16. while, do-while, Counter-Controlled for, Infinite Loops

CPSC 120: Introduction to Programming
Pratishtha Soni~ CSU Fullerton

Agenda

0. Sign-in sheet
1. Technical Q&A
2. `while` Loops
3. `do-while` Loops
4. Counter-Controlled `for` Loop
5. Infinite Loops

1. Technical Q&A

Technical Q&A

Let’s hear your noted questions about...

- This week’s Lab
- Linux
- Any other technical issues

Reminder: write these questions in your notebook during lab

2. `while` Loops

Review: For-Each Loop

- **Loop**: syntax to repeat statements
- For-each loop is one kind, covered last week

```
for (std::string argument : arguments) {
  std::cout << argument << endl;
}
```

- For-each works when we want to...
  - loop through a collection
  - visit each element exactly once
- Covers ≈80% of loops
- Today: syntax for the other 20%

while Loop

- `while`: loop **as long as** a predicate is true
- Iterates indefinitely
- Useful for
  - **Game loop:** as long as no winner, play another turn
  - **Work queue:** as long as there is more work to do, perform one task

Syntax: `while` Loop

statement:

while (*condition* **)**
body-statement

Semantics:

1. Evaluate *condition*
2. if **false**: stop loop, skip *body-statement* (program continues after the loop)
3. otherwise (**true**)
    a. execute *body-statement*
    b. go to step 1

```
int x{0};
std::cout << "Enter a number: ";
std::cin >> x;
int count{0};
while (x > 0) {
  x /= 2;
  ++count;
}
std::cout << "log_2(" << x
          << ") = " << count
          << "\n";
```

Pitfall: `while` may never iterate

Semantics:

1. Evaluate *condition*
2. if **false**: stop loop, skip *body-statement* (program continues after the loop)
3. otherwise (**true**)
   a. execute *body-statement*
   b. go to step 1

Observe: when *condition* is false to begin with, *body* **never executes**!

```
int x{0};
std::cout << "Enter a number: ";
std::cin >> x;
int count{0};
while (x > 0) {
  x /= 2;
  ++count;
}
std::cout << "log_2(" << x
          << ") = " << count
          << "\n";
```

3. `do-while` Loops

do-while

- `while`: check *condition*, then iterate
- **`do-while`**: iterate, then check *condition*
- Difference: **how the first iteration works**
  - `while`: may iterate zero times (when *condition* is initially false)
  - `do-while`: loop **always** iterates at least once
- Appropriate when
  - Loop body needs to initialize a variable before *condition*
  - A procedure always repeats at least once

Syntax: do-`while` Loop

statement:

do
body-statement
while (*condition* **);**

Semantics:

1. Execute *body-statement*
2. Evaluate *condition*
3. if **false**: stop loop, program continues after the loop
4. otherwise (**true**): go to step 1

```
int x{0};
do {
 std::cout << "Enter a positive number: ";
 std::cin >> x;
} while (x <= 0);
```

Observe:

- *body-statement* always iterates at least once
- **Semicolon** after parentheses
  - Different from all other loop syntax

Scenario: Input, Validate, Retry

- We want to read input from `cin`
- Previously: given invalid input, our programs misbehave
    - runtime error or logic error
    - no recovery
- Friendlier: error message, opportunity to retry
- Possible with `while`, but clunky

First Try: `while` Loop

- Validates that input is 1-10
- Makes user try again otherwise (ex. 12)
- This code works but is a poor pattern

```
int guess{0};
while (! ((guess >= 1) && (guess <= 10))) {
   std::cout << "Enter number 1 to 10: ";
   std::cin >> guess;
}
```

Pitfall: Iteration Depends on Initial Value

- What if 0 is valid input?
- *condition* is true to begin with
- Loop never iterates
- Subtle logic error
- Problem: initialization and loop condition are “**tightly coupled**”
  - Programmer needs to think about them together, even though they are unrelated
- Better: loop always iterates, regardless

```
int guess{0};
while (! ((guess >= 0) && (guess <= 5))) {
   std::cout << "Enter number 0 to 5: ";
   std::cin >> guess;
}
```

Improvement: `do-while` Loop

- Now user always enters at least once
- Initial value of `guess`, and loop *condition*, are **decoupled**
- Loop iterates regardless of how `guess` is initialized

```
int guess{0};
do {
   std::cout << "Enter number 1-10: ";
   std::cin >> guess;
} while (! ((guess >= 1) && (guess <= 10)));
```

```
$ ./a.out
Enter number 1-10: 22
Enter number 1-10: -9
Enter number 1-10: 4
```

4. Counter-Controlled for Loop

Counter-Controlled for Loop

- Alternative `for` loop syntax
- Predates for-each loop
- Abbreviates a loop that uses a control variable to **count up** or **count down**

