Computer Science 50
Fall 2011
Scribe Notes

Week 1 Friday: September 9, 2011
Andrew Sellergren

Contents

Computer Science 50
Fall 2011
Scribe Notes

Week 1 Friday: September 9, 2011
Andrew Sellergren

1 Announcements and Demos (0:00–7:00)

- 0 new handouts.
- Thanks for coming back, all 646 of you! Once again, we've hit record numbers for CS50 enrollment. Let the fun begin! Of course, you do retain the right to leave the course if you're truly feeling overwhelmed, but please listen to David when he implores you to reach out to a member of the teaching staff before you do so. There's a good chance we'll be able to address your frustrations and fears so that you'll be able to soldier on!
- You're invited to CS50 Lunches on Fridays at 1:15 p.m.! These are opportunities for you to interact on a more personal level with the teaching staff, so we hope you'll take advantage of them.
- To get an idea of just how many people are dedicated to helping you get through this semester, check out cs50.net/staff.
- A few statistics from the first installment of new-and-improved Office Hours:
  - A total of almost 200 questions were asked, with a peak of about 100 occurring on Tuesday.
  - Average wait time ranged from under 25 seconds on Monday to just over 150 seconds on Tuesday.
- Sectioning starts later today and ends Monday at noon. Supersections, which are open to all, will be held this Sunday, Monday, and Tuesday.
- This week's problem set will be our first in C. You'll be writing a few programs, the first of which will be nothing more than a "hello, world" app. The second program will involve solving a greedy counting problem and the third program will require implementing a crude graph of isawyouharvard.com statistics. In general, we'll focus this week on getting comfortable with the syntax in C. If you're feeling adventurous, try tackling the Hacker Edition!
- This week's Walkthrough will be held on Sunday at 7 p.m.

2 From Last Time (7:00–31:00)

- As we transitioned from Scratch to C, we discovered that learning a little bit of cryptic syntax is all that stands between us and the tremendous power of a low-level programming language.
- Recall that we set up the CS50 Appliance which allows you to run an operating system within your personal computer's default operating system. Once we did so, we opened a program called gedit, a text editor with a built-in command line that allows us to write, compile, and run our code

all within the same window. If you come to find the command line in gedit too small for your needs, you can open Terminal for a full-screen version.

- By convention, we saved our C source code file with a `.c` extension. To compile our program, we ran the `make` command followed by the name of the program, which happened to be `hello`. Finally, we ran our program by typing `./hello`.
- `main` is first function we ever wrote. We talked about it as a miniature program that gets executed as soon as we run our larger program. `main` can also call other functions, a fact we made use of in order to print to the screen using `printf`, which takes one or more arguments specifying what is to be printed and how it is to be formatted.
- One advantage, by the way, of saving our file with the `.c` extension is that gedit will recognize it as C source code and perform syntax highlighting in order to illuminate keywords of the C language and make our program more readable. By default, it will also perform autocompletion. For example, it will insert a close parenthesis as soon as we type an open parenthesis, which can either be helpful or annoying. If you find it annoying, you can easily disable it in the program's preferences.
- Once we save our code, we can see that the `hello.c` file exists by typing `ls` in `~/Desktop`, where `~` stands for our home directory. Then we type `make hello` and `./hello` to compile and run our program. Note that we could run these same commands from the Terminal program as opposed to within gedit itself.
- In order to make use of functions like `printf` that were written by other programmers, we need to reference the libraries that contain their code. We do this using `#include`.
- The keyword `void` was used to denote that `main` takes no arguments in this simple program of ours.
- To make our program more interesting, we referenced the CS50 Library and used a function called `GetString` in order to retrieve some input from the user. Let's do the same this time, except let's forget the `#include <cs50.h>`. This time when we try to compile our program, we'll get all sorts of errors. Because errors can compound themselves, you should always work from the top down to fix them. Here, the first error we see is as follows:

  ```
  hello.c:6:5: error: unknown type name ‘string’
  ```

  Turns out that the `string` type we introduced to you is not native to C, but rather is defined in the CS50 Library. Because we forgot the `#include <cs50.h>` line, the compiler didn't know what the keyword `string` meant.

