06. Tracing, Functions, Multiple Source Files

CPSC 120: Introduction to Programming
Pratishtha Soni ~ CSU Fullerton

Agenda

0. Sign-in sheet
1. Technical Q&A
2. Tracing
3. Using Functions
4. Defining Functions
5. Multiple Source Files

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. Tracing

Tracing

- **Trace** (v)
  - Imagine how CPU runs source code
  - Sequential order (top-to-bottom)
  - Write down contents of variables
  - Update variables
- Essential skill
  - Visualize semantics
  - Debugging
  - (later) algorithm design

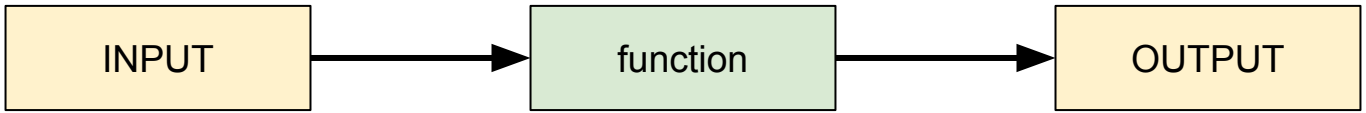
Example: Tracing

```
1
2   #include <iostream>
3
4   int main(int argc, char* argv[]) {
5       int this_year{ 2022 },
6           birth_year{ 1956 },
7           age{ this_year - birth_year };
8       std::cout << "Age is " << age << "\n";
9       return 0;
10  }
```

3. Using Functions

Function Concept

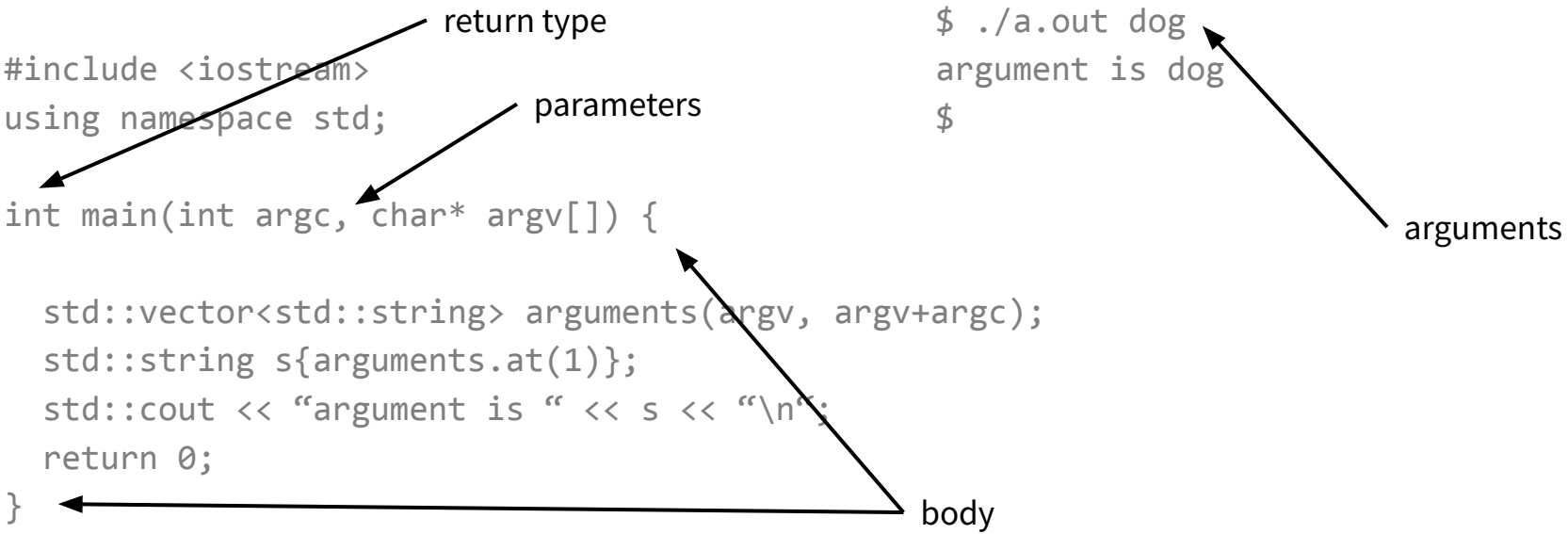
- *Function:* sub-program with its own INPUT and OUTPUT
- Tool to break programs into manageable chunks
- Inspired by math functions like $f(x) = x+1$
- `main` is a function with special status
  - Where program starts
  - Command line arguments become INPUT to main
  - return value of main is program exit code



