13. Designing Loops, Loop Patterns, File I/O

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

Agenda

0. Sign-in sheet
1. Q&A
2. Designing Loops
3. Loop Patterns
4. File I/O

1. Q&A

Q&A

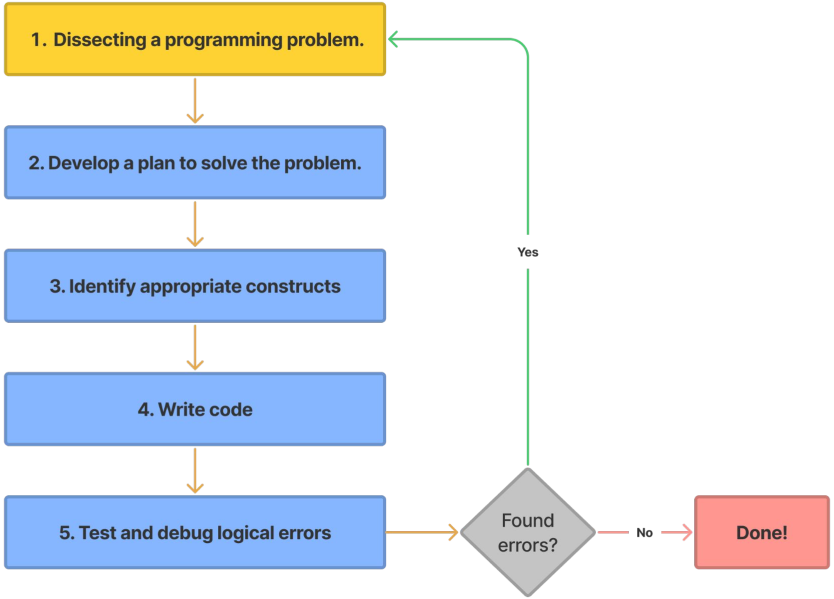
Let’s hear your questions about...

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

Reminder: write these questions in your notebook during lab

2. Designing Loops

Steps for Solving a Programming Problem



1. Dissecting a programming problem.

2. Develop a plan to solve the problem.

3. Identify appropriate constructs

4. Write code

5. Test and debug logical errors

Found errors?

Done!

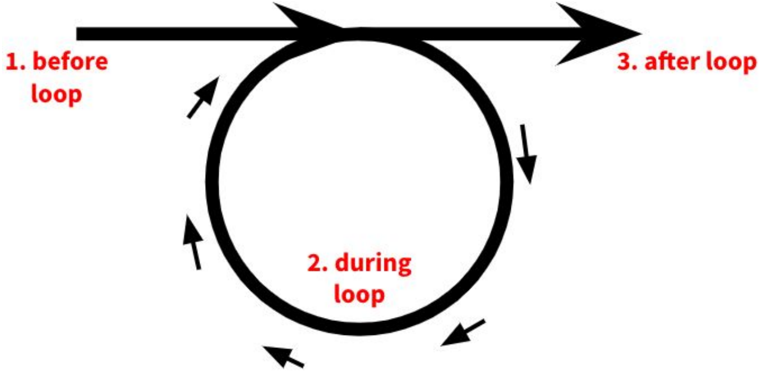
1. Dissect the Problem

- **Understand the problem:** read three times, take notes
- **Identify inputs:** what will the program iterate through?
- **Identify outputs:** what should the program do to each element?
- **Identify test cases:** what happens in...
  a. ordinary container
  b. container is empty
  c. container only has one element

2. Develop a Plan

- **What** are we looping through? container or range of integers?
- **Before:** what statements happen once, before the loop iterates?
- **During:** what statements happen to each element in the loop?
- **After:** what statements happen once, after the loop finishes?

```
before-statements
for ( for-range-decl : container ) {
        each-element-statements
}
after-statements
```



Before, During, After

- Need to plan statements that happen **before / during / after** loop
- **Work backwards**
  - What happens **after** the loop finishes?
  - What needs to happen **during** the loop to be ready for that?
  - What needs to happen **before** the loop to be ready for that?
- Example: count how many students have lab on Monday
  - After? say the number
  - During? decide if a student has lab Monday; if so increase the count
  - Before? tell students with Monday lab to raise hands; start a count at zero

3. Identify Appropriate Constructs

- **Kind of loop:** for, while, do-while
- New **variables** to control the loop?
- **if statement(s)** in the body of the loop?

