Tutorial 4

- Loops
- Overflow
- Data Types
- Debugging Tips
Loops: for loops & while loops

- Using a loop to solve a problem is called *iteration*.

- **while** is similar to if statements but **while** repeatedly “loops back” and executes the **statement until the expression is false**.

- General format of a **while** loop:
  
  setup statement(s)
  while (expression) {
    body statement(s)
    update statement(s)
  }

- **for** loops are a “condensed” version of a **while** loop.
for vs. while

Recall the for syntax.

\[
\text{for (setup; expression; update) \{ body statement(s) \}}
\]

This while example

\[
i = 100; \quad \text{// setup}
while (i >= 0) \{ \text{// expression}
    printf("%d\n", i); \quad \text{// expression}
    --i; \quad \text{// update}
\}
\]

is equivalent to

\[
\text{for (i = 100; i >= 0; --i) \{}
    printf("%d\n", i);
\}
\]
Loop: Exercise

Define the following C function: (use iteration)

// draw_circle(size) draws a circle inside of a square
// with dimensions (size * 2 + 1)
// requires: size >= 1
// effects: produces output

- There is a simple example of the output in "simple.expect"
Integer Overflow: Introduction

- Any variable in C takes up a certain amount of memory (bits).
- This limits the range of values that can be represented.
- Any time you try to go past this limit it is called an “overflow”
Integer Overflow

- A variable of type `int` allocates 32 bits of memory.
- We want to be able to represent negative and positive numbers, so roughly half of this range is negative and roughly half is positive.
- Using this logic, Integers range from $-2^{31}$ to $2^{31} - 1$, or $-2147483648$ to $2147483647$
Overflow

As an INT it is impossible to represent outside of the range of:

<table>
<thead>
<tr>
<th>INT_MIN</th>
<th>$-2^{31}$</th>
<th>$-2147482648$</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT_MAX</td>
<td>$2^{31} - 1$</td>
<td>$2147482647$</td>
</tr>
</tbody>
</table>

which is why we have other data-types
Integer Overflow Example

The following function can overflow for large values of $a$ and $b$.

```c
// find_mid(low, high): returns the middle integer between
// two boundaries, low and high, inclusively
// [round down to the whole integer]
// Requires: 0 <= low <= high
int find_mid(int low, int high) {
    return (low + high) / 2;
}
```

Even though it can never return a number larger than INT_MAX, the result of computing $(a + b)$ is undetermined.

**Practice:** On seashell, implement the `find_mid` function that would fix the issue above
Practice Problem: Overflow

The function `not_overflow_add(a, b)` returns true if adding non-negative integers `a` and `b` will not cause overflow, otherwise, returns false.

For example,

```
not_overflow_add(1, 0);     // => true
not_overflow_add(INT_MAX, 1); // => false
```
# Data Types

<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Description</th>
<th>Printf</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Integer (numbers)</td>
<td>%d</td>
</tr>
<tr>
<td>char</td>
<td>Characters</td>
<td>%c</td>
</tr>
<tr>
<td>float</td>
<td>Floating Point decimal numbers</td>
<td>%f</td>
</tr>
<tr>
<td>double</td>
<td>Double precision floating value</td>
<td>%f</td>
</tr>
</tbody>
</table>
Characters

Characters are **integers** that are typically used to hold single pieces of text data. They are 8-bit (max value of 127).

```java
char nine = '9';
char not_nine = 9; // Be careful! This is a tab!

char a = 'a';
char also_a = 97; // Equivalent, but bad practice.

char space = ' ';
char newline = '\n'; // Some characters use escape codes.
```

Later on, we will learn how to use a series of characters in a row to represent more complicated text like words and sentences.
Debugging Tips

• Use trace statements:
  – Print out the values of variables.
  – Print out statements to show control flow.

• Automate:
  – Write your own tests!

• Simplify:
  – Comment out parts that aren’t a likely cause.
  – Remove components until you isolate the problem.
  – Writing modular code helps immensely.