Computer Science 50
Fall 2008
Scribe Notes

Week 1 Wednesday: September 24, 2008
Andrew Sellergren, Anjuli Kannan

Contents

Computer Science 50
Fall 2008
Scribe Notes

Week 1 Wednesday: September 24, 2008
Andrew Sellergren, Anjuli Kannan

1 Announcements (0:00–3:00)

- David gets made fun of for going to Harvard, as will the rest of you once you graduate.
- 0 new handouts today.
- Don't forget to section![1]
- Sections will be announced by Saturday and will be starting on Sunday.
- It's not too late to get a Scratch Board for the Hacker Edition of Problem Set 0. Don't worry, getting a Scratch Board doesn't imply a commitment to turn in the Hacker Edition!
- "Lunch with David" on Fridays: e-mail `rsvp@cs50.net` if you'd like to join.
- Problem Set 1 will be released on Friday at 7 p.m. If you turn it in by 7:15 p.m. the same day, you will be exempt from all future problem sets and will be permitted to take over David's job as professor of the course.[2]

2 Demographics (4:00–5:00)

- The course is composed mostly of sophomores. Yikes.
- Leverett has the highest enrollment. Lowell has the lowest enrollment.[3]
- Female enrollment has increased by 60% from last fall. Lucky you, boys.

3 Academic Honesty (5:00–8:00)

- Don't cheat!
- There's a clear line between chatting and cheating.
- 15 students were sent to the Ad Board last year. They were caught by easy-to-recognize patterns in their work.
- You have 9 late days! The lowest problem set grade is dropped! These are perfect alternatives!

4 Stupid Internet Forward of the Day (8:00–11:00)

- iPhone development is one of the possibilities for the Final Project.
- As it turns out, the iPhone *will* blend.

[1] Actually, by the time you read this, it's too late.
[2] This is not true.
[3] Mather is not the best. Winthrop is.

5 How To Write a Program in C (11:00–15:00)

- From last time, the simplest program possible:
  1. Pull up a terminal window and login using `ssh` protocol.
  2. Use the command `cd` to navigate to a folder where you can open a text file in `nano` or your favorite text editor.
  3. Type the following lines:

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

     }
     ```

     For right now, don't worry about what this is doing, just know that it's necessary.
  4. If you want to print to `stdout`, use the function `printf` like so:

     ```
     int
     main(int argc, char *argv[])
     {
         printf("O hai, class!\n");
     }
     ```
  5. However, `printf` is not a function that we ourselves have written, so we must `#include` its library at the top of our file:

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

     Quotation marks are for local libraries, angle brackets are for non-local libraries.
  6. Now, let's compile. At the command prompt, type the following:

     ```
     gcc hello.c -o hello
     ```

     The `-o` flag controls the name of the output file.
  7. That's it! Try running the program by typing `hello` at the command line.
- Let's try adding a syntax error. If you take out the semicolon after the `printf` function call, GCC will yell at you and tell you that it "expected" a semicolon on a certain line. This is the good kind of error because it's very specific.
- Oops! If you try to open the file `hello` in `nano`, you'll get a bunch of junk. This is because `hello` is written in object code, or binary, which `nano` doesn't recognize.

6 Variables in C (15:00–28:00)

- Unlike in Scratch, in C you have to declare your variables, i.e. state their types.
- Write a program in C to convert a temperature from Fahrenheit to Celsius! The formula is $°C = (5/9) \times (°F - 32)$.

  1. Begin your code as before:

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

     }
     ```

  2. Write a prompt for your user:

     ```
     int
     main(int argc, char *argv[])
     {
        printf("Give me a temp in F: ");
        int f = GetInt();
     }
     ```

     The `=` sign is sometimes read as "gets" because it is assigns a value to the lefthand side. Notice that we're using CS 50's function `GetInt()`, so we're going to have to make sure to `#include` the CS 50 library at the top of our file. The `==` sign is read as "equals."