`if` Inside a Loop

- Recall: any kind of *statement* can go inside a loop body
- Applies to `if` statements
- **Purpose:** make a decision for **each** element
- Examples
  - Handle **first** element differently
  - **Skip** unwanted elements

```cpp
std::vector<double> scores{ 91.0, 102.5,
  86.0, 110.0, 58.5, 102.0 };
std::cout << "Scores with extra credit:";
for (double score : scores) {
  if (score > 100.0) {
    std::cout << " " << score;
  }
}
std::cout << "\n";
```

Output:

```
Scores with extra credit: 102.5 110 102
```

Loop Control Variables

- **Loop control variable:** variable intended to manage the loop
- No special syntax or semantics
- Just a variable we choose to use that way
- Examples:
  - `int`: how many times have we iterated?
  - `bool`: is this the first iteration?

```
std::vector<double> scores{ 91.0, 102.5,
  86.0, 110.0, 58.5, 102.0 };
std::cout << "Scores: ";
bool needs_comma{ false };
for (double score : scores) {
  if (needs_comma) {
    std::cout << ", ";
  }
  std::cout << score;
  needs_comma = true;
}
std::cout << "\n";
```

Output:

```
Scores: 91, 102.5, 86, 110, 58.5, 102
```

4. Write Code

- Fill in the blanks

```
before-statements
for ( for-range-decl : container ) {
        each-element-statements
}
after-statements
```

5. Test and Debug Errors

- As usual, test your program
- Debug
  - Compile errors
  - Logic errors
  - Runtime errors

3. Loop Patterns

Loop Pattern: Accumulate

Accumulate: combine all elements

- Add, multiply, append, ...

```
result-type result { default-result };
for ( element-type element : container ) {
 combine-element-with-result-statement
}
use-result-statement
```

```cpp
std::vector<double> scores{ 91.0, 102.5,
  86.0, 110.0, 58.5, 102.0 };

 // accumulate sum of scores
double sum{ 0.0 };
for (double score : scores) {
  sum += score;
}
std::cout << "Total: " << sum << "\n";
```

Output:

```
Total: 550
```

Loop Pattern: Filter with `if`

Filter: skip unwanted elements

```
for ( element-type element : container ) {
 if ( element-is-wanted-expression ) {
  use-element-statement
 }
}
```

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

for (std::string argument : arguments) {
  if (argument.size() > 1) {
    std::cout << "[" << argument << "]";
  }
}
std::cout << "\n";
```

```
$ ./a.out a b cat d eagle frog
[./a.out][cat][eagle][frog]
```

Loop Pattern: Filter with `continue`

Filter: skip unwanted elements

```
for ( element-type element : container ) {
  if ( element-is-unwanted-expression ) {
    continue;
  }
  use-element-statement...
}
```

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

for (std::string argument : arguments) {
  if (argument.size() < 2) {
    continue;
  }
  std::cout << "[" << argument << "]";
}
std::cout << "\n";
```

```
$ ./a.out a b cat d eagle frog
[./a.out][cat][eagle][frog]
```

Loop Pattern: Count

Count: tally wanted elements

- Hybrid of accumulation and filter
- Counter variable starts at zero
- If an element is wanted, increment counter

```
int counter { 0 };
for ( element-type element : container ) {
  if ( element-is-wanted-expression ) {
    ++counter;
  }
}
use-counter-statement
```

```
int passing_count{ 0 };
for (double score : scores) {
  if (score >= 60.0) {
    ++passing_count;
  }
}
std::cout << passing_count
          << " students passed\n";
```

Loop Pattern: Skip First with `if/else`

Skip first element:

- Filter out first element entirely
- Ex. skip ./a.out in arguments

```
bool first { true };
for ( element-type element : container ) {
 if ( first ) {
  first = false;
 } else {
  handle-subsequent-element-statement...
 }
}
```

```cpp
int total{ 0 };
bool first{ true };
for (std::string argument : arguments) {
 if (first) {
   first = false;
 } else {
   int number{ std::stoi(argument) };
   total += number;
 }
}
std::cout << "Total = " << total << std::endl;
```

```
$./a.out 5 12 -1 2
Total = 18
```

Loop Pattern: Skip First with `continue`

Skip first element:

- Filter out first element entirely
- Ex. skip ./a.out in arguments

```
bool first { true };
for ( element-type element : container ) {
 if ( first ) {
  first = false;
  continue;
 }
 handle-subsequent-element-statement...
}
```

```cpp
int total{0};
bool first{true};
for (std::string argument : arguments) {
 if (first) {
   first = false;
   continue;
 }
 int number{std::stoi(argument)};
 total += number;
}
std::cout << "Total = " << total << std::endl;
```

```
$./a.out 5 12 -1 2
Total = 18
```

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

```
// 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
- Recap: **program’s working directory** = working directory of shell command that started program
- To debug: make sure you are running program from .
  - (current directory)