CS50 Supersection

(for those less comfortable)

Friday, September 8, 2017
3 – 4pm, Science Center C

Maria Zlatkova, Doug Lloyd

Today’s Topics

- Setting up CS50 IDE
- Variables and Data Types
- Conditions
- Boolean Expressions
- Loops
- Using the Command Line
- Compiling
- CS50 Library
- Overflow
- Floating-Point Imprecision
- Problem Set 1

Setting up CS50 IDE

CS50 IDE

1. Visit cs50.io and log in
2. In the Terminal at the bottom type `update50`. Do this **every time** you see this message, as important updates could be pushed. This shouldn't ever affect your own files, just system files!

Variables and Data Types

Variables and Data Types

- types of variables in C (for now!):

Type	Use Case
`int`	Integer variables, counters
`char`	Letters and other single characters
`float`	floating-point values
`double`	floating-point values requiring more precision
`long long`	integers that can get very large (> 4B)
`string`	words, phrases, paragraphs
`bool`	true or false

Using a Variable

- Before you can use a variable, you must first *declare* it. Declaring a variable is like creating a container out of thin air. To declare a variable, specify its type, then its name:
  - `int year;`
  - `float pi;`
- After you *declare* a variable, you may *assign* a value to that variable. Assigning a variable is like putting something into that container. We assign with =. You can only put a value of a compatible type into the variable.
  - `year = 2017;`
  - `pi = 3.14159;`
- It is possible to declare and assign a value to a variable simultaneously:
  - `string course = "CS50";`

Remember!

- You should only *declare* a variable **once**! (Doing otherwise can lead to trouble down the line...)
- You may *assign* a variable as many times as you like, after having declared it.

- If you *assign* a variable without having *declared* it first, you may see an error message like this when you try to compile your program.
  - `error: use of undeclared identifier`
- Incrementing the value of a variable

```
variable = variable + 1;
variable += 1;
variable++;
```

Conditions
Boolean Expressions

Scratch to C

- The puzzle blocks that you're familiar with from Scratch mostly translate to C as well.
- Just a matter of familiarizing yourself with the new syntax!

Boolean Expressions

- A boolean expression is a statement that evaluates to either `true` or `false`.
- In C, all values are considered true except for `0` and `false` (and a few others that are certainly not important at this stage).
- They can be simple
  - `5`
  - `var < 7`
  - `year == 2016`
  - `change >= 1.00`
- Or they can be compound
  - `year >= 2007 && year <= 2016`
  - `file_open == false || (file_exists == false && directory_created == true)`

Boolean Expressions

- Boolean expressions are most commonly used in the context of a conditional statement.
- C has a few different types of conditional statements.
  - `if`
  - `if-else`
  - `if-else if`
  - `if-else if-else`
  - `switch`
  - Ternary operator (`?:`)
- If the Boolean expression associated with a conditional statement is true, the code between the conditional statement's curly braces will execute.

Conditions - `if`

```
if (boolean-expression)
{

   // code

}
```

- If `boolean-expression` evaluates to true, then the code between the curly braces will execute.

- If `boolean-expression` evaluates to false, then the code between the curly braces will not execute.

Conditions - `if-else`

```
if (boolean-expression)
{

   // code snippet 1

}
else
{

   // code snippet 2
}
```

- If `boolean-expression` evaluates to true, then any lines of code where code snippet 1 is will execute.
- If `boolean-expression` evaluates to false, then any lines of code where code snippet 2 is will execute.
- Note that these are mutually exclusive, and exactly 1 of these 2 things will happen.

Conditions - `if-else if`

```
if (boolean-expression-1)
{

   // code snippet 1

}
else if (boolean-expression-2)
{

   // code snippet 2
}
```

- If `boolean-expression-1` evaluates to true, then any lines of code where code snippet 1 is will execute.
- Otherwise, if `boolean-expression-2` evaluates to true, then any lines of code where code snippet 2 is will execute.
- Note that these are mutually exclusive, and if both Boolean expressions evaluate to false, it's possible that nothing will happen.

Switch Statement

- Instead of using Boolean expressions, the different possible values of a variable are enumerated using *case statements*, below each of which go any number of lines of code. At the end of each *case*, you must include a `break;` so that the system knows where to stop executing lines of code.
- After the final case you specify, provide a `default` case in case none of your prior case statements catch.

Conditions - `switch`

```
char x = get_char();

switch (x)
{
   case 'a':
   case 'b':
      // code snippet 1
      break;

   case 'c':
   case 'd':
      // code snippet 2
      break;

   // ...

   default:
      // code snippet n
      break;
}
```

- If the value of `x` is ‘a’ or ‘b’ at the time of the `switch`, then any lines of code where code snippet 1 is will execute.
- Otherwise, if the value of `x` is ‘c’ or ‘d’ at the time of the `switch`, then any lines of code where code snippet 2 is will execute.
- ...
- Otherwise, if `x` has not matched any preceding case statements, then by *default*, any lines of code where code snippet n is will execute.