Function Terminology

- **Call** a function: execute it to transform INPUT into OUTPUT
- **Caller:** code that calls a function
- **Callee:** function that is called
- **Return value:** object that is OUTPUT of function
- **Return type:** data type of return value
  - Either a data type (`int`) or `void` (nothing)
- **Parameter(s):** INPUT data types and names
- **Arguments:** concrete objects used as parameters
- **Body:** block containing statements to execute when function is called

Example: main




```
#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {

  std::vector<std::string> arguments(argv, argv+argc);
  std::string s{arguments.at(1)};
  std::cout << “argument is “ << s << “\n“;
  return 0;
}
```

```
$ ./a.out dog
argument is dog
$
```

Function Prototype

- *Function prototype:* notation for function return type, name, parameters

type name(*parameter*…)

Each *parameter* is a *type identifier* pair

- Example: `stoi`

```
int stoi(const std::string& str)
```

- (full prototype includes optional arguments)

Syntax: Function Call Expression

expression:

function **(** *argument-expr...* **)**

Semantics:

1. Evaluate *argument-expr*... and copy objects into parameter variables
2. Types must match, or compile error
3. Execute function body
4. Return value becomes expression value

Example:

```
int x { std::stoi(“1234”) };
```

Void Functions

- `void`: return type is “nothing”
- Purpose of void function is side effects
- No return value
- Can’t be used in an expression that needs a value

Functions Can Be Called Multiple Times

- Misconception: can only call a function once
- Reality: Can call a function an unlimited number of times
- Encouraged
- **Reuse**: write code once and use it multiple times

Example:

```
int year{ std::stoi(“2022”) };
int month{ std::stoi(“09”) };
int day{ std::stoi(“19”) };
```

4. Defining Functions

Reasons for Defining Functions

- **Organization:** separate code by topic
  - instead of all inside `main`
- **Reuse:** define statements once, instead of copy-pasting
- **Testing:** can test one function at at time
  - instead of entire program
  - **unit testing**

Single Point Of Truth (SPOT)

- **Single Point Of Truth (SPOT):** an idea is represented in **only one place**
  - aka **Don't Repeat Yourself (DRY)**
- General principle
- In programming:
  - define a "magic number" **once** in a **constant variable**
  - define an algorithm **once** in a **function**
- **Ideal Division of Labor** principle:
  - **humans** should not copy-paste the same idea
  - tedious, error-prone
  - **computer** should do that by looking at the **SPOT**

Review: Pattern of a Source File

source-file:

directive, declaration, or definition...

Semantics:

- The compiler processes each *directive, declaration,* or *definition* in top-to-bottom order.

Review: Function Prototype

- *Function prototype:* notation for function return type, name, parameters

type name(*parameter*…)

Each *parameter* is a *type identifier* pair

- Example: `stoi`

```
int stoi(const std::string& str)
```

- (full prototype includes optional arguments)

Function Definition

definition:

prototype {

body-statement...

}

Semantics:

- The function is defined and can be called later in the source file

```
#include <iostream>
...
std::string prompt_string(std::string query) {
 std::string result;
 std::cout << "Enter " << query << ": ";
 std::cin >> result;
 return result;
}
...
int main(int argc, char* argv[]) {
...
```

Pass By Value

When a function is called:

1. A variable is created for each parameter
2. Arguments are evaluated and initialize the parameter variables
3. Jump to start of function body
4. (eventually) Return statement jumps back

```
...
std::string prompt_string(std::string query) {
 std::string result;
 std::cout << "Enter " << query << ": ";
 std::cin >> result;
 return result;
}
...
int main(int argc, char* argv[]) {
 std::string p{prompt_string("protein")};
 std::string b{prompt_string("bread")};
 std::string c{prompt_string("condiment")};
...
```

Tracing Pass By Value

1. `prompt_string("protein")`
   a. query=”protein”
   b. execute prompt_string body
   c. return to main
2. `prompt_string("bread")`
   a. query=”bread”
   b. execute prompt_string body
   c. return to main
3. `prompt_string("condiment")`
   a. query=”condiment”
   b. execute prompt_string body
   c. return to main

```
...
std::string prompt_string(std::string query) {
 std::string result;
 std::cout << "Enter " << query << ": ";
 std::cin >> result;
 return result;
}
...
int main(int argc, char* argv[]) {
 std::string p{prompt_string("protein")};
 std::string b{prompt_string("bread")};
 std::string c{prompt_string("condiment")};
...
```

Output:

```
Enter protein: ham
Enter bread: wheat
Enter condiment: mustard
```

Scope of a Variable

- **Scope** of variable: part of source code where variable is visible
- Compile-time property of source code
- Scope start = point of declaration
- Scope end = } that ends block where variable is declared
- Variable is “**in scope**” or “**out of scope**”
- Referring to a variable that is out of scope is a compile error

