12. Counter-Controlled for, Infinite Loops, Jump Statements

CPSC 120: Introduction to Programming
Kevin A. Wortman ~ CSU Fullerton

Agenda

0. Announce
    a. Sign-in sheet
    b. Reminder: Notes Check on Jump Statements
1. Technical Q&A
2. Counter-Controlled `for` Loop
3. Infinite Loops
4. Jump Statements

1. Q&A

Q&A

Let’s hear your questions about...

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

Reminder: write these questions in your notebook during lab

2. 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;
}

```
// 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

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

```
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...
}

```
// 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...
}

```
// 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,

  $i \leq n - 1$

  is the same as

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

```
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 <u>Some</u> 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...
}

```
// 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)

3. 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

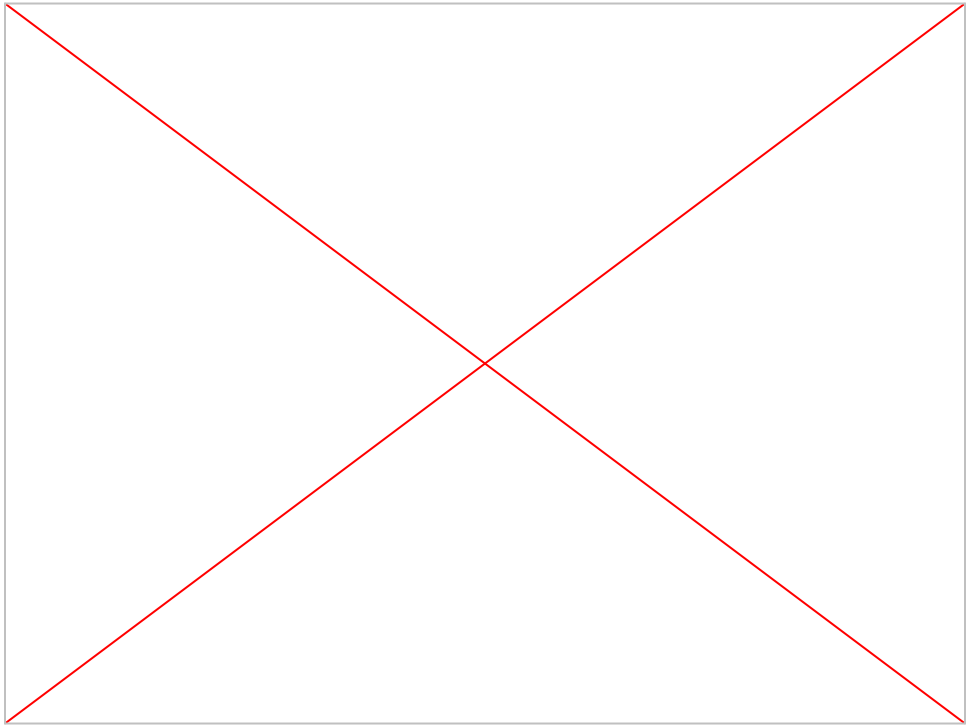
Output (of second loop):

```
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



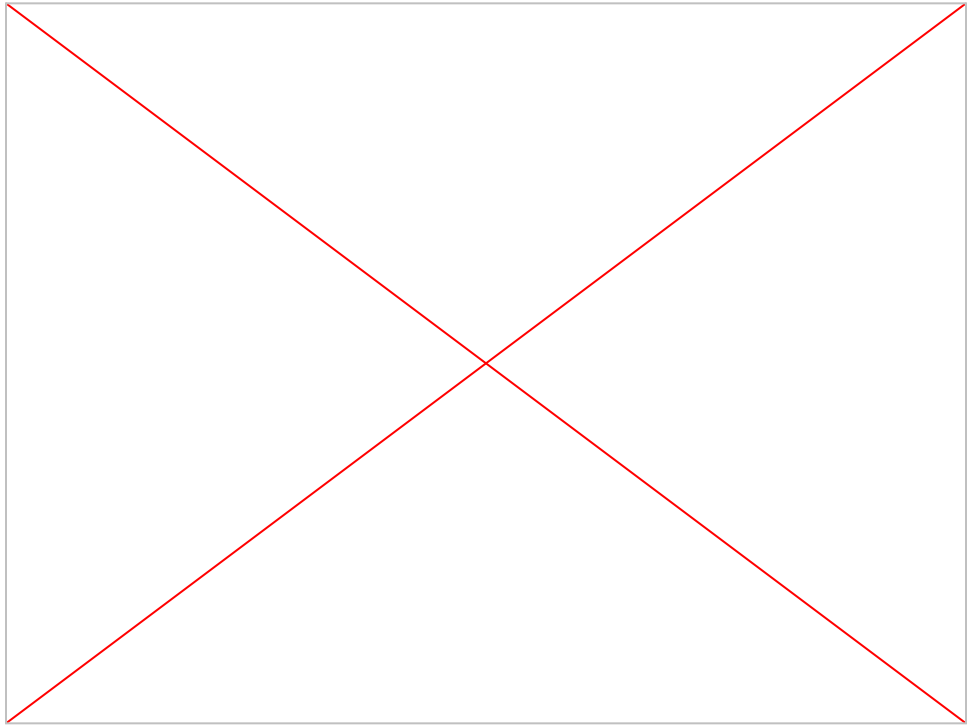
Pitfall: Counter-Controlled Loop Doesn't Advance

- 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

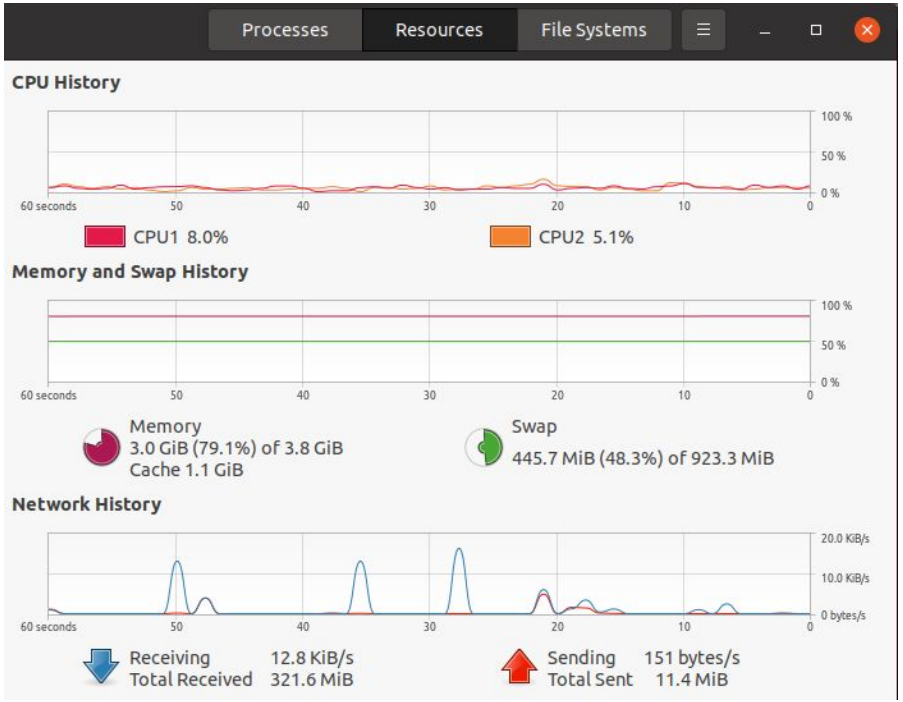


