Week 1, continued

This is CS50. Harvard University. Fall 2015.

Anna Whitney

Table of Contents

1. Imprecision

- Last week, we saw in `imprecision.c`[1] that `1.0/10.0` does not in fact equal `0.1` as expected. Well, actually it does equal `0.1`, of course - but the computer gets it a little bit wrong.
- Computers only have a finite amount of memory, so they have to pick and choose what values they're going to support.
- If we only have 8 bits, we can only represent 256 values (and since we use one of those values for zero, the greatest number we can represent is 255).
- Floating point values are stored a little differently, but the computer still only uses typically either 32 or 64 bits to store a floating point number—so it can't possibly represent infinitely many values.

[1] http://cdn.cs50.net/2015/fall/lectures/1/f/src1f/imprecision.c

- When we get these inaccuracies after many decimal places, we're running up against the hardware limitations of the computer.
- Paying attention to how numbers are stored in your code can be critical—this clip from Modern Marvels[2] shows how disasters can result when numerical imprecision isn't taken into account in high-precision systems.

2. Announcements

- **Supersections** this weekend; they will be filmed and streamed live for those unable to attend.
- **Problem Set 1** is live on the course website, and due next Thursday.
- **Office hours** will take place Monday through Thursday this week.

3. C

- Let's return to our canonical program from last time:

```
#include <stdio.h>

int main(void)
{
    printf("hello, world"\n);
}
```

- Recall that `#include <stdio.h>` allows us to use the functions of the standard I/O library, written by other programmers in the past, and declared in a file called `stdio.h` elsewhere in our system.
- We introduced `main` last week as the analog of Scratch's `[ when [green flag] clicked ]` block. In C and several other languages, your first function must be called `main`. We'll explain later what `int` and `void` are doing here.
- `printf` is a function that prints out a formatted string. It takes one or more arguments (also known as parameters or simply inputs). The first is a string - a word or phrase or even a whole essay - which you usually want to terminate with a `\n` to ensure that the output ends with a newline. Subsequent arguments tell `printf` what values to

[2] http://youtu.be/7yFh7v6XMTo?t=4m1s

fill in for any format strings (such as `%f` for a floating-point number) that you included in the first string argument.

  # `\n` in the above is what's known as an **escape character** - rather than being interpreted literally as a backslash and the letter n, it tells the compiler to do something else. In this case, that "something else" is starting a new line.

- Semicolons indicate the end of statements, and curly braces delineate blocks of code (like the structures of control puzzle pieces in Scratch).
- Volunteer Kopal acts as the `printf` function, accepting input from David (acting as the `hello` program that calls `printf`) written on a piece of paper and writing the input given on the touchscreen, simulating the effect of `printf`.
- Another volunteer, Ife, represents the `GetString` function. Kopal, as `printf`, writes "State your name" on the touchscreen. Ife then gets a name from the audience and brings it back. David stores the returned name, "Nik", in a variable called `s` (by writing it on a sheet of paper labeled `s`).
- David now gives Kopal/`printf` a sheet of paper that says `hello, %s\n` and the sheet of paper containing the value of `s`. He fills in the name stored in the variable `s`, "Nik", in place of the placeholder `%s`.
- This same model of message passing underlies all the code we write, as outputs of functions are passed as inputs to other functions and so on.

4. Types

- We've been talking mostly about strings thus far, but values in C can have a few other types:
  # `char`, a single character (like `a` or `7` or `%`), which takes up one **byte**, or 8 bits; uses a `printf` format code of `%c`
  # `float`, a floating-point value (a number with a decimal point, like `10.0` or `3.14159`), which takes up 32 bits (four bytes); `%f`
  # `double`, a floating-point number that takes up twice as much space as a `float` (so 64 bits/8 bytes); also `%f`
  # `int`, an integer, also 32 bits/4 bytes, meaning that the largest integer we can represent is roughly 4 billion; `%i` or `%d`
  # `longlong`, a 64 bit integer (which can represent much larger values!); '%lld`

- And from the CS50 Library, found in `cs50.h`:
  - `bool`, true or false
  - `string`, a sequence of characters
- We can also use various escape sequences in `printf` format strings:
  - `\n` for a newline
  - `\t` for a tab character
  - `\"` to include a double-quote in the middle of a `printf` format string (since a bare double-quote would make the compiler think it had reached the end of the string!)
- In the CS50 Library, we provide functions like `GetString`, `GetInt`, `GetFloat`, `GetLongLong` and so on, that let you get input of a specific type from the user. These functions include error checking to prevent the user from providing invalid input.

4.1. Conditions

- Conditions have the following structure:

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

- The `//` in line 3 marks a comment, English words directed at yourself or other readers of your code. Lines starting with `//` (or multi-line blocks beginning with `/*` and ending with `*/`) tell the compiler not to look for actual instructions here.
- There can also be two exclusive branches:

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

- Or three:

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

- Boolean expressions (the conditions inside the conditional) can be combined with `&&` as "and", and `||` as "or":

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

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

- **Switches** express the same thing as certain `if`/`else if`/.../`else` constructs, but can be more elegant and involve fewer curly braces. They provide no additional functionality that can't be done with regular conditionals, but can sometimes be stylistically preferable.
- You can use a switch whenever all the conditions of your conditional would be of the form `expression == value` for the same expression but different values.

```
switch (expression)
{
    case i:
        // do this
        break;

    case j:
        // do that
        break;

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

4.2. Loops

- One type of loop in C is the `for` loop, which has the following basic structure:

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

- A specific example:

