17. Jump Statements, Designing Loops, Loop Patterns

CPSC 120: Introduction to Programming
Pratishtha Soni ~ CSU Fullerton

1. Technical Q&A

Agenda

0. Announce
   a. Sign-in sheet
   b. M10: Notes Check 2: Sun Oct 29
   c. Midterm 2: Wed Nov 1 (week 11)
1. Technical Q&A
2. Jump Statements
3. Designing Loops
4. Loop Patterns

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. 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)

```cpp
// 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;
  }
}
```

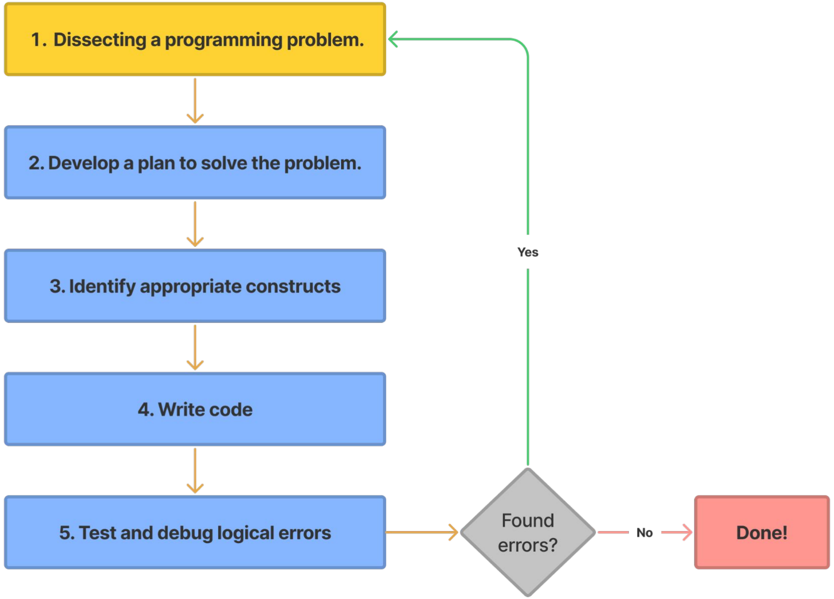
immediately stops all of `main`

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

3. 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?

Yes

No

Done!

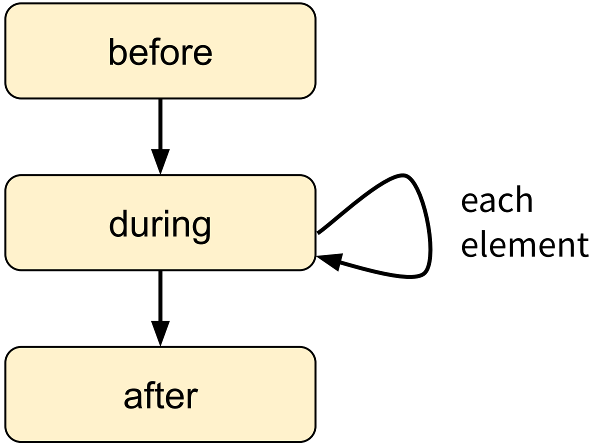
1. Dissect the Problem

- **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

- **Container:** which container object holds the elements?
- **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-each, while, do-while, counter-controlled for
- New **variables** to control the loop?
- **if statement(s)** in the body of the loop?
- **Multiple** loops
  a. One and then another?
  b. (soon) **Nested loop**?
- **Jump statements** (break, continue, return)?

`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?

```cpp
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

4. 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
  }
}
```

```
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
```

```cpp
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

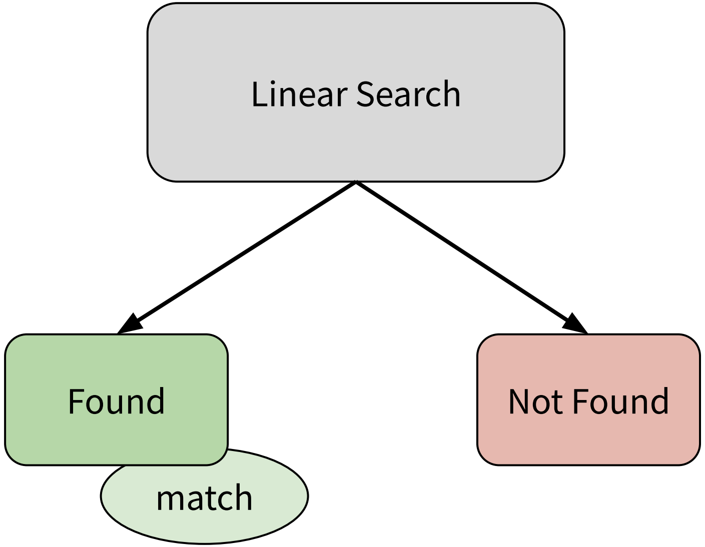
```
bool first { true };
for ( element-type element : container ) {
 if ( first ) {
  first = false;
  continue;
 }
 handle-subsequent-element-statement...
}
```

```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
```

Linear Search

- Linear search: algorithm for finding an element, which may not exist
  - Brute force password cracking
  - Ray tracing computer animation
- Check each element in order
- If the current one is what we want, **stop** (**success/found**)
  - Stop with `break`
- Get to the end: **failure/not-found**
  - Match does not exist
- Two outcomes



Pattern: Linear Search

- `found`: `bool` variable remembers success/failure
- `match`: copy of matching element
    - only valid when `found` is `true`

```
bool found{false};
elt-type match{default-value};
for (elt-type element : container) {
 if (elt-is-match-condition) {
  found = true;
  match = element;
  break;
 }
}
// use found and match
```

```
std::vector<int> values{5, 11, -2, 8};

// find a negative value
bool found{false};
int match{0};
for (int value : values) {
  if (value < 0) {
      found = true;
      match = value;
      break;
  }
}
if (found) {
  std::cout << match << " is a negative value\n";
} else {
  std::cout << "there are no negative values\n";
}
```

What is the Logic Error?

```
bool found{false};
int match{0};
for (int value : values) {
  if (value < 0) {
      found = true;
      match = value;
      break;
  } else {
      break;
  }
}
```

`front` function

- std::vector::front
- simpler, more readable than `at(0)`

Loop Pattern: Optimize

Optimize: find the “most” element

- best/worst/minimum/maximum
- “Champion”: most so far
- Variable for reigning champion
- First element is champion by default
- Subsequent elements compare to champion
    - *Jeopardy* game show

```
element-type champion{container.front()};
for ( element-type element : container ) {
 if ( element-more-than-champion ) {
  champion = element;
 }
}
```

```
double top_score{scores.front()};
for (double score : scores) {
 if (score > top_score) {
   top_score = score;
 }
}
std::cout << "Top score: "
          << top_score << "\n";
```

Output:

```
Top score: 110
```