Example Scope: p in main

```
int main(int argc, char* argv[]) {

 std::string p{prompt_string("protein")};
 std::string b{prompt_string("bread")};
 std::string c{prompt_string("condiment")};

 print_order(p, b, c);

 return 0;
}
```

Example Scope: b in main

```
int main(int argc, char* argv[]) {

 std::string p{prompt_string("protein")};
 std::string b{prompt_string("bread")};
 std::string c{prompt_string("condiment")};

 print_order(p, b, c);

 return 0;
}
```

Example Scope: c in main

```
int main(int argc, char* argv[]) {

 std::string p{prompt_string("protein")};
 std::string b{prompt_string("bread")};
 std::string c{prompt_string("condiment")};

 print_order(p, b, c);

 return 0;
}
```

Example Scope: `result` in `prompt_string`

```
std::string prompt_string(std::string query) {
 std::string result;
 std::cout << "Enter " << query << ": ";
 std::cin >> result;
 return result;
}
```

Example Scope: query in prompt_string

```
std::string prompt_string(std::string query) {
    std::string result;
    std::cout << "Enter " << query << ": ";
    std::cin >> result;
    return result;
}
```

Whole Program

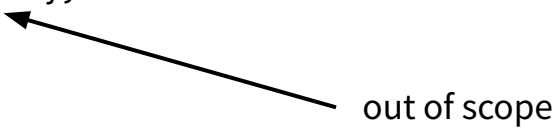
```
#include <iostream>
#include <string>
#include <vector>

std::string prompt_string(std::string query) {
 std::string result;
 std::cout << "Enter " << query << ": ";
 std::cin >> result;
 return result;
}

void print_order(std::string protein,
                 std::string bread,
                 std::string condiment) {
 std::cout << "A " << protein << " sandwich on "
           << bread << " with "
           << condiment << ".\n";
}
```

```
int main(int argc, char* argv[]) {
 std::string p{prompt_string("protein")};
 std::string b{prompt_string("bread")};
 std::string c{prompt_string("condiment")};

 print_order(p, b, c);

 return 0;
}
```

Pitfall: Refer to Out-of-Scope Variables

- Referring to an out-of-scope variable is a **compile error**
- Function can only access
  - parameters
  - variables declared inside function body
- Caller can only access return value of functions

```
...
std::string prompt_string(std::string query) {
 std::string result;
 std::cout << "Enter " << query << ": ";
 std::cin >> result;
 return result;
}
...
int main(int argc, char* argv[]) {
 prompt_string("protein");
 std::string p{result};
...
```



Defining Void Functions

- “Void” = “nothingness”
- **Void function:** has no return value
- So no return type
- Purpose of a void function is a **side effect**, instead of returning an object
  - Ex. printing
  - Ex. changing an existing object
  - Ex. cin::clear

```
void print_order(std::string protein,
                 std::string bread,
                 std::string condiment) {
 std::cout << "A " << protein
           << " sandwich on "
           << bread << " with "
           << condiment << ".\n";
}
```

Function Calls May Be Arguments

- Each argument must be filled in with an **expression**
- A function call is an expression
- So a function call can be used as an argument
- Ex. simplifies our `main`

```
int main(int argc, char* argv[]) {
 print_order(prompt_string("protein"),
             prompt_string("bread"),
             prompt_string("condiment"));

 return 0;
}
```

Functions May Be Called Multiple Times

- Functions may be called multiple times
- **Reuse**: define statements once, instead of copy-pasting
- One of the reasons for functions

```
int main(int argc, char* argv[]) {
 print_order(prompt_string("protein"),
             prompt_string("bread"),
             prompt_string("condiment"));

 return 0;
}
```

Review: Pattern of a Source File

source-file:

directive, declaration, or definition...

Semantics:

- The compiler processes each *directive, declaration,* or *definition* in top-to-bottom order.

Pitfall: Call before Definition/Declaration

- Syntax rule: a function can only be called **after** it has been **defined** or **declared**
- Compile error if out-of-order

```
error: use of undeclared identifier 'prompt_string'
```

- Two solutions:
  - move definition before all calls
  - forward declaration
- Reason why the definition of `prompt_string` and `print_order` are before `main`

OK: Function Defined Before Called

```
#include <iostream>
#include <string>
#include <vector>

std::string prompt_string(std::string query) {
 std::string result;
 std::cout << "Enter " << query << ": ";
 std::cin >> result;
 return result;
}

void print_order(std::string protein,
                 std::string bread,
                 std::string condiment) {
 std::cout << "A " << protein << " sandwich on "
           << bread << " with "
           << condiment << ".\n";
}
```

```
int main(int argc, char* argv[]) {
 std::string p{prompt_string("protein")};
 std::string b{prompt_string("bread")};
 std::string c{prompt_string("condiment")};

 print_order(p, b, c);

 return 0;
}
```