```
for (int i = 0; i < 50; i++)
{
    printf("%i\n", i);
}
```

In this case, `int i = 0` is the **initialization** of the loop, telling it to start counting at zero by creating a variable called `i` and assigning it the value `0`.

`i < 50` is the **condition** of the loop: immediately after the initialization, and at the start of every step of the loop thereafter, the condition is checked, and the code in the body of the loop will only be executed if the condition evaluates to `true`.

`i++` is the **update** of the loop, which will be executed after the body of the loop to move to the next step.

We've put `printf("%i\n", i);` in the **body** of the loop, so this code will print the numbers from `0` to `49` (not `50`, because when we update to `i = 50`, the condition `i < 50` evaluates to `false` and the body of the loop is not executed).

The curly braces are not syntactically required if the body of the loop is only one line, but we will always use them in class (and we request that you do too!) for clarity and to prevent mistakes.

4.3. Integer Overflow

- Just as floating point values have limits on their precision, integers have limits on the size of values they can represent.
- For a 32-bit integer, the maximum value is roughly 4 billion.
- When a binary number overflows, we go from a value like this (255 stored in 8 bits):

```
128     64     32     16     8      4      2      1
1       1      1      1      1      1      1      1
```

To a value like this:

```
??      128    64     32     16     8      4      2      1
1       0      0      0      0      0      0      0      0
```

But this is still only an 8-bit value, so there's nowhere to put the leading 1, and instead we get:

```
128     64     32     16     8      4      2      1
0       0      0      0      0      0      0      0
```

- We can see the effects of these limits on integers in various software:

In the video game Lego Star Wars, the number of coins you can collect is capped at 4 billion exactly - from which we can infer that the original developer for this game used a 32-bit integer to store the user's number of coins.

In the original Civilization game, each world leader was assigned an aggressiveness score, and Gandhi was given the lowest score of 1. If a nation transitioned to democracy in the game, the leader's aggressiveness score was decreased by

2. However, the aggressiveness scores were stored in unsigned 8-bit integers (meaning they couldn't be negative) - so decreasing Gandhi's aggressiveness score to -1 had the effect of looping around to 255 (so Gandhi became the most aggressive leader in the game!)

The Boeing 787 would lose all power after 248 days of continuous operation due to an integer overflow in the control units of its power generators (the workaround solution is to reboot the plane more often than that!)

4.4. Loops, continued

- Slightly different from a `for` loop, we have a `while` loop, that merely depends upon a single condition which is checked before every iteration of the loop:

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

- Similarly, in a `do-while` loop, the condition is checked after every iteration of the loop (as indicated by the syntax):

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

4.5. Variables

- As we've discussed, a variable in C has a particular type, which must be declared when the variable is created. Here, the first line creates a new variable of the type `int`, and the second assigns a value of `0` to it. The declaration of the variable and assigning it a value can happen as far away from each other in code as you like, but for clarity it's best to keep them close together.

```
int counter;
counter = 0;
```

- A more succinct way to write the above code:

```
int counter = 0;
```

4.6. Functions and Arguments

- Functions are followed by parentheses, which contain any arguments that are being passed to the function:

```
string name = GetString();
printf("hello, %s\n", name);
```

- In `function-0.c`[3], we show how to define your own function:

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

// prototype
void PrintName(string name);

int main(void)
{
        printf("Your name: ");
        string s = GetString();
        PrintName(s);
}

/**
 * Says hello to someone by name.
 */
void PrintName(string name)
{
    printf("hello, %s\n", name);
}
```

- Separating out this logic in `PrintName` is a form of abstraction, hiding the low-level implementation details of how we print the name.
- Similarly, in `function-1.c`[4], we can use the **return value** of a function:

[3] http://cdn.cs50.net/2015/fall/lectures/1/f/src1f/function-0.c
[4] http://cdn.cs50.net/2015/fall/lectures/1/f/src1f/function-1.c

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

// prototype
int GetPositiveInt();

int main(void)
{
    int n = GetPositiveInt();
    printf("Thanks for the %i!\n", n);
}

/**
 * Gets a positive integer from a user.
 */
int GetPositiveInt(void)
{
    int n;
    do
    {
        printf("Please give me a positive int: ");
        n = GetInt();
    }
    while (n < 1);
    return n;
}
```

- The `int` in `int GetPositiveInt(void)`, as well as the `void` in `void PrintName(string name)`, indicates the **return type** of the function.
- `PrintName` doesn't return anything (it just prints a name to the screen, which is a **side effect**), so its return type is `void`.
- `GetPositiveInt` returns an `int` - the first positive integer value the user enters - using the `return` command. Any non-void function must have a return value (if you don't have a `return` command, your return value is assumed to be `0`, for reasons we'll discuss later in the course).
- In `return.c`[5], we have another example of returning a value from a function:

[5] http://cdn.cs50.net/2015/fall/lectures/1/f/src1f/return.c

```c
#include <stdio.h>

// function prototype
int cube(int a);

int main(void)
{
    int x = 2;
    printf("x is now %i\n", x);
    printf("Cubing...\n");
    x = cube(x);
    printf("Cubed!\n");
    printf("x is now %i\n", x);
}

/**
 * Cubes argument.
*/
int cube(int n)
{
    return n * n * n;
}
```

- In this case, this function both accepts an input argument (an `int`, which we're calling `n`) and outputs, or returns, a value.

5. Problem Set 1

- On this problem set, you'll implement in C an ASCII version of Mario's pyramid (or a more challenging version in the hacker edition!)
- You'll also implement a **greedy algorithm** for determining the coins necessary when giving change, and investigate rates of water flow.