Ternary Operator

- The ternary operator, also known as `?:` (question-mark-colon) is used as a clever shorthand for if-else, when the code inside each of the branches of if and else are extremely short.
- It's not necessary to use it, but you may occasionally see it in code, including in our distribution code.

Conditions - Ternary Operator

Using if-else

```
int x;

if (boolean-expression)
{
   x = 1;
}
else
{
   x = 0;
}
```

Using ?:

```
int x;

x = (boolean-expression) ? 1 : 0;
```

Loops

Loops

- C offers three primary flavors of loops.
  - `while`
  - `for`
  - `do-while`
- Each type of loop has its own use case and syntax, and normally leverages a Boolean expression to determine when the looping should stop.
- Each of the three types of loops are often (but not always!) interchangeable.

`while` loops

- A `while` loop's primary use case is to run code repeatedly, until some condition is false, without necessarily knowing ahead of time how many *iterations* that loop will require for the condition to be true.

- It is possible for the body of a `while` loop not to execute, if its condition is false at the outset.

Loops - `while`

```
int n = 0;

while (n < 10)
{
   n = n + 1;
   printf("%i\n", n);
}
```

- The steps of a `while` loop are:
  a. Check the Boolean expression -- in this case n < 10.
  b. If the Boolean expression is false, stop, and proceed to the next line of code after the closing curly brace.
  c. If the Boolean expression is true, execute each line of code between the curly braces one line at a time.
  d. Return to step a.

`for` loops

- A `for` loop's primary use case is to run code a specific number of times.

Loops - for

```
for (int n = 1; n <= 10; n++)
{
    printf("%i\n", n);
}
```

- The steps of a `for` loop are:
  a. Execute the initializer which normally, as here, sets the starting value of a counter variable.
  b. Check the Boolean expression -- in this case, n <= 10.
  c. If the Boolean expression is false, stop, and proceed to the next line of code after the closing curly brace.
  d. If the Boolean expression is true, execute each line of code between the curly braces one line at a time.
  e. Execute the updater which normally, as here, modifies the value of a counter variable (not always by +1).
  f. Return to step b.

`do-while` loops

- A `do-while` loop's primary use case is to run code repeatedly, until some condition is false, without necessarily knowing ahead of time how many *iterations* that loop will require for the condition to be true.

- Unlike a `while` loop, which may never run if its condition is false at the outset, a `do-while` loop is guaranteed to run at least one time, because the Boolean expression is only checked at the end of the loop.

Loops - do-while

```
int n = 0;

do
{
    n++;
    printf("%i\n", n);
}
while (n < 10);
```

- The steps of a `do-while` loop are:
    a. Execute each line of code between the curly braces, one line at a time.
    b. Check the Boolean expression -- in this case n < 10.
    c. If the Boolean expression is false, stop, and proceed to the next line of code after the condition.
    d. If the Boolean expression is true, return to step a.

Using the Command Line

Command Line

- Though CS50 IDE provides you with a file browser/tree on the left side of the screen, you may discover that you prefer to work in the terminal at the bottom of the window.
- As you become familiar with the commands, you may find that it is faster or easier to do the same things you would otherwise do with the mouse.

Command Line - Listing a Folder’s Contents

- You can get a list of all the files and folders inside your current folder (aka *directory*) in your CS50 IDE with the `ls` command, which is short for “list”.

Command Line - Creating a Folder

- You can create a directory in your CS50 IDE with the `mkdir` command.

```
mkdir <directory name>
```

```
mkdir pset1
```

Command Line - Navigate to Another Folder

- You can navigate to another directory in your CS50 IDE with the `cd` command.

```
cd <directory name>
cd pset1
cd .. (takes you “up one level” to the parent directory)
cd (takes you back to your workspace directory)
```

- Be sure to use `ls` to see which directories you may navigate to! Without specifying a more complex *relative path*, you can only generally move to those directories that you can see from your current position in the file tree!

Command Line - Delete Files and Directories

- You can delete a file with the `rm` command (*remove*). This will remove those files from the system -- be careful, this is irreversible!

```
rm <target>
```

```
rm mario.c
```

- You can also delete an empty directory with the `rmdir` command. This will remove the directory from the system only if it is empty.

```
rmdir <directory>
```

Compiling

Compiling

- In this class, we'll be writing quite a bit of code, initially using a language called C, but ultimately in Python, JavaScript, and other languages.
- A computer, however, doesn't understand programming languages in the same way that humans do. Computers require a program's *source code* to be transformed into *machine code* or *object code*, which is ultimately just 0s and 1s, that it knows how to process.

Compiling

- When we write a file in C, we need to explicitly *compile* our code down to 0s and 1s. We do this using a utility called `make`, which relies on a compiler called `clang`.
  - If you have a file called “hello.c” and you want to compile it into a program called “hello”, all you need to do is “`make hello`” from within the same directory where hello.c lives.
  - Later in the semester, we’ll learn about Makefiles, which allow us to build more complex, multi-file programs.
- As we’ll see, some programming languages (including Python) hide this “translating” step from you, but still do it behind the scenes.

