you might want to have an upper limit in mind just for readability's sake.

- Question: can you reassign the value of `n` within the code block of the loop? Yes, just be careful not to redeclare it.

4.3 positive2.c

- positive2.c implements the exact same program as positive1.c but with the use of a boolean variable:

```
/**************************************************************************
 * positive2.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Demands that user provide a positive number.
 *
 * Demonstrates use of bool.
 **************************************************************************/

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

int
main(void)
{
    // loop until user provides a positive integer
    bool thankful = false;
    do
    {
        printf("I demand that you give me a positive integer: ");
        if (GetInt() > 0)
            thankful = true;
    }
    while (thankful == false);
    printf("Thanks for the positive integer!\n");
}
```

Notice that the return value of `GetInt` is not actually stored in a variable, but instead directly compared to 0. If that return value turns out to be greater than 0, we set the boolean variable `thankful` to `true`. Here we also introduce the `==` operator, which returns true if it's two operands are equal to each other. Why `==` instead of `=`? The latter is actually the assignment operator, so if you wrote `thankful = false`, `thankful` would be assigned the value false instead of compared to the value false. Thus, no matter what value the user provided, the loop would not repeat itself.

4.4 positive3.c

- positive3.c is a final example that demonstrates the use of the `!` or bang operator. This inverts the value of whatever expression comes after it. So

if we write `while (!thankful)`, it reads as "while not thankful," which actually makes for pretty readable code.

```
/**************************************************************************
 * positive3.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Demands that user provide a positive number.
 *
 * Demonstrates use of !.
 **************************************************************************/

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

int
main(void)
{
    // loop until user provides a positive integer
    bool thankful = false;
    do
    {
        printf("I demand that you give me a positive integer: ");
        if (GetInt() > 0)
            thankful = true;
    }
    while (!thankful);
    printf("Thanks for the positive integer!\n");
}
```

4.5 progress1.c

- `progress1.c` demonstrates use of the `sleep` function as well as a for loop:

```
/**************************************************************************
 * progress1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Simulates a progress bar.
 *
 * Demonstrates sleep.
 **************************************************************************/
```

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

int
main(void)
{
    // simulate progress from 0% to 100%
    for (int i = 0; i <= 100; i++)
    {
        printf("Percent complete: %d%%\n", i);
        sleep(1);
    }
    printf("\n");
}
```

This code is a very simple implementation of a progress bar, counting from 0 to 100 over the course of 100 seconds or so. As you might have guessed, the `sleep` function causes the program to halt execution for some number of seconds, in this case 1. To use this function, we must include the `unistd.h` library.

- The `%%` is the escape syntax to print a literal percent character.

4.6 progress2.c

- Let's make our output a little more visually appealing:

```
/**************************************************************************
 * progress2.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Simulates a better progress bar.
 *
 * Demonstrates \r, fflush, and sleep.
 **************************************************************************/

#include <stdio.h>
#include <unistd.h>

int
main(void)
{
    // simulate progress from 0% to 100%
    for (int i = 0; i <= 100; i++)
```

```
    {
        printf("\rPercent complete: %d%%", i);
        fflush(stdout);
        sleep(1);
    }
    printf("\n");
}
```

Here, we're using `\r` instead of `\n`. `\n` specifies a newline, but `\r` specifies a carriage return, meaning that the cursor will be moved all the way to the left on the screen. Unfortunately, there is some inconsistency when it comes to newline characters across different operating systems. Linux and Mac OS use `\r` as the standard newline character, but Windows uses `\r\n`. This can lead to problems with file conversion.

- If we compile and run this program, we see that the progress indicator remains on a single line. Using `\r` causes the next line to overwrite the previous one which gives the appearance of animation since the lines are the same length.

4.7 progress3.c

- Programmatically, we can achieve the same effect from `progress2.c` using a while loop instead of a for loop:

```
/**************************************************************************
 * progress3.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Simulates a better progress bar.
 *
 * Demonstrates a while loop.
 **************************************************************************/

#include <stdio.h>
#include <unistd.h>

int
main(void)
{
    int i = 0;

    /* simulate progress from 0% to 100% */
    while (i <= 100)
    {
```

```
        printf("\rPercent complete: %d%%", i);
        fflush(stdout);
        sleep(1);
        i++;
    }
    printf("\n");
}
```

We have to be careful to include the update condition `i++` within our loop code block somewhere or else our loop will repeat infinitely.

- Question: what would happen if we counted in the reverse direction? Because the numbers are getting smaller rather than larger, they don't fully overwrite each other, so when we move from `100%` to `99%`, an extra percent character will be visible through our green cursor. To fix this, we could pass the formatting string as `%3d` which would tell the program to use 3 spaces to print the number, no matter how small or large.
- What's with the `fflush`? The operating system normally waits until it receives a `\n` character to actually print anything to the screen. This is for the sake of efficiency, so that it can print everything at once rather than a few characters at a time. However, in our case, we definitely want the printing to occur on each iteration of the loop. Calling `fflush` tells the operating system to do that. If you wanted to look up what `fflush` does, you could actually type `man 3 fflush` at the command line.[2] This opens up the manual entry on `fflush`. For a more readable version, you can go to the C Reference on the course website. For example, the `printf` entry contains all the formatting codes.

5 Typecasting (60:00–73:00

5.1 `ascii1.c`

- As `ascii1.c` demonstrates, you can actually explicitly convert an integer to an ASCII character simply by casting it:

```
/**************************************************************************
 * ascii1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Displays the mapping between alphabetical ASCII characters and
 * their decimal equivalents using one column.
 *
```

[2] The 3 is not always necessary, FYI.

