COMP 1633: Intro to CS II

Pass by reference

Charlotte Curtis

January 22, 2024

Where we left off

- Predefined functions in C++
- Function calls
- Declaring and defining functions
- Variable scope
- A week of async!

```
#include <iomanip>
cout.precision(2);
cout << fixed;
cout << "Total: $" << setw(8)
     << bill << endl;
cout << "With GST: $" << setw(8)
     << bill*1.05 << endl;
```

> *Textbook Sections 4.1-4.5*

Today's topics

- More on variable scope
- Pass by reference
- Bottom-up vs top-down testing

> *Textbook Sections 4.5, 5.1-5.5*

Variable scope

- Python only has two scopes: global and local
- C++ has **block-level** scope
  - Variables declared inside `{}` are only accessible inside that block
  - This applies to functions, but also `if` statements, `for` loops, etc.

```
int global_var = 23; // bad idea, but legal
int main() {
      int local_var = 42; // only accessible within main
      cout << global_var << endl; // 23
      cout << local_var << endl; // 42
}
```

Function parameters *are* local variables

- Parameters are local variables that are initialized with the values of the arguments
- Do not redefine function parameters!

  Common error in COMP 1701:

```
def some_function(arg_1: int, arg_2: int) -> int:
    arg_1 = 42
    arg_2 = input("Enter a number: ")
    return arg_1 + arg_2
```

Scope guidelines

- Declare variables in the smallest scope possible
- Variable names *can* be repeated in different scopes, but make sure the usage is consistent
  - e.g. if `temp` is used for temperature in one scope, don't use it for temporary values in another
  - Similarly if `x` is an `int` in one scope, don't use it as a `double` in another
- Avoid global variables altogether (except constants shared across scopes)
  - Improper use of global variables *will* affect your grade on assignments

Functions check-in 1/2

Based on the following function **prototype** (**declaration**), which of the following is a valid function call?

```
double compute_interest(double balance, double rate, int years);
```

A. `int interest = compute_interest(1000, 0.05, 3);`

B. `compute_interest(1000, 0.05, 0.5);`

C. `double interest = compute_interest(1000, 0.05, 3);`

D. `cout << compute_interest() << endl;`

Functions check-in 2/2

Predict the output of the following code:

```
void fun(int x);

int main() {
    int y = 0;
    fun(y);
    cout << y << endl;
}

void fun(int x) {
    x = x + 10;
}
```

Returning multiple things

- Functions allow you to return either **nothing** (`void`) or **one thing** (any other type)
- Python (sort of) allows you to return multiple values from a function by implicitly packing them into a tuple:

```python
def get_initial_and_age() -> tuple[str, int]:
    initial = input("Enter your initial: ")
    age = int(input("Enter your age: "))
    return initial, age
```

- How can we do this in C++ (98)?
    - Objects and data structures, pointers, and **references**

Note: C++ 11 introduced tuples, and they're probably a better way to do things, but learning about pass by reference is still valuable

Pass by value

So far, all of our functions have used **pass by value**

- The *value* of the argument is assigned to the parameter
- Example: given the following function:

```
void increase_salary(double salary, double percent_increase) {
    salary = salary * (1 + percent_increase);
}
```

trace the execution of the following code, showing memory locations:

```
double wage = 10000.0;
increase_salary(wage, 0.05);
cout << wage << endl;
```

Pass by reference

- Instead of passing a value, we can pass a **reference** to a memory location
- This allows us to modify the original value, in a different scope!

> *Use with caution!* ***Side effects*** *can lead to chaos*

- Syntactically, one tiny change: `&` after the type in the parameter list

```
void increase_salary(double& salary, double percent_increase) {
    salary = salary * (1 + percent_increase);
}
```

- The `&` is called the **reference operator**

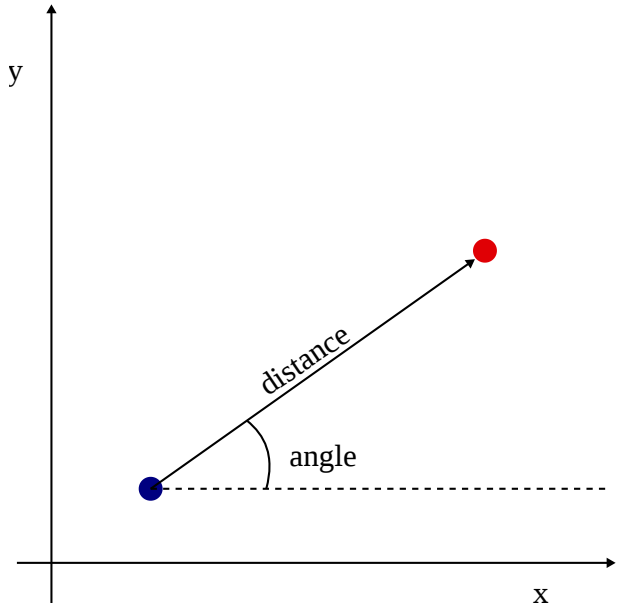
Rules and conventions for pass by reference

- Only things with an **address** can be passed to a reference parameter
  - Variables only, no literals or expressions
- Reference parameters can be both **read** and **written**
  - The called function can modify or even **destroy** the original value!