Compiling

- Assuming your code is perfect, you'll be rewarded with an executable that does exactly what you want.
- More likely, though, is that you will encounter some *compiler errors*. It's okay, it happens to everyone--even experienced programmers!
- Compiler errors are notorious for being arcane and indecipherable, so we've written a tool called help50 to make error messages a bit more user-friendly. To use it when you have compiler errors, simply prepend "`help50`" to your use of the `make` command.

```
help50 make hello
```

CS50 Library

CS50 Library

- A tool that makes it easier to write and debug programs, get user input, and more!

```
get_char
get_double
get_float
get_int
get_long_long
get_string
```

Useful Functions

- Functions that prompt the user for input and can return chars, doubles, floats, ints, long longs, and strings that can be stored in variables

```
int course = get_int("What is the course number for this class? ");
printf("This is CS%i.\n", course);
```

```
Output: This is CS50.
```

eprintf

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

int main(void)
{
    string name = get_string();
    eprintf("hello, %s\n", name);
}
```

- Output is different this time:
  - `program:program.c:7: hello, doug`

Data Representation: Integer Overflow

Integer Overflow

- Occurs when the result of an arithmetic operation is a value that is too large to fit in the space for a given variable
- Integers have finite ranges in computers and when the result of an arithmetic operation cannot be represented, it overflows.
    - This can lead to inconsistencies in programs

Integer Overflow

```
  1 1 1 1 1 1 1 1
+               1
-----------------
  0 0 0 0 0 0 0 0
```

Data Representation: Floating-Point Imprecision

Imprecision

- Because a computer must represent everything in binary, some values are extremely difficult for a computer to represent.
- Much like we are not able to finitely represent ⅓ as a decimal, a computer is also not capable of displaying some numbers easily.
- One major limitation for a computer, though, is that as humans we have an infinite number of place values that allow us to get ever closer to ⅓. The amount of space a computer has to represent a value is finite!

1/10

- If 1/10 is stored in a *float*, we only have **32** bits to work with, so we have to stop at some point.
  - A *double* gets us to **64**. Better, but still finite.
- That means that at some point, the computer will reach its limit. It will have to decide that, for all intents and purposes, some number is “close enough” to 1/10. And usually, for what it’s worth, it is good enough.
- This happens for just about every floating point number. And the effects can cascade.
- Take-away: You cannot represent infinitely many floating-point values with a finite number of bits, so approximation happens, and approximation is imprecise.

Problem Set 1

Problem Set 1

- hello.c
- mario.c
- cash.c / credit.c

How can we print a square of #’s using loops?

######

######

######

######

######

How can we print a square of #’s using loops?

```
for (int i = 0; i < 5; i++)
{
     for (int j = 0; j < 5; j++)
     {
          printf("#");
     }

     printf("\n");
}
```

######

######

######

######

######

How can we print a square of #’s using loops?

```
for (int i = 0; i < 5; i++)
{
    for (int j = 0; j < 5; j++)
    {
        printf("#");
    }

    printf("\n");
}
```

How can we print a square of #’s using loops?

```
for (int i = 0; i < 5; i++)
{
     for (int j = 0; j < 5; j++)
     {
          printf("#");
     }

     printf("\n");
}
```

#

How can we print a square of #’s using loops?

```
for (int i = 0; i < 5; i++)
{
     for (int j = 0; j < 5; j++)
     {
          printf("#");
     }

     printf("\n");
}
```

##

How can we print a square of #’s using loops?

```
for (int i = 0; i < 5; i++)
{
     for (int j = 0; j < 5; j++)
     {
          printf("#");
     }

     printf("\n");
}
```

###

How can we print a square of #’s using loops?

```
for (int i = 0; i < 5; i++)
{
     for (int j = 0; j < 5; j++)
     {
          printf("#");
     }

     printf("\n");
}
```

#####

How can we print a square of #’s using loops?

```
for (int i = 0; i < 5; i++)
{
     for (int j = 0; j < 5; j++)
     {
          printf("#");
     }

     printf("\n");
}
```

######

How can we print a square of #’s using loops?

```
for (int i = 0; i < 5; i++)
{
     for (int j = 0; j < 5; j++)
     {
          printf("#");
     }

     printf("\n");
}
```

######

How can we print a square of #’s using loops?

```
for (int i = 0; i < 5; i++)
{
     for (int j = 0; j < 5; j++)
     {
          printf("#");
     }

     printf("\n");
}
```

#####

#

How can we print a square of #’s using loops?

```
for (int i = 0; i < 5; i++)
{
     for (int j = 0; j < 5; j++)
     {
          printf("#");
     }

     printf("\n");
}
```

#####

##

How can we print a square of #’s using loops?

```
for (int i = 0; i < 5; i++)
{
     for (int j = 0; j < 5; j++)
     {
          printf("#");
     }

     printf("\n");
}
```

```
#####
#####
#####
#####
#####
```

Tools

Reminders!

- reference50

  https://reference.cs50.net/

- help50

  `help50 <command>` (e.g. `help50 make hello`)

- check50 / submit50

  `check50 <identifier> / submit50 <identifier>`

- style50

  `style50 hello.c`

Questions?

lloyd@cs50.harvard.edu
maria@cs50.harvard.edu

heads@cs50.harvard.edu