11. Loops

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

Agenda

0. Announce
    a. Sign-in sheet
    b. (Collected immediately from now on)
    c. Grader is grading Notes Checks and Discussions
1. Q&A
2. For-Each Loops
3. `while` Loops
4. `do-while` Loops

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. For-Each Loops

Recap: Ideal Division of Labor

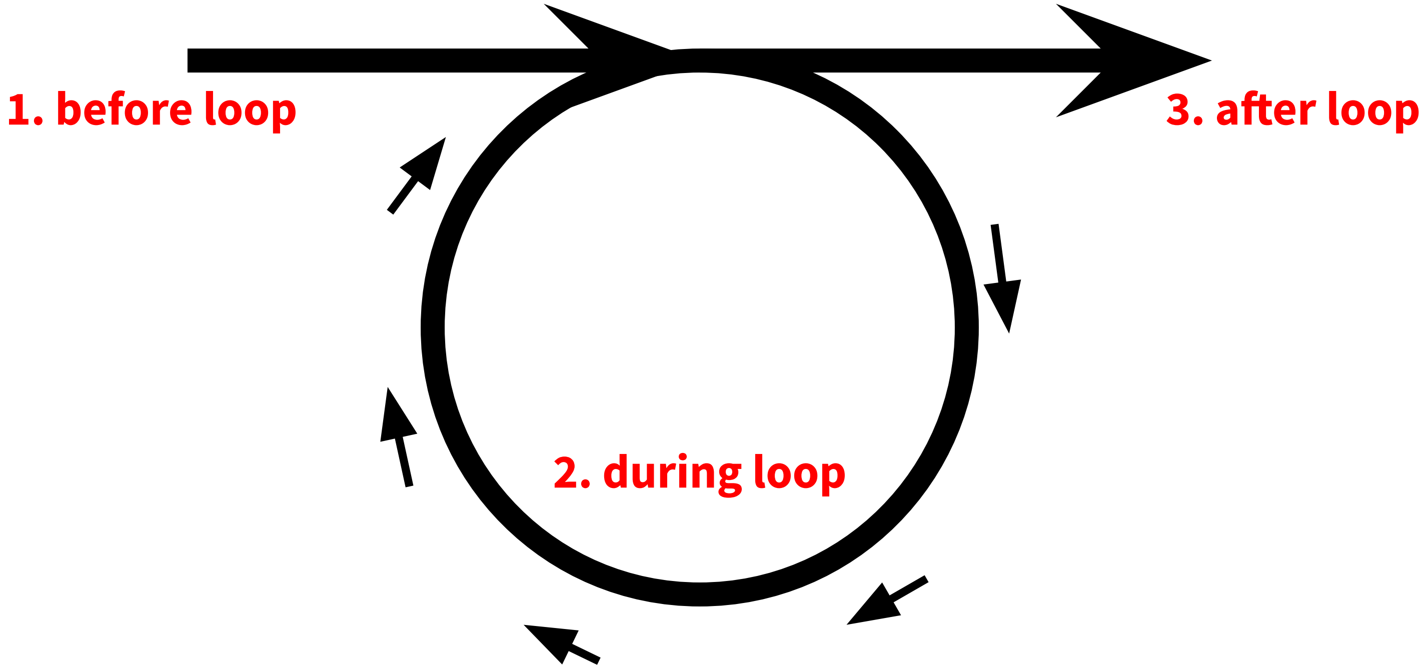
- **Business Logic:** the human meaning of algorithm data
- Programs
  - **Cannot** understand business logic or design algorithms
  - Can perform tedious, repetitive work flawlessly, quickly, cheaply
- Humans
  - **Can** understand business logic and design algorithms
  - Busy-work is tedious, error-prone, expensive
- Division of Labor Best Practice
  - Humans think about business logic and algorithms
  - Computer programs do repetitive work

Loops

- **Loop:** repeat statements to handle multiple things
- Replace manual repetition
  - Writing many emails vs...
  - Algorithm:

```
PostCanvasAnnouncement(roster, message):
    for each student email in roster:
        send message to current email
```

Phases Of A Loop



Loop Terminology

- **Loop** (n): control flow statement that **repeats**
- Loop **body**: statement that is repeated, usually a compound statement
- **Iterate** (v): repeat
- **Iteration** (n): one individual repetition

Syntax: For-Each Loop

statement:

for (*for-range-decl* **:** *container***)**
body-statement

container: expression for a container object

for-range-decl: elt-type elt-identifier

Semantics:

- *elt-type* must match base type of *container*
- for each element in *container*:
  - initialize new *elt-identifier*{ current element }
  - execute *body-statement*
  - *elt-identifier* destroyed

```
// prints -2-7-8-2-0-1-1
std::vector<int> digits{ 2, 7, 8, 2, 0,
  1, 1 };
for (int d : digits) {
   std::cout << "-" << d;
}
std::cout << "\n";

// prints Mon Tue Wed Thu Fri
std::vector<std::string> weekdays{"Mon",
"Tue", "Wed", "Thu", "Fri"};

for (std::string today : weekdays) {
   std::cout << today << " ";
}
std::cout << "\n";
```

