(6-1) Iteration in C
H&K Chapter 5

Instructor - Andrew S. O’Fallon
CptS 121 (February 19, 2020)
Washington State University
Iterative Constructs (1)

- Recall that algorithms are composed of three different kinds of statements:
  - **Sequence**: the ability to execute a series of instructions, one after the other.
  - **Conditional**: the ability to execute an instruction contingent upon some condition.
  - **Iteration**: the ability to execute one or more instructions repeatedly.

- This week, we'll learn about loops: the ability to repeatedly execute a sequence of statements.
Iterative Constructs (2)

- You have already seen many examples of iteration
- Consider, for example, the following segment of the tollbooth application presented in lecture:

```c
axles1 = read_num_axles(infile);
weight1 = read_weight(infile);
axles2 = read_num_axles(infile);
weight2 = read_weight(infile);
axles3 = read_num_axles(infile);
weight3 = read_weight(infile);
toll1 = compute_toll(axles1, weight1);
toll2 = compute_toll(axles2, weight2);
toll3 = compute_toll(axles3, weight3);
display_toll(axles1, weight1, toll1);
display_toll(axles2, weight2, toll2);
display_toll(axles3, weight3, toll3);
```

- Notice that this segment repeats four statements:

```c
axles = read_num_axles(infile);
weight = read_weight(infile);
toll = compute_toll(axles, weight);
display_toll(axles, weight, toll);
```
Iterative Constructs (3)

How to decide when a loop is needed
- Are any steps repeated?
  - No → No loop required
  - Yes → Do you know in advance how many steps are repeated?
    - No → Use a conditional loop
    - Yes → Use a counting loop

Kinds of loops
- Counting loop (for or while): executes a fixed number of times
- Sentinel-controlled or Endfile-Controlled loop (for or while): (process data until a special value is encountered, e.g., end-of-file)
Iterative Constructs (4)

- Kinds of loops (cont.)
  - *Input validation loop* (*do-while*): Repeatedly accept interactive input until a value within a specified range is entered
  - *General conditional loop* (*while, for*): Repeatedly process data until a desired condition is met

- Let’s look at examples of each of these loop patterns, and how they are implemented in C
Counter Loops (1)

- Pseudocode for a counter loop:
  
  ```
  Set loop counter to 0
  while (loop counter < final count)
      … (Do data processing)
      add 1 to loop counter
  endwhile
  ```
Counter Loops (2)

- Implementing Counter Loops: the **while** loop

```plaintext
while (<repetition-condition>)
{
    <body>
}
```
Counter Loops (3)

- **Notes on while loops:**
  - `<repetition-condition>` is evaluated at beginning of loop. If it evaluates to true, the loop body is executed. If it evaluates to false, control shifts to first statement after loop body.
  - `<body>` contains one or more C statements.
  - After last statement in `<body>` is executed, control is shifted back to beginning of loop, and `<repetition-condition>` is re-evaluated.
  - “Progress” must be made within the loop. That is, something must be done so that `<repetition-condition>` eventually evaluates to false. Otherwise we have an “infinite loop”
Example: Notice that the Tollbooth code segment we looked at earlier can be rewritten as a counter `while` loop in C:

```c
int count;
count = 0; /* Initialize counter */
while (count < 3)
{
    axles = read_num_axles(infile);
    weight = read_weight(infile);
    toll = compute_toll(axles, weight);
    display_toll(axles, weight, toll);
    count = count + 1; /* increment counter */
}
```
Counter Loops (5)

Another alternative for implementing Counter Loops: the \texttt{for} loop

\begin{verbatim}
for (<initialization>; \\
    <repetition-condition>; \\
    <update-expression>) \\
{
    <body>
}
\end{verbatim}
Counter Loops (6)

Notes on for loops:

- `<initialization>` statement initializes the loop control variables before loop is executed the first time.
- `<repetition-condition>` is tested at beginning of loop. If it is true, loop `<body>` is executed.
- `<body>` contains one or more C statements.
- After last statement in `<body>` is executed, control is shifted back to beginning of loop. Then, `<update-expression>` is executed. Finally, `<repetition-condition>` is re-evaluated.
- As with while loops, the `<update-expression>` must define “progress.” That is, something must be done so that `<repetition-condition>` eventually evaluates to false. Otherwise we have an “infinite loop.”
Counter Loops (7)