5. Multiple Source Files

Multiple Source Files

- **Problem:** a single .cc file is only practical for tiny programs/teams
  - See https://github.com/chromium/chromium/tree/main/chrome/renderer
- >1000 lines is difficult to organize, understand
- Merge conflicts come from two programmers editing the same file
  - One file means **every** commit is a merge conflict
- Cannot share code between multiple programs
- **Solution:** divide source code into **multiple** source files

.h/.cc Files

- C++ **module:** paired .h, .cc file
  - Code for one specific topic/feature
  - Same base filename
- .h file: **declarations**, documentation, and include directives
  - Audience is **user** of module
- .cc file: **definitions**
  - Workspace for **creator** of module
- Example:
  - rectangle_area_functions.h: declarations and documentation for functions about rectangles
  - rectangle_area_functions.cc: definitions for the functions
- See https://google.github.io/styleguide/cppguide.html#Header_Files

Review: Declaration and Definition

- Recall: function must be **declared** or **defined** before it can be called
- **Function declaration:**
  - Prototype but **no body**
  - Tells compiler that a function **exists**
  - Must eventually be defined, or else compile error
- **Function definition:**
  - Prototype **and body**
  - Gives declaration and body at once

What `#include` Actually Does

- See https://en.cppreference.com/w/cpp/preprocessor/include
- *“Includes other source file into current source file at the line immediately after the directive.”*
- Copy-pastes another source file
- Example
  - `#include <iostream>`
  - Copy-pastes a source file named iostream
  - See https://github.com/llvm/llvm-project/blob/main/libcxx/include/iostream
- All those declarations and definitions are copied into your .cc file

Why Two Separate Files?

- **Goal:** share functions with other programmers/programs
- **Idea** (that doesn't work): function definitions go in a .h file
  - Compiling slows down
  - Recall: `#include` copy-pastes an entire header
  - **Multiple definitions:** function definitions can get copy-pasted multiple times
- Want the .h file to be minimal

Organizing .h/.cc Files

- .h file has:
  - Include guard (later)
  - Declarations
  - Comments that explain how to use
- .cc file has:
  - One definition for each function

Example: read_csv.h

```
#ifndef READ_CSV_H_
#define READ_CSV_H_

#include <string>
#include <vector>

// Read CSV data from filename.
// Returns a 2D vector of strings whose width is columns.
// On I/O error, returns an empty vector.
std::vector<std::vector<std::string>> ReadCSV(
    const std::string& filename,
    int columns);

#endif // READ_CSV_H_
```

Example: read_csv.cc

```cpp
#include <fstream>

#include "read_csv.h"

std::vector<std::vector<std::string>> ReadCSV(
   const std::string& filename,
   int columns) {
 std::vector<std::vector<std::string>> table;
 std::ifstream file(filename);

 // read each row
 while (file.good()) {
   std::vector<std::string> row;
   // read each column
   for (int i = 0; i < columns; ++i) {
     std::string cell;
     file.ignore(1, '"'); // leading quote
     std::getline(file, cell, '"');
     file.ignore(1, ','); // comma
     row.push_back(cell);
   }
   if (file.good()) {
     table.push_back(row);
   }
 }

 return table;
}
```

Header Include Guards

- See https://google.github.io/styleguide/cppguide.html#The__define_Guard
- A .h file should
  - **start** with boilerplate `#ifndef/#define` preprocessor directives
  - **end** with #endif preprocessor directive
- Purpose:
  - Header contents are only included the first time the file is `#included`
  - Subsequent times have no effect
  - Speeds up compilation
  - Prevents certain compile errors

Syntax: Header Include Guard

directive at start of .h file:

#ifndef *HEADER_FILE_NAME_H_*
#define *HEADER_FILE_NAME_H_*

directive at end of .h file:

#endif // *HEADER_FILE_NAME_H_*

Semantics:

- The first time the .h file is included, the identifier *HEADER_FILE_NAME_H_* is defined and contents of file are copy-pasted
- Subsequent times, *HEADER_FILE_NAME_H_* is already defined so the .h file is effectively skipped

```
#ifndef READ_CSV_H_
#define READ_CSV_H_

...

#endif // READ_CSV_H_
```

Example: read_csv.h

```
#ifndef READ_CSV_H_
#define READ_CSV_H_

#include <string>
#include <vector>

// Read CSV data from filename.
// Returns a 2D vector of strings whose width is columns.
// On I/O error, returns an empty vector.
std::vector<std::vector<std::string>> ReadCSV(
    const std::string& filename,
    int columns);

#endif // READ_CSV_H_
```