CMPUT 201: Practical Programming Methodology

Guohui Lin
guohui@ualberta.ca
Department of Computing Science
University of Alberta
September 2019

Lecture 12: Functions

Agenda:

- Definition
  - recall the following?

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

- Defining and calling functions
- Function declarations
- Arguments: passed-by-value

Reading:

- Textbook: Chapter 9

Functions:

- A function is
  - a series of statements
  - grouped together and given a name
- They are building blocks
  - **just like the previously seen `main` function**
  - each is a small program, w/declarations and statements
  - not necessarily have arguments, nor necessarily compute
  - purpose of existence:
    - divide the program for easier understanding/modifying
    - avoid duplicating code, or re-use
    - (yet, you do not have to have them!)

Defining and calling functions:

- The `main` function is the starting point of the program

```
int main(void) {
    ...
}

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

- Frequent computation of the average of two values:
  - math: `(a + b) / 2`
  - perhaps don't want to code this, but "refer it to as" `average(a, b)`

Defining and calling functions:

- (Page 184–185)

```
double average(double a, double b) {
    return (a + b) / 2;
}
```

  - each time, `average` returns a value of type `double`
  - parameters `a` and `b` are supplied when `average` is called
  - `a` has type `double`
  - { ... } is the function body
  - calling `average(x, y)`, values of `x` and `y` are **copied** into `a` and `b`, respectively
  - e.g.,

```
printf("Average: %g\n", average(5.1, 8.9));

z = average(x, y);
```

Defining and calling functions:

- ```
  /* Computes pairwise average of three numbers */

  #include <stdio.h>

  double average(double a, double b) {

      return (a + b) / 2;
  }

  int main(void) {

      double x, y, z;

      printf("Enter three numbers: ");
      scanf("%lf%lf%lf", &x, &y, &z);
      printf("Average of %g and %g: %g\n", x, y, average(x, y));
      printf("Average of %g and %g: %g\n", y, z, average(y, z));
      printf("Average of %g and %g: %g\n", z, x, average(z, x));

      return 0;
  }
  ```

  - compiler must know what is `average` before using it
  - so definition of `average` comes before `main`, which "calls" `average`
  ```

Function definitions:

- The general form

```
return-type function-name ( parameters ) {
    declarations
    statements
}
```

  - return a value, cannot be an array
  - do not have to use the return value (recall `printf`? it is a function)

    (can return no value (type `void`))
  - parameters separated by `,`, each must be w/a type
  - parameters must be enclosed by `( ... )`, even empty

    (can have no parameter (`void`))
  - variables declared inside a function belong exclusively to the function
  - body must be enclosed by `{ ... }`, even empty

- Example dividing a C program:
  - primality testing (we did it in Chapter 6)

```
/* Tests whether a number is prime */

#include <stdio.h>

int main(void) {

    int n;
    int d;

    printf("Enter a number: ");
    scanf("%d", &n);

    if (n > 1) {
        for (d = 2; d < n && n % d != 0; d++);
        if (d < n)
            printf("%d is not prime\n", n);
        else
            printf("%d is prime\n", n);
    }
    else
        printf("%d is not prime\n", n);

    return 0;
}
```

```
/* Tests whether a number is prime */

#include <stdio.h>
#include <stdbool.h>

bool is_prime(int n) {

    int d;

    if (n <= 1)
        return false;
    for (d = 2; d < n && n % d != 0; d++);
    if (d < n)
        return false;
    return true;
}

int main(void) {

    int n;

    printf("Enter a number: ");
    scanf("%d", &n);
    if (is_prime(n))
        printf("%d is prime\n", n);
    else
        printf("%d is not prime\n", n);

    return 0;
}
```

Function declarations:

- In the above, definition of `is_prime` is placed above `main`
- Too many such function definitions might make `main` difficult to find
- Such function definitions do not have to precede `main`, but
- Rule: (complete) information of a function must be known before its first call
- Function declaration:

  ```
  return-type function-name ( parameters );
  ```

  - the `;` to declare the (complete) information for a function, before `main`
  - to satisfy the need for its first call
  - the detailed definition in another place
  - known as **function prototypes**

```
/* Tests whether a number is prime */

#include <stdio.h>
#include <stdbool.h>

bool is_prime(int n);

int main(void) {
    int n;

    printf("Enter a number: ");
    scanf("%d", &n);
    if (is_prime(n))
        printf("%d is prime\n", n);
    else
        printf("%d is not prime\n", n);

    return 0;
}

bool is_prime(int n) {
    int d;

    if (n <= 1)
        return false;
    for (d = 2; d < n && n % d != 0; d++);
    if (d < n)
        return false;
    return true;
}
```

Arguments:

- Parameters appear in function definitions

```
double average(double a, double b);
```

- Arguments are expressions appear in function calls

```
z = average(x, y + 2.3);
```

  - **passed by value** — always! always!
  - each argument is evaluated, and its value assigned to the corresponding parameter

    (parameter acts like a variable: `b = y + 2.3;` does not change anything about `y`)
  - the follow call does not change the value of `x`!

```
double average(double a, double b) {
    a = (a + b) / 2;
    return a;
}
    ...
    z = average(x, y + 2.3);
```

More on arguments:

- The type of an argument vs. the type of the corresponding parameter
  - is (automatically, implicitly) converted
- Array arguments

```
int sum_array1(int a[10]);
int sum_array2(int a[], int n);
```

  - `sum_array1` has a parameter — a 1-dimensional array of length 10
  - cares:
    - fixed length, cannot be changed during calls
    - the actual array length must be ≥ 10, only the first 10 elements used
  - `sum_array2` has a parameter — a 1-dimensional array of unknown length

    normally specified by a second parameter — `n` in this case
  - cares:
    - without `n`, difficult to determine the length (can we use `sizeof`?)
    - when called, the length argument must be $\leq$ the actual array length

More on array arguments:

- If a parameter is a multi-dimensional array
  - only the length of the 1st dimension can be "unknown"

    ```
    int sum_array3(int a[][100], int n);
    ```

  - later we will use "**pointers**" to overcome this constraint

- Variable-length array parameters
  - recall variable-length array?

    ```
    int n;
    int a[n];
    ```

  - similarly, we may use the following to explicitly state the array length

    ```
    int sum_array1(int a[10]);
    int sum_array2(int n, int a[n]);
    ```

    (mind the order!)

  - summary: the following are all legal (for the same purpose)

    ```
    int sum_array2(int a[], int n);
    int sum_array2(int n, int a[n]);
    int sum_array2(int n, int a[]);
    int sum_array2(int n, int a[*]);
    ```

  - particularly useful for multi-dimensional array parameters

More on array arguments:

- Use keyword `static` to declare the minimum length

  ```
  int sum_array3(int a[static 10][100], int n);
  ```

  - meaning the length of the 1st dimension is ≥ 10
  - can only be used for 1st dimension though
  - use of `static` has no effect on anything else
  - (compiler uses it for some possible speed-up)

Agenda:

- `return` statement: goes back to where function is called
- `exit()` function in `<stdlib.h>`: terminates program
- Recall statements: `break`, `continue`, `goto`
- Recursion and applications
  - `quick sort`
  - `merge sort`

Reading:

- Textbook: Chapter 9

Functions:

- A function is
  - a series of statements
  - grouped together and given a name
- They are building blocks
  - **just like the previously seen `main` function**
  - each is a small program, w/declarations and statements
  - not necessarily have arguments, nor necessarily compute
  - purpose of existence:
    * divide the program for easier understanding/modifying
    * avoid duplicating code, or re-use
    * (yet, you do not have to have them!)

The `return` statement:

- Form

  ```
  return expression ;
  ```

- e.g.,

  ```
  return;
  ```

  ```
  return 0;
  ```

  ```
  return false;
  ```

  ```
  return -1;
  ```

  ```
  return n >= 0 ? n : 0;
  ```

  - the type of the expression must match the type of the function
  - otherwise implicit conversion happens
  - no expression in `return;` for a `void` function
  - without `return` statement in a non-`void` function could cause errors

- For `int main`, return value is a status code — 0: terminates normally

The `exit` function:

- It's a function, from

  ```
  #include <stdlib.h>
  ```

- e.g.,

  ```
  exit(0); /* normal termination, very the same as 'return 0;' */
  ```

  ```
  exit(EXIT_SUCCESS);
  ```

  ```
  exit(EXIT_FAILURE);
  ```

  - the argument has the same meaning as `main`'s return value
  - `EXIT_SUCCESS` and `EXIT_FAILURE` are macros defined in `<stdlib.h>`
  - `exit` terminates the program (within any function)

Recursion:

- A function that calls itself
- e.g., $n! = n \times (n-1)!$

```
int factorial(int n) {

    if (n <= 1)
        return 1;
    else
        return n * factorial(n-1);
}
```

```
int factorial(int n) {

    return n <= 1 ? 1: n * factorial(n-1);
}
```

  - must have termination condition
  - make sure termination condition **will be met**

Recursion example — Quicksort:

- A comparison-based sorting algorithm
- Goal: sort an array of integers into a non-decreasing order
- The algorithm:
  - assume the array is $a[i..j]$
  - select an element (say $a[i]$) as the pivot
  - rearrange the array such that

