Computer Science 50
Fall 2010
Scribe Notes

Week 1 Friday: September 10, 2010
Andrew Sellergren

Contents

Computer Science 50
Fall 2010
Scribe Notes

Week 1 Friday: September 10, 2010
Andrew Sellergren

1 Announcements (0:00–5:00)

- This is CS50.
- 1 new handout.
- If you're ready to, you can return Scratch boards with a slip of paper including your name and HUID and one other piece of information that I can't read on the chalkboard.
- Supersections, open to students of all comfort levels, begin on Sunday. Check the course website for times and locations. If you can't make any of them, not to worry, one will be filmed.
- The first walkthrough will be this Sunday at 7 PM. One of the teaching fellows will show you how to get started on this week's problem set. If you plan on attending, you would do well to read through the problem set specification beforehand.
- Thanks for joining us! Enrollment has never been higher:

2007	283
2008	330
2009	336
2010	525

  Most of you are sophomores (43%), but the remainder of you are well apportioned among the freshman, junior, senior classes. Also, 38% of you are women, more than ever before!
- This will be our last Friday lecture ever!

2 From Last Time (5:00–9:00)

- Writing a program in C consists of three main steps, each represented by a command:

  1. `nano hello.c`
  2. `gcc -o hello hello.c -lcs50`
  3. `./hello`

  Recall that Nano is one of many text editors available (including Vim and Emacs) and GCC is one of many compilers available. The `-o` switch tells GCC to name the output file something other than the default `a.out` and the `-lcs50` switch tells GCC to link in the CS50 library. Some other flags include `-lm`, which links in the `math.h` library and `-lncurses`, which links in the `ncurses.h` library, a graphic library for C. In step 3, we actually execute our `hello` program, being careful to prepend `./` so that the computer knows to look in the present working directory to find the binary.

3 Some Simple Math (9:00–42:00)

3.1 math1.c

- Let's try doing some math in `math1.c`:

```
/**************************************************************************
 * 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()
{
    int x = 1;
    int y = 2;
    int z = x + y;
}
```

- One bit of confusion that might arise from the code above is that `=` is actually the assignment operator. When we write `x = 2`, we're assigning the value 2 to `x`, not asking if `2` equals `x`. To ask if `x` equals 2, we write `x == 2`.
- A rule of thumb: any line of C code which constitutes a statement must be terminated by a semicolon.
- If we compile and run this program, nothing happens. Is it working properly? In fact, it is, but it gives no output since we haven't told it to. Our next version, `math2.c` will actually print out the result of our computation using the function `printf` and the formatting character `%d`.

3.2 math2.c

- To save ourselves some time at the command line, we'll begin now to use the command `make`. `make` takes as an argument the name of the program we wish to create which doesn't include the `.c` suffix.
- When we run `make math2` from the command line, we see the following output:

```
gcc -ggdb -std=c99 -Wall -Werror -Wformat=0 math2.c -lcs50 \
    -lm -o math2
```

As you can see, `make` is actually shorthand for a much longer command and, by default, it's including the CS50 library and the `math.h` library.

- The `-ggdb` flag enables debugging symbols so that you can walk through your program while it's executing using a debugging program called GDB. For the next week or so, however, `printf` will be your friend as you debug your code. When in doubt, print out some variables because they might not have the value you think they do.
- `-std=C99` sets the compiler to use the C99 standard. You don't need to know the details here, but realize that this version of C has some modern conveniences that other versions don't.
- All of the `-W` flags tell GCC to be nitpicky about errors, reporting any and all of them so that you will write the best, most secure code possible.[1]
- When we run `math2`, we see the output of 3, but it runs together with the command-line prompt. This is because we didn't add a `\n` character at the end of our `printf` string.

3.3 `math3.c`

- You might think that arithmetic operators are pretty straightforward, but take a look at `math3.c` to find out why you're DEAD WRONG:[2]

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

#include <stdio.h>

int
main()
{
```

[1] Yes, it's annoying, but someday you'll miss C when you compare two variables of different types and wonder why your program runs just fine, but doesn't work.

[2] "Don't you ever get tired of being wrong?"

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

If we compile and run this program we get 1.00 as output. What went wrong? Well, the way we've written it, the numbers 17 and 13 are being stored as integers. When you divide one integer by another in C, the answer will always be an integer. Thus, even though there's a remainder when we divide 17 by 13, it gets chopped off when the answer is stored as an integer.

- What about the `.2f`? With this syntax, we're specifying the number of digits (2) after the decimal point that will be displayed.

3.4 `math4.c`

- If we instead write 17.0 and/or 13.0, then the compiler will store them as floating-point values. We see this in `math4.c`, which gives an answer (rounded to two digits) of 1.31.

3.5 `math5.c`

- Instead of writing 17.0 and 13.0, we can also explicitly *cast* the numbers 17 and 13 as `float`'s. Casting compels a given variable type to be forcibly converted into another. We do this with the following syntax, seen in `math5.c`:

```
/**************************************************************************
* math5.c
*
* Computer Science 50
* David J. Malan
*
* Computes and prints a floating-point total.
*
* Demonstrates use of casting.
**************************************************************************/

#include <stdio.h>

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

Casting will become useful when we want to convert between alphabetic characters and numbers, as with ASCII. If you want to turn the number 65 into the letter 'A,' for instance, you need only cast that `int` to a `char`. You'll be introduced to this for Problem Set 2.