- Once we add back in the reference to the CS50 Library, our program looks like so:

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

int
main(void)
{
    string s = GetString();
    printf("O hai, %s", s);
}
```

When we compile and run this, we see nothing but a blinking cursor at first, but once we type in a name and hit Enter, we get the output we expect. To make it a little more user friendly, though, let's add a line that prompts for a name:

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

int
main(void)
{
    printf("Say your name: ");
    string s = GetString();
    printf("O hai, %s", s);
}
```

This time when we run the program, we get... a blinking cursor. What happened? We forgot to recompile our program.

- Incidentally, if you begin executing a program and quickly want to bring it to an end, hit Ctrl + C to kill it.
- Question: if `void` specifies that `main` takes no input, how were we able to get input from the user? Long story short, there are two types of input. Our program `hello` takes no input at the command line, i.e. when it is first executed. In contrast, `make` is a program that takes input at the command line. When we type `make hello`, we are providing the word "hello" as an argument to the `main` function of the `make` program.
- Question: is all of the whitespace strictly necessary? No, even the indentation isn't necessary for the program to compile and run. However, it makes for a more readable program, so we'll encourage it as part of proper style throughout the semester. Before Problem Set 1, we'll ask you to read a short style guide that will outline best practices.

- Question: is it necessary to explicitly return 0 at the end of a C program? No. By default, the program will return 0 once it is finished executing unless it is instructed to return something else.
- The CS50 Library defines the following functions:
  - `GetChar`
  - `GetDouble`
  - `GetFloat`
  - `GetInt`
  - `GetLongLong`
  - `GetString`

  What is a `char`? A `char` is a single character which requires 1 byte of storage. A `float` is a floating point number or, in other words, a number with a decimal point. Typically it requires 4 bytes of storage. A `double` is identical to a `float` except that it requires 8 bytes of storage which thereby allows it to store numbers with greater precision. Generally, an `int` requires 4 bytes of storage and a `long long` requires 8 bytes of storage. Keep in mind that 1 byte is 8 bits.
- For convenience, we've defined two extra types in the CS50 Library: `bool` and `string`. The verb/bool/ type is a formalization of a boolean and can take either true or false as a value. A `string` is a sequence of `char`'s that together form a word or phrase.

3 More on C (31:00–53:00)

3.1 Operator Precedence and Formatting Strings

- Many of the operators that you're familiar with from math (e.g. $+$, $-$, $*$, $/$) are identical in C. And, as in arithmetic, there is an order in which operations are applied. In the world of computer science, this is called *operator precedence*. For example, the grouping operator (i.e. parentheses) has the highest precedence in the C language. Here is a full list of operators and their precedence.
- As we mentioned earlier, the `printf` function can take many different formatting characters. Just a few of them are:
  - `%c` for `char`
  - `%d` for `int`
  - `%f` for `float`
  - `%lld` for `long long`
  - `%s` for `string`.

Be sure to use `%lld` if you want to print out a number larger than a `long` can store, lest you get a negative number when you try to print it out with `%d`!

3.2 Conditions

- As we mentioned last time, conditions in C look quite similar to conditions in Scratch:

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

This syntax implies that assuming `condition` evaluates to true, the code within the curly braces will be executed. Incidentally, the line beginning with `//` is a *comment*, which is an explanatory note regarding the code. Multi-line comments require an opening `/*` and a closing `*/`. We'll ask you to comment your code thoroughly this semester as it helps us tremendously with following along!

- Question: does `#` work for comments in C? No.
- When we have two forks in the road, we can use if-else:

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

We can even handle more than two forks in the road with the if-else if-else syntax:

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

```
    // do this other thing
}
```

- Question: does gedit correct formatting in C? No, but other tools such as Eclipse do. We'll get familiar with these later in the semester. You're welcome to use any text editor you choose.

3.2.1 Boolean Expressions

- The following code snippets demonstrate the use of Boolean expressions in C:

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

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

`||` represents "or" and `&&` represents "and."