Symptoms of Infinite Loop

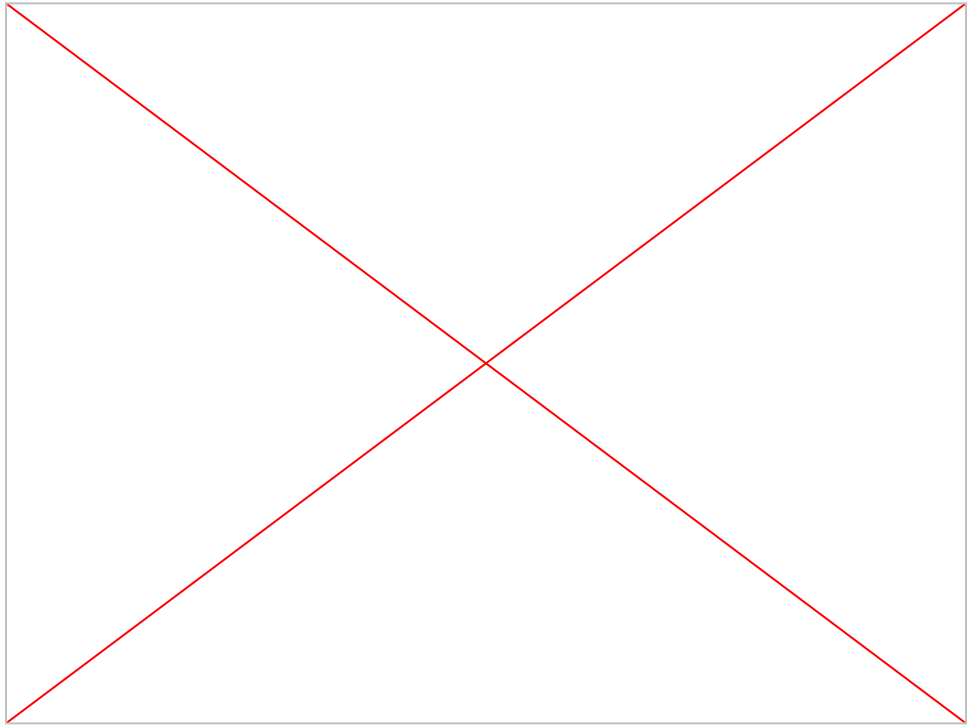
- CPU “**spins**” around the loop as fast as it can
- Program output is either
  - (with output in loop body): never-ending stream of output
  - (without output in loop body): no output, program “**hangs**” (gets stuck)
- Once CPU core spends ≈100% time on your program
- Cooling fan at full speed

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%



Screencast: Infinite Loop in System Monitor



4. Jump Statements

Jump Statements

- **Jump**: immediate move execution flow somewhere else
- Skips over part of the program
- Jumps make tracing code harder
- Peter Parker Principle: "**With great power comes great responsibility.**"
  - *Structured programming* adherents say to never use jumps
- Best practice: **only simple, short jumps**
- `break`, `continue`: adjust the flow of a loop
  - **Acceptable if you keep it simple**
- `goto`: jump from anywhere to anywhere else
  - Not justifiable
  - **Never use goto**

Review: return statement

statement:

return *expression*(optional)**;**

Semantics:

- **Stop** executing the current function
- Use *expression* as **return value**
- *expression* is
  - omitted for `void` functions
  - required for non-`void`
  - mismatch is compile error

break statement

statement:

break;

Semantics:

- Must be inside a loop
  - Or inside a switch, which we are not covering
- **Stop the loop and immediately jump past the end of the loop** (“break”)