3.6 `sizeof.c`

- Let's take a look at some different variable types with `sizeof.c`:

```
/**************************************************************************
 * sizeof.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Reports the sizes of C's data types.
 *
 * Demonstrates use of sizeof.
 **************************************************************************/

#include <stdio.h>

int
main()
{
    // some sample variables
    char c;
    double d;
    float f;
    int i;

    // report the sizes of variables' types
    printf("char: %d\n", sizeof(c));
    printf("double: %d\n", sizeof(d));
    printf("float: %d\n", sizeof(f));
    printf("int: %d\n", sizeof(i));
}
```

The function `sizeof` gives us the number of bytes required to store each of these variable types. For historical reasons, a `long` is 4 bytes, the same as an `int`. How big a number can an `int` store? If it's `unsigned`, meaning it can't store negatives, it can store up to the number 4 billion. Although this may sound like a lot, a maximum number of 4 billion might present a problem in industries like finance or biology. In fact, allocating too few bits to a variable is an unfortunate problem we've seen before with the Y2K bug. There is, at least, the capacity to store larger numbers with

a `long long`, which can hold up to 2^{64}. In fields like cryptography, even larger numbers are necessary for the sake of security which is why you'll see 1024-bit and 4096-bit hashes being mentioned.

- As a quick shortcut at the command line, you can use the up and down arrows to scroll through recent commands. You can also type `!` followed by the first letter of a command you recently ran to have it run again.
- Later in the semester when we begin working with databases, you'll see the need to count to very high numbers and 64- and higher-bit data types will come in handy!

3.7 `f2c.c`

- Now we'll challenge you to implement a program that converts a user's temperature input from Fahrenheit to Celsius. Use the `GetFloat()` function and the equation $°C = (5/9) \times (°F - 32)$. Hopefully, after a few minutes, your program looks something like the following:

```
/**************************************************************************
* f2c.c
*
* Computer Science 50
* David J. Malan
*
* Converts Fahrenheit to Celsius.
*
* Demonstrates arithmetic.
**************************************************************************/


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

int
main(void)
{
    // ask user user for temperature in Fahrenheit
    printf("Temperature in F: ");
    float f = GetFloat();

    // convert F to C
    float c = 5 / 9.0 * (f - 32);

    // display result
    printf("%.1f F = %.1f C\n", f, c);
}
```

- Although for the past few lectures, we've been content to write `main()`, the correct way to specify that `main` takes no arguments is to pass it the keyword `void`.
- Notice that we write 9.0 because if we do integer division with 5 and 9, we'll always get 0 (the decimal part is rounded down). We write `%.1f` to limit the values to one decimal place each.
- Question: isn't `f` already a `float`? Yes, that's true, but because of operator precedence and the placement of parentheses, the integer division will be evaluated first and the result will still be a miscalculation if either 5 or 9 isn't cast to a `float` beforehand. We could also move the $(f - 32)$ to the left side of the division to fix this.
- If we compile and run this program, we see that an input of 32 degrees Fahrenheit returns 100 degrees Celsius, which we know to be correct.

4 Programming Constructs of C (42:00–68:00)

4.1 Conditions

- Conditions take the following form in C:

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

  Between the parentheses, we place a boolean expression which, if true, will cause the code in the curly braces to be executed.

- Conditions can get more complicated if you decide to take two or more different actions based on different boolean expressions like so:

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

- Question: do you need a space between `if` and the parentheses? Technically, no, but it's considered good style to include it. Style is the third and least important of three metrics (correctness, design, and style) by which we'll evaluate your problem sets. Style is the aesthetic appearance and overall readability of your code. Style may vary from programmer to programmer in the small details, but it will be considered "good" so long as it is consistent, well-commented, properly indented, and a few other things which are described in more detail in the CS50 Style Guide. One example of varying styles that are evaluated the same is the following:

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

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

  Both are considered correct as long as only one or the other is used throughout a program.