Pattern: Count Up With `while`

- Goal: iterate through integers *start*, *start*+1, …, *stop*
- Convention: variable identifier `i` for "iteration"

```
int i{ start };
while (i <= stop) {
 body-statement…
 ++i;
}
```

```cpp
// print 10 through 15
int i{10};
while (i <= 15) {
    std::cout << i << "\n";
    ++i;
}
```

Output:

```
10
11
12
13
14
15
```

Syntax: Counter-Controlled for Loop

statement:

for (*init-statement***;** *condition***;** *advance-statement* **)**
body-statement

Semantics:

1. Execute *init-statement* (assign control variable)
2. Evaluate *condition*
3. if **false**: stop loop, skip *body-statement*
4. otherwise (**true**)
   a. execute *body-statement*
   b. execute *advance-statement*
   c. go to step 2

```
// print 10 through 15
int i{ 0 };
for (i = 10; i <= 15; ++i) {
    std::cout << i << "\n";
}
```

Output:

```
10
11
12
13
14
15
```

How the Two Loops Correspond

```cpp
int i{10};
while (i <= 15) {
    std::cout << i << "\n";
    ++i;
}
```

```cpp
int i{ 0 };
for (i = 10; i <= 15; ++i) {
    std::cout << i << "\n";
}
```

Observe

- for loop is more compact
- every statement/expression on the left, is also on the right
    - but in different places
- `for` loop groups all the counter logic in one place

Pattern: Count-Up for Loop

Count from *start* **up to and including** *stop*

(so *start* < *stop*)

```
for (i = start; i <= stop; ++i) {
   statement-using-i...
}
```

```cpp
// print 10 through 15
int i{ 0 };
for (i = 10; i <= 15; ++i) {
    std::cout << i << "\n";
}
```

Output:

```
10
11
12
13
14
15
```

Pattern: Count-Down for Loop

Count from *start* **down to and including** *stop*

(so *start* > *stop*)

```
for (i = start; i >= stop; --i) {
   statement-using-i...
}
```

```cpp
// print 10 down to 0
int i{ 0 };
for (i = 10; i >= 0; --i) {
   std::cout << i << "\n";
}
```

Output:

```
10
9
8
7
6
5
4
3
2
1
0
```

First Try: Iterate Through All Indices

Count from 0 up to and including *n*-1

```
for (i = 0; i <= size - 1; ++i) {
  statement-using-i...
}
```

```
std::vector<std::string> arguments{argv, argv + argc};

for (std::string arg : arguments) {
 std::cout << arg << "\n";
}
```

```
$ ./a.out cat dog bird
./a.out
cat
dog
bird
```

```
int i{0};
for (i = 0; i <= (arguments.size() - 1); ++i) {
 // can use index i
 std::cout << "argument " << i << " is "
           << arguments.at(i) << "\n";
}
```

```
$ ./a.out cat dog bird
argument 0 is ./a.out
argument 1 is cat
argument 2 is dog
argument 3 is bird
```

Opportunity for Improvement

- Counter-controlled loop is OK
- Tedious part: **<=** *size* - 1
- Mathematically,

  $i \le n - 1$

  is the same as

  $i < n$
- Streamlined pattern: **<** *size*

```cpp
std::vector<std::string> arguments{argv, argv + argc};

for (std::string arg : arguments) {
 std::cout << arg << "\n";
}

int i{0};
for (i = 0; i <= (arguments.size() - 1); ++i) {
 // can use index i
 std::cout << "argument " << i << " is "
           << arguments.at(i) << "\n";
}
```

Pattern: Iterate Through All Indices

Count from 0 up to and including *n*-1

```
for (i = 0; i < size; ++i) {
  statement-using-i...
}
```

```
std::vector<std::string> arguments{argv, argv + argc};

for (std::string arg : arguments) {
 std::cout << arg << "\n";
}

int i{0};
for (i = 0; i < arguments.size(); ++i) {
 // can use index i
 std::cout << "argument " << i << " is "
           << arguments.at(i) << "\n";
}
```

Pattern: Iterate Through Some Indices

Start at index *start*

Stop as if size is *effective-size*

```
for (i = start; i < effective-size; ++i) {
    statement-using-i…
}
```

Pattern: Iterate Arguments, Skip Command

- first element of `arguments` vector is command name
- usually need to skip it
- can use previous pattern with
  start = index 1
  effective-size = actual size

```
for (i = 1; i < effective-size; ++i) {
  statement-using-i...
}
```

```cpp
// sum arguments
std::vector<std::string> arguments{argv, argv + argc};
double sum{0.0};
for (int i = 1; i < arguments.size(); ++i) {
   sum += std::stod(arguments.at(i));
}
std::cout << "sum is " << sum << "\n";
```

Output:

```
$ ./a.out 12.5 6 3.2
sum is 21.7
```

Pitfall: Off by One

- **Off by one error:** loop start or end is 1 too high or too low
- Easy oversight to make
- **Recall**
  - first index is 0 (not 1)
  - last index is *n*-1 (not *n*)
  - counter ends up 1 too big (or 1 too small)

5. Infinite Loops

Infinite Loop

Algorithm: a process for solving a problem that

1. is defined **clearly**, and
2. **always works**, and
3. **eventually stops** (no infinite loop).