  3. But we want to be a little more versatile and account for decimals, so let's use a `float` instead of an `int`:

     ```
     int
     main(int argc, char *argv[])
     {
        printf("Give me a temp in F: ");
        float f = GetFloat();
     }
     ```

  4. Now we'll implement the actual formula. Note that we kept the parentheses just to be sure that order of operations was preserved. Also, notice that `f` is lowercase:

     ```
     int
     main(int argc, char *argv[])
     {
        printf("Give me a temp in F: ");
        float f = GetFloat();

        float c = (5/9) * (f - 32);
     }
     ```

5. Finally, we want to print the converted value to `stdout`:

```
int
main(int argc, char *argv[])
{
   printf("Give me a temp in F: ");
   float f = GetFloat();

   float c = (5/9) * (f - 32);

   printf("That's %f in C!\n", c);
}
```

   Notice that the `printf` function now takes two arguments. The `%f` is a placeholder for a `float` which we specify after the comma, namely `c`. The other placeholders are as follows:

`%c`	char
`%d`	integer
`%e`	scientific notation
`%E`	scientific notation
`%f`	double or float
`%s`	string
`%u`	unsigned int
`%x`	hex

6. Well, it compiles when we run `make program` from the command line, but when we actually execute `program` and give 212 in Fahrenheit as input (the boiling point of water, which *should* output 100 in Celsius), we get 0.0000 as output. What's going on? We have a subtle bug in our formula:

```
int
main(int argc, char *argv[])
{
   printf("Give me a temp in F: ");
   float f = GetFloat();

   float c = (5.0/9.0) * (f - 32);

   printf("That's %f in C!\n", c);
}
```

   The `(5/9)` needs to be replaced with `(5.0/9.0)` in order that it doesn't get rounded down or truncated to 0. Dividing two `int`'s will return another `int`. Thus, if the numbers don't divide evenly, information on the remainder will be lost.

7. What about "corner cases," or invalid inputs? If we type in anything but a valid number, `GetFloat()` will prompt us again. In this case, we've done it for you, but in the future, you'll need to consider how you're going to deal with corner cases.
8. The six decimal places are a little ugly. Let's cut it down to 1:

```
int
main(int argc, char *argv[])
{
   printf("Give me a temp in F: ");
   float f = GetFloat();

   float c = (5.0/9.0) * (f - 32);

   printf("That's %.1f in C!\n", c);
}
```

   This is a format string that we've added. The `printf` function will round, not truncate, to fulfill the format string. As you continue programming, you'll find that you have to make a compromise between precision and size of numbers. If you want to be very precise, you can't represent very large numbers and vice versa.

9. For one final improvement, we'll remind the user of their input:

```
int
main(int argc, char *argv[])
{
   printf("Give me a temp in F: ");
   float f = GetFloat();

   float c = (5.0/9.0) * (f - 32);

   printf("%.1f in F is %.1f in C!\n", f, c);
}
```

   Now, the `printf` function takes **three** arguments, the last two being variables for which we've provided placeholders. Notice that the variables are in the order of their placeholders!

7 Conditions (28:00–36:00)

- The if-else construct allows you to do something in one case, and do something else in all other cases:

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

- Let's take a look at `conditions1.c`:

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

         /* ask user for an integer */
         printf("I'd like an integer please: ");
         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");
}
```

- What's the bug? Code doesn't deal with case of 0

- Use the if-else if construct to do something in one case and do something else in any number of other cases:

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

- Replace the if-else in `conditions1.c` with the following:

```
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");
```

- Oops! Don't forget to use `==` instead of `=` when you're checking and not assigning a variable's value! If you're lucky, GCC will yell at you for not having double parentheses around an assignment within a condition.
- Notice we can exclude the squiggly braces if the contents of the if or else is only one line. This is simply a matter of style. Placement of brackets, spacing, and indentation are all aspects of style. Every student has her own style, but just make sure your code is consistent and readable.

8 More Boolean Expressions (36:00–40:00)

- `if (condition1 || condition2)` means if `condition1` is true *or* `condition2` is true
- `if(condition1 && condition2)` means if `condition1` is true *and* `condition2` is true
- `nonswitch.c`:

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

```
        int n;

        /* ask user for an integer */
        printf("Give me an integer between 1 and 10: ");
        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");
}
```

Notice the `<=` and `>=` syntax meaning "less than or equal to" and "greater than or equal to," respectively.

9 Switch Statements (40:00–45:00)

- Use `switch` statements if you have an `int` or `char` that you want to check the value of and do different things depending on its value.
- You could do a whole bunch of if-else if statements, but switch statements may be more readable and understandable.
- Switch statements are preferred in certain scenarios, for example a game in which different actions are taken based on user input. Shuttleboy, for one, has a large switch statement.
- `switch1.c`:

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

    /* ask user for an integer */
    printf("Give me an integer between 1 and 10: ");
    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");
    }
}
```