```
 * Demonstrates casting from int to char.
 ****************************************************************************/

#include <stdio.h>

int
main(void)
{
    // display mapping for uppercase letters
    for (int 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 (int i = 97; i < 97 + 26; i++)
        printf("%c: %d\n", (char) i, i);
}
```

This program simply prints all the letters in the alphabet, both lowercase and uppercase, along with their ASCII mappings to integers.

5.2 ascii2.c

- In grade school, you may have passed a note to your crush or best friend[3] that was written in code. A simple way of encrypting the note would have been to shift each of the letters down by one. So "a" would become "b," "b" would become "c," and so on. How do we express this programmatically? If we cast a letter to a number and then add 1, then cast it back to a letter, we would effectively shift it down the alphabet by one. `ascii2.c` demonstrates that characters and numbers are actually interchangeable in C:

```
/****************************************************************************
 * ascii2.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Displays the mapping between alphabetical ASCII characters and
 * their decimal equivalents using two columns.
 *
 * Demonstrates specification of width in format string.
 ****************************************************************************/
```

[3] Or was she both?

```
#include <stdio.h>


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

Here we're simply casting `i` to a `char` in order to turn it into its corresponding ASCII character.

5.3 battleship.c

- Let's make things a little more interesting by implementing the Battleship gameboard:

```
   1  2  3  4  5  6  7  8  9  10
A  o  o  o  o  o  o  o  o  o  o
B  o  o  o  o  o  o  o  o  o  o
C  o  o  o  o  o  o  o  o  o  o
D  o  o  o  o  o  o  o  o  o  o
E  o  o  o  o  o  o  o  o  o  o
F  o  o  o  o  o  o  o  o  o  o
G  o  o  o  o  o  o  o  o  o  o
H  o  o  o  o  o  o  o  o  o  o
I  o  o  o  o  o  o  o  o  o  o
J  o  o  o  o  o  o  o  o  o  o
```

- So to begin, we'll probably have to print out that first row of numbers, which shouldn't be too hard. The middle of the gameboard isn't too hard, either, since we just need to print 10 lowercase o's in a row. But what about that first column of letters? Let's take a look at the code:

```
/**************************************************************************
 * battleship.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Prints a Battleship board.
 *
 * Demonstrates nested loop.
```

```
 ********************************************************************************/

#include <stdio.h>


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

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

Take a look at the second for loop. Notice we could've started at 1 and iterated *through* 10, but we chose to start from 0. This is handy in the first line when we use `i` as an offset. `'A' + 0` gives us `A`. C actually treats a `char` just like an `int`, so we don't need to explicitly cast from one to another.

- The outermost for loop is taking care of moving from one row to another while the innermost for loop is taking care of printing the column values.

5.4 `beer1.c`

- Let's take a simple problem and try to solve it in an efficient way. We want to print out all the lyrics from "99 Bottles of Beer on the Wall," but we obviously don't want to have to hardcode every line. For starters, we know that the number 99 counts down to 1, which seems pretty easy to handle with a loop. But also, the last line of the song will be "1 bottle of beer on the wall" whereas the last line of all the previous stanzas used the word "bottles" instead. So we need to convert plural to singular when it's appropriate.

```
/********************************************************************************
 * beer1.c
```

```
 *
 * Computer Science 50
 * David J. Malan
 *
 * Sings "99 Bottles of Beer on the Wall."
 *
 * Demonstrates a for loop (and an opportunity for hierarchical
 * decomposition).
 ********************************************************************************/

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


int
main(void)
{
    // ask user for number
    printf("How many bottles will there be? ");
    int 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 (int 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;
}
```

In this program, we actually ask the user how many bottles of beer he wants to start with. We capture this value by storing the output of the `GetInt` function in the variable `n`. In the next step, we do some error

checking by making sure the user's input is greater than 1. Notice that in the case the user has given an integer less than 1 as input, we print an error message and then execute the line `return 1`. Since `main` is actually a function itself, it can have return values, specifically of type `int`. Generally speaking, a return value of 0, which is implicitly returned by default, means "everything went okay." Any other return value indicates an error occurred. It's useful to return different values for different errors so that when the program breaks, you know exactly where it broke.

- Moving on to the loop, we see that it iterates downward instead of upward. This is perfectly fine (and suits our purposes of counting down from 99 here) so long as your terminating condition is eventually reached. `i--` is shorthand for `i = i - 1`.

- We took a shortcut here by writing "bottle(s)" so that the last line of the song is grammatically correct. We could do this more elegantly by checking on each iteration of the loop if `i == 1` and then acting accordingly. We might also treat the singular as a special case by iterating down to 1 instead of 0 and then hardcoding the last verse.