Infinite loop: loop that will never stop

- Logic error
- Wastes CPU time, energy
- Impossible to automatically detect; see *CPSC 439 Theory of Computation*

Pitfall: Advancing in the Wrong Direction

- Count-up loop must **increment** counter
- Count-down loop must **decrement** counter
- Pitfall: mix up `++i` with `--i`
- Logic error: loops get further and further away from stopping

```
for (i = 1; i <= 10; --i) {
 std::cout << i << "\n";
}
for (i = 10; i >= 1; ++i) {
 std::cout << i << "\n";
}
```

Bug

Output (of second loop):

```
10
11
12
13
...
```

Stopping an Infinite Loop

- In shell:
- **CTRL-C**: cancel (“kill”) program
    - Hold Control (Ctrl) and C button at same time
- Operating system halts program immediately

Screencast: Infinite Loop with Output

Pitfall: Counter-Controlled Loop Doesn’t Advance

- Count-up loop must **increase counter**
- Count-down loop must **decrease counter**
- **Infinite loop** if that doesn’t happen

```
double sum{0.0};
for (int i = 1; i < arguments.size(); i + 1) {
   sum += std::stod(arguments.at(i));
}
std::cout << "sum is " << sum << "\n";
```

Bug

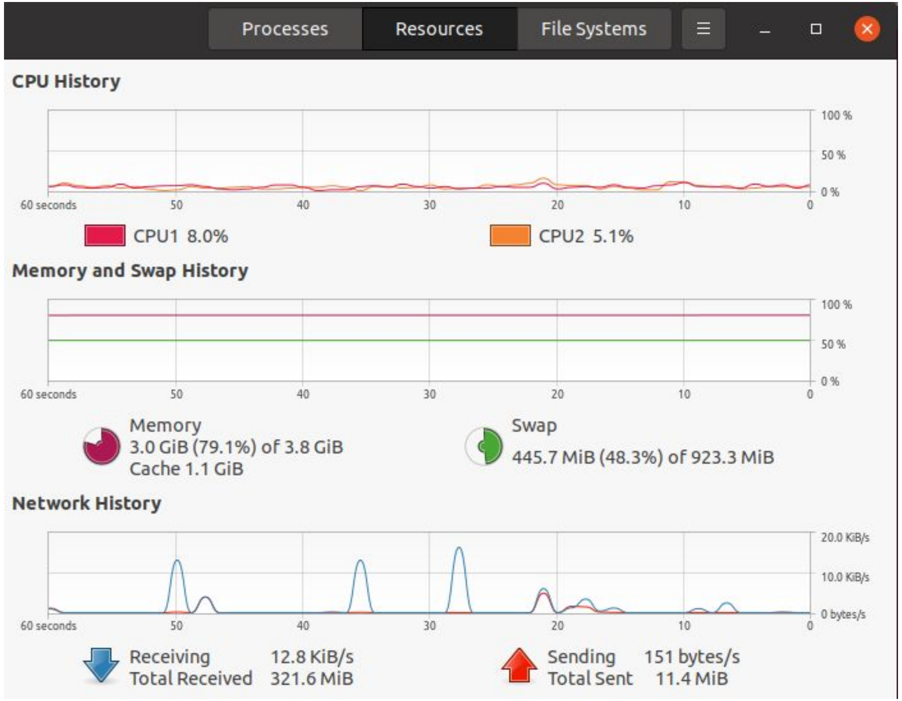
Screencast: Infinite Loop Without Output

Symptoms of Infinite Loop

- CPU “**spins**” around the loop as fast as it can
- Program output is either
  - (with output in loop body): never-ending stream of output
  - (without output in loop body): no output, program “**hangs**” (gets stuck)
- Once CPU core spends ≈100% time on your program
- Cooling fan at full speed

Diagnosing an Infinite Loop

- Scientific approach
- Use measurement instrument to observe empirical evidence
- Ubuntu **System Monitor**
  - macOS: Activity Monitor
  - Windows: Task Manager
- Shows line graph of CPU core utilization
- **Infinite loop** = one core at ≈100%



CPU History

🟥 CPU1 8.0%   🟧 CPU2 5.1%

Memory and Swap History

Memory
3.0 GiB (79.1%) of 3.8 GiB
Cache 1.1 GiB

Swap
445.7 MiB (48.3%) of 923.3 MiB

Network History

Receiving 12.8 KiB/s
Total Received 321.6 MiB

Sending 151 bytes/s
Total Sent 11.4 MiB

@copyrights Kevin A. Wortman

Screencast: Infinite Loop in System Monitor