> *Style note: functions with reference parameters should usually be* `void` *or return* `bool` *(more on that later)*

Example 1

Write a **prototype** for a function that will "move" a point in a 2D plane according to an angle and a distance. Assume the point is represented by two `double` parameters `x` and `y`.



Example 2

Implement a function with the prototype `void swap(int& a, int& b)` that exchanges the values of two `int`s

> *Try this on paper for a few minutes, then we'll go through a solution*

Testing: drivers and stubs

- Functions are great because they let us break our code into smaller pieces
- Don't wait until you've written everything to test!
- Two approaches:
  - Top-down: Start with the main logic and then fill in the pieces
  - Bottom-up: Start with the pieces and then put them together

> *Either way you need to "fake" the parts you haven't written yet*

Testing review 1/2

A function with an `int` parameter `num` implements the following logic. How many test values are needed to exhaustively test it?

A. 1

B. 2

C. 3

D. 4

E. Impossible to test exhaustively

```
result = num
if num < 0
    result = -num

return result
```

Testing review 2/2

It is acceptable to hard-code magic numbers for test purposes.

A. True

B. False

Test Drivers

- Used in **bottom-up** testing
- After writing a complete function, a **test driver** is a "dummy" `main` function that calls the function with a variety of test values
- At its simplest, the test driver should print the results of the function calls along with a label for context

```
cout << "my_func(2) = " << my_func(2) << endl;
cout << "my_func(-2) = " << my_func(-2) << endl;
cout << "my_func(0) = " << my_func(0) << endl;
```

> *You can also use `assert` or a test framework like GoogleTest, but those are beyond the scope of this course*

Function Stubs

- While a driver acts as the *calling* function, a **stub** acts as the *called* function
- Used in **top-down** design
- Write your `main` logic first, then write stubs for the functions you need
- Stubs match the data type, name, and number of parameters for a function you want to write, but don't do anything useful

```
int my_func(int whole_num, double dec) {
    return 0;
}
```

> *What value should the stub return? Something that makes sense in the context of how the function will be used.*

Side Tangent: input redirection

- We talked about testing functions individually with **hard-coded** values, but eventually you need to test with input as well
- You *can* repeatedly type your input...

```
$ ./a1
Enter the range of R0 values (0 - 20): 0.5 12
Enter the range of p values (0 - 1): 0.1 0.95
```

- But it's easier to **redirect** input from a file:

```
$ ./a1 < input.txt
```

> *This is a bash thing, not a C++ thing - you could do the same with Python*

Boolean preview

`bool` is a data type that can only have two values: `true` or `false`

Python	C++	Description
`==`	`==`	Equal to
`!=`	`!=`	Not equal to
`<`	`<`	Less than
`<=`	`<=`	Less than or equal to
`>`	`>`	Greater than
`>=`	`>=`	Greater than or equal to

Functions can return `bool`, just like in Python:

```python
def is_even(num: int) -> bool:
    return num % 2 == 0
```

```cpp
bool is_even(int num) {
    return num % 2 == 0;
}
```

Compound Boolean expressions

Python	C++	Description
`and`	`&&`	Logical and
`or`	`||`	Logical or
`not`	`!`	Logical not

- Same behaviour and precedence as Python, just different symbols
- Example: `bool in_range = x > 0 && x < 10`

`if` statement syntax

```
if (boolean_expression) {
    // code to execute if true
} else {
    // code to execute if false
}
```

- note the `()` around the boolean expression - this is mandatory in C++
- Like Python, the `else` is optional
- More nuance on `if` and booleans next lecture

Coming up next

- Lab: Pass by reference
- Lectures: Decisions and Loops
- Assignment 1 now available! 🎉

> *Textbook Sections 2.4, 3.1-3.2*

Extra: another pass-by-reference example

Write a function called `sort2` that takes two `int` parameters and sorts them in ascending order - that is, after a call to `sort2(m, n)`, the smaller value is in `m` and the larger value is in `n`.

> *Hint: you can use the `swap` function from earlier*