3.2.2 `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("I'd like an integer please: ");
    int n = GetInt();

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

At the top of this file we have a multi-line comment that we've pretty-printed using a lot of asterisks in order to create the effect of a header. This, along with the single-line comments scattered throughout the program, are matters of good style. You'll thank yourself if you choose to comment your code as you write it rather than at the very end since you're less likely to wake up the morning after and wonder what the heck your program does.

- C is a *strictly typed* language, which means we have to be explicit about what type each variable is. In this case, since we're asking the user for an integer, we must assign it to a variable that has type `int`.
- It's worth noting that the line `int n = GetInt()` actually *declares* and *initializes* the variable `n` in one step. We could also have written it as follows:

  ```
  int n;
  n = GetInt();
  ```

  The important thing is that we initialize the variable, i.e. give it a starting value, as soon we declare it, i.e. announce its name and type. If we declare a variable and forget to initialize it, we can run into problems down the road.
- 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, but we're telling the user that it's negative. We can fix this by using an `else if` block as well, as we do in `conditions2.c`.

3.2.3 `conditions2.c`

- Let's fix that bug that caused 0 to be mishandled:

```
/**************************************************************************
 * 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("I'd like an integer please: ");
    int n = GetInt();

    // analyze user's input
    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");
}
```

- Question: what happens if we try to provide this program with a floating-point value? When it doubt, try it out! Turns out the program will ask you to retry entering an integer. The CS50 Library has a bit of *validation* built in here so that `GetInt` can only get an integer.
- Question: can `GetString` handle more than one line of text? No. To do that, we'll need slightly more sophisticated methods, but by the time we need this functionality, the training wheels will be off and you'll be ready to tackle it on your own!

3.2.4 Switches

- A common theme this semester will be the ability to accomplish a task in many different, but equally efficient ways. When this is the case, the way that you choose will often come down to a matter of style. For example, instead of using the if-else if-else construct, we can use a switch statement to accomplish the same:

```
switch (expression)
{
    case i:
        // do this
        break;
    case j:
        // do that
        break;
    default:
        // do this other thing
}
```

Although functionally identical, the switch statement is arguably more elegant in some cases than the if-else if-else alternative.

3.3 Loops

- `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, often named `i` by convention. Between the two semicolons, we're providing a condition which, if true, will cause another iteration of the loop to be executed. Finally, we provide code to update our iterator.

- Let's write a simple loop to demonstrate how they're used:

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

int
main(void)
{
    int n = GetInt();
    for (int i = 0; i < n; i++)
        printf("%d\n", i);
}
```

Here our iterator is named `i` and it will be incremented (increased by 1) on every iteration of the loop until it reaches a number `n` provided by the user. On every iteration of the loop, the current value of `i` is printed out. So we've written a counting program!

- What if the user provides a negative value? The program won't print anything because `i` is already greater than `n`.
- What if we made a mistake and wrote the threshold of our loop as `i > n` instead of `i < n`? Our program would run (almost) forever!
- Let's change the updates section of our loop to increase the value faster than one at a time:

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

int
main(void)
{
    int n = GetInt();
    for (int i = 0; i > n; i *= 2)
        printf("%d\n", i);
}
```

  This fancy syntax just multiplies `i` by 2 on every iteration of the loop.

- When we run this version of the program, it just prints out 0 on every line. Oops, $0 * 2 = 0$ so we need a different starting value for `i`. Once we change it to 1, the program works as intended.
- Very quickly, this program terminates. Why? Once it reached a value above 2 billion, the largest number an 32-bit integer[1] can store, `i` actually became negative. One of the bits in this integer is actually reserved for storing the sign of the integer, so once we filled that in with a 1, our program interpreted `i` as being negative and suddenly `i` was no longer greater than `n` and the loop terminated.
- In addition to `for` loops, there are `while` loops in C:

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

- For some fine, try downloading, compiling, and running `thadgavin.c`, available on the course website!

[1] A *signed* 32-bit integer, that is.