```
std::vector<std::string> arguments{argv, argv + argc};
// determine if any argument is "--quiet"
bool is_quiet{false};
for (std::string argument : arguments) {
  if (argument == "--quiet") {
      is_quiet = true;
      break;
  }
}
if (is_quiet) {
  std::cout << "quiet enabled\n";
} else {
  std::cout << "quiet disabled\n";
}
```

```
$ ./a.out fish --quiet cat bird
quiet enabled
$ ./a.out snake dog worm
quiet disabled
```

continue statement

statement:

continue;

Semantics:

- Must be inside a loop
- **Skip over the rest of the current iteration of the loop**
- Keep iterating ("continue")

```
double sum{0.0};
bool first{true};
for (std::string argument : arguments) {
  if (first) { // skip first element
      first = false;
      continue;
  }
  sum += std::stod(argument);
}
std::cout << "sum is " << sum << "\n";
```

```
$ ./a.out 12.5 7 1.1
sum is 20.6
```

return Inside Loop

- `return` semantics: **stop** executing the current function
- Automatically stops any loops
- **`return`** always immediately stops the entire function (main)

```
// validate every argument is positive
bool first{true};
for (std::string argument : arguments) {
  if (first) {
      first = false;
      continue;
  }
  int as_int{std::stoi(argument)};
  if (as_int <= 0) {
      std::cout << "error: all arguments must be positive\n";
      return 1;
  }
}
```

immediately stops all of `main`

Summary of Jump Statements

Jump Statement	Syntax	Stops	Example Uses
return	**return** *expression*(optional)**;**	entire function (inside `main`, that is the entire program)	• stop `main` due to error • stop program early (ex. game won) • define exit code at end of `main`
break	break;	nearest loop	• stop loop when its work is done
continue	continue;	nothing; loop proceeds	• skip an unwanted element in a loop, but keep iterating