Example: Notice that the Tollbooth segment we just looked at can also be rewritten as a counter for loop in C:

```c
int count;
for (count = 0; count < 3; count = count + 1)
{
    axles = read_num_axles(infile);
    weight = read_weight(infile);
    toll = compute_toll(axles, weight);
    display_toll(axles, weight, toll);
}
```
Aside: Compound Assignment Operators

- Notice that the `<update-expression>`s in loops are often of the form:

  \[ \text{count} = \text{count} + 1 \]

- C defines special assignment operators to define statements of this form more compactly:
  - `count += 1` is equivalent to `count = count + 1`
  - `count -= increment` is equivalent to `count = count - increment`
  - `product *= product` is equivalent to `product = product * product`
  - `sum /= divisor` is equivalent to `sum = sum/divisor`
  - `remainder %= 2` is equivalent to `remainder = remainder % 2`
Aside: Increment and Decrement Operators (1)

- The ++ and -- operators take a single variable as their operands. The *side effect* of the operator is to increment or decrement its operand by one:
  - `count++` has the effect of `count = count + 1`
  - `count--` has the effect of `count = count - 1`

- **Note:** ++ and -- can be placed either *before* or *after* their variable operator:
  - Pre-increment or pre-decrement (e.g., `++count`, `--count`): value of expression is value of variable *after* the increment or decrement is applied
  - Post-increment of post-decrement (e.g., `count++`, `count--`): value of expression is value of variable *before* the increment or decrement is applied
Aside: Increment and Decrement Operators (2)

You try it: What are the values of $i$, $j$, and $k$ after each of the following statements is executed?

```c
int i, j, k;
i = 2;
j = 3 + i++;  
k = 3 + ++i;
i *= ++k + j--;  
i /= k-- + ++j;
```
Counter Loops (8)

- Notice that we can rewrite our previous while loop with these new operators:

```c
int count;
count = 0; /* Initialize counter */
while (count < 3) {
    axles = read_num_axles(infile);
    weight = read_weight(infile);
    toll = compute_toll(axles, weight);
    display_toll(axles, weight, toll);
    count++; /* increment counter */
}
Counter Loops (9)

- Notice that we can also rewrite our previous for loop with these new operators:

```c
int count;
for (count = 0; count < 3; count++)
{
    axles = read_num_axles(infile);
    weight = read_weight(infile);
    toll = compute_toll(axles, weight);
    display_toll(axles, weight, toll);
}
```
Counter Loops (10)

- Accumulating a value
  - Suppose that we want to keep track of the total number of tolls collected by the tollbooth. We could accumulate this value from within the loop body:

```c
int count, toll_total;
for (count = 0, toll_total = 0; count < 3; count++)
{
    axles = read_num_axles(infile);
    weight = read_weight(infile);
    toll = compute_toll(axles, weight);
    display_toll(axles, weight, toll);
    toll_total += toll; /* add toll to toll_total */
}
```
Counter Loops (11)

- Loop increments other than 1

```c
/* Conversion of Celsius to Fahrenheit temperatures */
#include <stdio.h>

/* Constant macros */
#define CBEGIN 10
#define CLIMIT -5
#define CSTEP 5

int main(void)
{
    /* Variable declarations */
    int celsius;
    double fahrenheit;

    /* Display the table heading */
    printf(" Celsius Fahrenheit\n");

    /* Display the table */
    for (celsius = CBEGIN;
     celsius >= CLIMIT;
     celsius -= CSTEP) {
        fahrenheit = 1.8 * celsius + 32.0;
        printf("%d\%7.2f\n", celsius, fahrenheit);
    }
    return (0);
}
```

<table>
<thead>
<tr>
<th>Celsius</th>
<th>Fahrenheit</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>50.00</td>
</tr>
<tr>
<td>5</td>
<td>41.00</td>
</tr>
<tr>
<td>0</td>
<td>32.00</td>
</tr>
<tr>
<td>-5</td>
<td>23.00</td>
</tr>
</tbody>
</table>
Next Lecture…

- We'll discuss several additional loop patterns:
  - Conditional loops
  - Sentinel-controlled loops
  - Endfile-controlled loops
  - Flag-controlled loops
References

Collaborators

- Chris Hundhausen