4.1.1 `conditions1.c`

- Let's actually write some programs using conditions, beginning with `conditions1.c`:

```
/**************************************************************************
* conditions1.c
*
* Computer Science 50
* David J. Malan
*
* Tells user if his or her input is positive or negative (somewhat
* innacurately).
*
* Demonstrates use of if-else construct.
**************************************************************************/

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

int
main(void)
{
   // ask user for an integer
    printf("Id like an integer please: ");
    int n = GetInt();
```

```
    // analyze users input (somewhat inaccurately)
    if (n > 0)
        printf("You picked a positive number!\n");
    else
        printf("You picked a negative number!\n");
}
```

- `GetInt()` will prompt the user for an integer and wait until he provides one. If he provides an incorrect input (such as a `string`), it will reprompt him.
- Clearly, this program simply tells the user whether he picked a positive or negative number.
- Notice we can omit the curly braces around our conditional statements so long as they don't exceed one line each. Indenting isn't enough! If you added another `printf` line after the first one in the `else` block, it will **always** execute, even if the number is negative.
- But there's a bug here. Can you spot it? Looks like we're not properly handling the case where the user provides the number 0. After all, it's neither positive nor negative. We can fix this by using an `else if` block as well, as we do in `conditions2.c`.

4.1.2 `conditions2.c`

- Let's fix that bug:

```
/**************************************************************************
* conditions2.c
*
* Computer Science 50
* David J. Malan
*
* Tells user if his or her input is positive or negative.
*
* Demonstrates use of if-else if-else construct.
**************************************************************************/

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

int
main(void)
{
   // ask user for an integer
```

```
    printf("Id like an integer please: ");
    int n = GetInt();

    // analyze users input (somewhat inaccurately)
    if (n > 0)
        printf("You picked a positive number!\n");
    else if (n == 0)
        printf("You picked zero!\n");
    else
        printf("You picked a negative number!\n");
}
```

- Know that when a program is currently running (such as when `conditions2` is waiting for you to provide an integer), you can kill it by hitting Ctrl + C on your keyboard.

4.2 Boolean Expressions

- If you want to check whether one of two conditions is true (or actually if both are true), you can use the "or" operator (`||`) like so:

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

  In the real world, an example might be at a movie theater for a rated-R movie. If someone is 17 and over **or** he is with a parent, then let him into the movie.

- If you want to check that both conditions are true, use the "and" operator:

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

4.2.1 `nonswitch.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. 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![3] We do so using a construct called a *switch*, as we see in `switch1.c`.

4.3 The `switch` Statement

4.3.1 `switch1.c`

-
```
/***************************************************************************
 * switch1.c
 *
```

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

```
 * 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. This is a common source of bugs in programs! Don't forget the `break` statements!

4.4 Loops

4.4.1 progress1.c

- `for` loops take the following general structure:

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

Within the parentheses after the `for` keyword, there are three parts. Before the first semicolon, we are initializing a variable which will be our iterator or counter. Between the two semicolons, we're providing a threshold which, once reached, will cause the loop to be terminated. Finally, we provide code to update our iterator.

- `progress1.c` introduces us to 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 1 to 100 over the course of 100 seconds or so.

- In this case, it's perfectly reasonable to use a variable name like `i` that conveys very little information since the variable is only being used as a counter.
- The `%%` is the escape syntax to print a literal percent character.
- Question: can you get the `%` character to say in one place? Yes, by specifying a number before the `d` in the `printf` formatting character:

  ```
  printf("Percent complete: %3d%%\n", i);
  ```

  This will ensure that 3 spaces are used to print every number, no matter how many digits it contains.
- Question: what is `sleep()`? `sleep()` is a function which simply pauses execution of the program for a given number of seconds. It is defined in a the `unistd.h` library. How would we have known this? There's a Linux command called `man` that provides a manual page for any function you want to look up. Specifically, you would want to run a command like `man 3 sleep`, which will jump to the third section of the manual where function definitions live. If you're feeling lost, don't worry, Problem Set 1 will walk you through this in more detail.

4.4.2 `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");
}
```

- Ignore the `fflush(stdout)` line for now.
- If we compile and run this, we see that progress indicator remains on a single line. We achieve this by using `\r` instead of `\n`. This causes the next line to overwrite the previous one which gives the appearance of animation since the lines are the same length.

4.4.3 `progress3.c`

- In addition to `for` loops, there are `while` loops in C:

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

  Ultimately, you can implement the exact same programs with `while` loops that you can with `for` loops. In some cases, one is more advantageous or cleaner than the other.

- If there's a variable that the `while` condition depends on, it's up to you to update it within the code block of the loop. `progress3.c` demonstrates this:

```
/**************************************************************************
 * 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");
}
```

4.4.4 positive1.c

- One last type of loop is the `do-while`, demonstrated below:

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

- `do-while` has 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. We can see this put to use in taking user input 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).

4.4.5 `positive2.c`

- `positive2.c` implements the exact same program 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`.

4.4.6 `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.