- When a true condition is encountered, all the code that follows is is executed until a `break` is reached. This means that the program "falls" into other cases unless you explicitly tell it not to.
- If no condition is true, the `default` case will be carried out. It is not necessary to have a default case, however.
- You can also do a switch with `char`'s (because they are just numbers), but not strings. You must use single quotes around `char`'s, as in `switch2.c` below:

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

    /* ask user for a char */
    printf("Pick a letter grade: ");
    c = GetChar();

    /* judge user's input */
    switch (c)
    {
        case 'A':
```

```
            case 'a':
                printf("You picked an excellent grade.\n");
                break;

            case 'B':
            case 'b':
                printf("You picked a good grade.\n");
                break;

            case 'C':
            case 'c':
                printf("You picked a fair grade.\n");
                break;

            case 'D':
            case 'd':
                printf("You picked a poor grade.\n");
                break;

            case 'E':
            case 'e':
                printf("You picked a failing grade.\n");
                break;

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

10 Loops (45:00–1:11:00)

- A for loop is implemented like so:

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

  The initializations (such as a variable assignment) are carried out before the first iteration of the loop, the condition is checked before each iteration of the loop, and the updates are executed after each iteration of the loop.

- `progress1.c`:

  ```
  for (i = 0; i <= 100; i++)
  {
  ```

```
    printf("Percent complete: %d%%\n", i);
    sleep(1);
}
```

The function `sleep` is self-explanatory. It takes a number of seconds as its only argument.

- Why `%%`? We are "escaping" the percent character so that it will actually be displayed and not be interpreted as the first part of a placeholder such as `%d`.
- If you try to compile `progress1.c` using the `gcc` command from the command line instead of CS 50's `make` command, the compiler will yell at you for having a variable declaration in the for loop. You need the flag `-std=c99`, which tells the compiler that we're using the 1999 version of C. This flag is built into `make`.
- If you run `progress1`, you can see that we have implemented a primitive kind of animation. Type `Ctrl+Z` to suspend the program and `fg` to resume it.
- `progress2.c`:

```
for (int i = 0; i <= 100; i++)
{
    printf("\rPercent complete: %d%%", i);
    fflush(stdout);
    sleep(1);
    }
    printf("\n");
}
```

- Now that we've done for loops, while loops are fairly straightforward:

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

- `progress3.c` implements the same progress bar as `progress2.c`, but with a while loop.
- The `\r` is called a carriage return. It resets the cursor all the way to the left on the *same* line without starting a new line. In the world of Windows, a newline is denoted by `\r\n`, whereas in Linux, a newline is denoted by `\n`. Sometimes this will cause "garbage" characters such as `^M` to appear on screen when you're moving between the two formats.

- The `fflush()` function is necessary because the `printf` function prints to a buffer before printing to the screen. Calling `fflush()` forces the program to spit out whatever is in the `printf` buffer. If it isn't called, then the for loop will constantly overwrite the contents of the `printf` buffer and only print it to the screen once at the very end.
- When you open a program and start typing before it's completely loaded, your computer will log what you're typing in a temporary buffer. Programs write to buffers rather than writing every single character one at a time.
- What if we were counting down instead of up? There's a bug such that two percent characters appear on screen because the string `100%` is longer than `99%`, `98%`, etc. The second `%` character is a remnant from when `100%` was printed. To fix this, use the format string `%3d%%`, which specifies that 3 spaces be used for *all* of the percentages.
- Remember to include an update statement to avoid infinite loops!
- Finally, we have do-while loops. These are useful when we want to make sure that a piece of code is executed `at least` once, even if the condition evaluates to false:

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

- `positive1.c`:

```
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);
```

  This program will keep asking for a positive integer until the user finally provides one.

- We've defined a data type called `bool` in the CS 50 library because it is implemented in many other programming languages. It holds a value either `true` or `false`. See `positive2.c`:

```
bool thankful = false;

// loop until user provides a positive integer
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");
```

By default, we are *not* thankful (`thankful` evaluates to `false`) because the user hasn't provided the input we're looking for. As soon as he does, we are thankful (`thankful` evaluates to `true`) and the while loop exits.

- What's different between `positive2.c` and `positive1.c`? The user's input is no longer stored, so we couldn't print it out later if we wanted. If we change the condition to `((n = GetInt()) > 0)`, the input will again be stored.
- We can do even more to condense the code. See `positive3.c`:

```
bool thankful = false;

// loop until user provides a positive integer
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");
}
```

The `!` operator means NOT, so the condition now reads "do {} while NOT thankful."