Tracing a Loop

For Loop:

```
std::vector<int> area_code{ 6, 5, 7 };
for (int x : area_code) {
   std::cout << x << "~";
}
std::cout << "\n";
```

Equivalent statements:

```
std::vector<int> area_code{ 6, 5, 7 };
{
  int x{ 6 };
  std::cout << x << "~";
}
{
  int x{ 5 };
  std::cout << x << "~";
}
{
  int x{ 7 };
  std::cout << x << "~";
}
std::cout << "\n";
```

Output:

```
6~5~7~
```

Example: Loop Through Command Line Arg's

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

for (std::string s : command) {
   std::cout << "[" << s << "]";
}
std::cout << "\n";
```

```
$ ./a.out one two three
[./a.out][one][two][three]
$ ./a.out
[./a.out]
```

2. `while` Loops

Recap: For-Each Loop

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

```
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
- Now: 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. File I/O

Recap: Filesystem

- Unix organizes storage into a **filesystem**
- A **file** holds data and has a **filename** (e.g. README.txt)
- A **directory** holds files or other directories
    - *Family tree* analogy: the "**parent"** directory holds "**child"** files/directories
- The **root** directory, written / (forward-slash), is the parent of everything else
- A **path** is the location of a file
- **Absolute path**: directions starting from /, with / separating each directory/file name
    - Ex: /usr/share/dict/words
    - The initial / means "start from the root"

File I/O

- **I/O**: Input/Output
- So far: standard I/O
  - `cin`, `cout`
- **File I/O**:
  - `ifstream`: input from a file
  - `ofstream`: output to a file
- Similar to standard I/O
  - `<<`, `>>`
- Output is simpler
  - Less can go wrong
  - Will discuss output first

Uses of File I/O

- INPUT other than command-line arguments, standard input
- Development tools: clang++, make, git
- **Data science**: read dataset with business logic data
- Save/open
  - Program saves information to file
  - Loads file next time it runs

ofstream

- ofstream: **O**utput **F**ile **Stream**
- put data **into** file
- in header `<fstream>`
    - `#include <fstream>`
- ofstream::ofstream (constructor): open file named by string
- ofstream::operator<<: write to file
- Converts to bool
    - `true` == no errors
    - `false` == errors

Example: File Output

```cpp
// save game
int x_coord{1}, y_coord{2}, score{1000};
std::cout << "You are at (" << x_coord << ", " << y_coord
          << "), score=" << score << "\n";
std::ofstream file{"game.dat"};
file << x_coord << " " << y_coord << " " << score << "\n";
if (!file) {
    std::cout << "I/O error writing game.dat\n";
    return 1;
}
```

Standard output:

```
You are at (1, 2), score=1000
```

Contents of game.dat:

```
1 2 1000
```

I/O Errors

- **I/O error**: an I/O operation failed
  - `open`, `<<`, `>>`
- We have seen
  - `cin::>>` fails on invalid input
- Additional reasons for I/O errors with files
  - file not found (wrong name)
  - disk full
  - hardware failure (broken)
- Best practice: **file I/O code must handle I/O errors**
  - if statement to decide whether file object is `true`

ifstream

- ifstream: **I**nput **F**ile **Stream**
- pull data **out of** file
- in header `<fstream>`
  - `#include <fstream>`
- ifstream::ifstream (constructor): open file named by string
- ifstream::operator>>: read from file
- Converts to bool
  - `true` == no errors
  - `false` == errors

Example: File Input

```cpp
// load game
int x_coord{0}, y_coord{0}, score{0};
std::ifstream file{"game.dat"};
file >> x_coord >> y_coord >> score;
if (!file) {
    std::cout << "I/O error reading game.dat\n";
    return 1;
}
std::cout << "You are at (" << x_coord << ", " << y_coord
          << "), score=" << score << "\n";
```

Output when game.dat does not exist:

```
I/O error reading game.dat
```

Contents of game.dat:

```
1 2 1000
```

Output when game.dat exists:

```
You are at (1, 2), score=1000
```

Recap: Current Directory

- **current directory** = location where a program "is"
    - a.k.a. **working directory**
- *State:* current configuration, subject to change
- Keep current directory in mind
    - Unlike search-based apps
- pwd command: **p**rint **w**orking **d**irectory

Program Working Directory

- **program's working directory** = working directory of shell command that started program
  - Rule varies by operating system
  - This is the rule for Unix/Ubuntu
- Working directory is not necessarily the same as where the program is stored
- Example: `git` is in /usr/bin/git, but we run it from other directories
- Could be same, ex. `$ ./a.out`
- Could be different, ex. `$ part-1/a.out`

Pitfall: Wrong Directory

- Runtime error:
    - Input file exists, but program fails to open it
    - Program writes output file, but it doesn’t exist
- Cause: program’s working directory is different than you think
- Review: **program’s working directory** = working directory of shell command that started program
- To debug: make sure you are running program from .
    - (current directory)