  1) all "elements $\leq$ pivot" precede the pivot: $a[i..(k-1)]$
  2) all "elements $>$ pivot" succeed the pivot: $a[(k+1)..j]$
  3) the pivot is in $a[k]$

  - recursively to quicksort $a[i..(k-1)]$ and $a[(k+1)..j]$

Quicksort, continued:

- The code design:
  - `void quicksort(int a[], int left, int right); /* representing array a[i..j] */`
  - inside `quicksort`:
    - need to determine the index for the pivot
    - at the same time partition the array into ["<=", pivot, ">"]
    - `int split(int a[], int left, int right); /* partitioning, getting index */`
    - recursively call `quicksort`
- Program appearance:

```
Enter 10 numbers to be sorted: 1 4 11 100 2 7 3 -1 99 6
In sorted non-decreasing order: -1 1 2 3 4 6 7 11 99 100
```

Lecture 14: Program Organization

Agenda:

- Local variables
  - declared inside the body of a function (**cannot** define a function inside a function!)
  - function parameters "are" local variables
  - `static` for permanent storage
- External (or, global) variables
- Blocks (often, nested)
- Scope — three levels: block, file, program
- Organizing a C program

Reading:

- Textbook: Chapter 10

When multiple functions:

- How shall we organize them?
- Will be many variables
  - where they can be accessed? modified?
  - how do we differentiate them?
  - re-use the variable names such as "`int i, j;`"?

General form of a C program (from Lecture 2):

```
/* directives */

int main(void) {

    /* statements */
}
```

- Typical `directives`,

```
/* headers */
#include <stdio.h>

/* macros */
#define FREEZING_PT 32.0f

/* global variables */
int num_pt = 0;

/* function prototypes */
void mst_prim(int n, int **point);

/* main function */
int main(void) {

    /* statements */
}
```

General form of a C program:

```
/* directives */

int main(void) {

    /* statements */
}
```

- Typical `statements`,

```
/* declarations */
float fahrenheit, celsius;

/* assignments */
celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;

/* function calls */
scanf("%f", &fahrenheit);

mst_prim(num_pt, point);

/* function terminates, and returns a value */
return 0;
```

Local variables:

- Declared in the body of a function
  - local to the function
  - automatic storage duration (life span the same as the function)
  - block scope (visible from its declaration to the end of enclosing body) — smallest
  - the following `int j;` only visible inside the `for`-loop

```
void f(void) {

    int i;
    ...
    for (i = 0; i < n; i++) {
        int j;

        ...
    }

    ...
}
```

Static local variables:

- Declared in the body of a function, using keyword `static`
  - local to the function
  - permanent storage duration (life span the same as the entire program)
  - block scope (visible from its declaration to the end of enclosing body) — smallest
  - the following `static int j;` only visible inside the `for`-loop

Summary: static local variable provides a place to hide data (from other functions) — for future calls of the same function/block !! (demo)

```
void f(void) {

    int i;
    ...
    for (i = 0; i < n; i++) {
        static int j;

        ...
    }

    ...
}
```

Function parameters:

- The same as local variables
  - local to the function
  - automatic storage duration (life span the same as the function)
    - recall the meaning of keyword `static`?
  - block scope (visible from its declaration to the end of enclosing body) — smallest
  - difference: when function is called, parameters are initialized with arguments
    - again, passed by value !
    - make sure what each value is !

External variables:

- Normally, accessed / modified by multiple functions
- In general (safer way), passing information to function by "arguments $\rightarrow$ parameters"
- Declared outside the body of a function
  - global (counterpart: local)
  - permanent storage duration
  - file scope (visible from its declaration to the end of enclosing file)
  - e.g., implementing a stack (Page 221)

Stack:

- A data structure
  - an array-like
  - cannot access an element by index/position
  - operations:
    push — add an element (to stack top)
    pop — remove an element (on the stack top)
  - use a variable `top` to store the number of elements
- Implementation:
  - the stack as an array
  - two operations as two separate functions
  - the array and `top` both external

Pros and cons of external variables:

- Convenient: no worries about using parameters
- Code modification? such as change the data type? — could be a problem
- If an error, difficult to locate
- Hard to re-use the functions
- Unexpected name conflicts

```
int i;

void f(void) {

    int i;

    for (i = 0; i < n; i++) {
        ...
    }
}
```

Blocks:

- We said earlier "block scope"
- General form of a block:

```
{
    declarations
    statements
}
```

  - acts like a function (without a name!)
  - local variables
  - static local variables
  - inside the body, (one level outside) relative "external" variables

- Running out of names? e.g.,

We could use `int i` very frequently:

- e.g. (Page 229),

```
int i;

void f(int i) {
    i = 1;
}

void g(void) {
    int i = 2;

    if (i > 0) {
        int i;

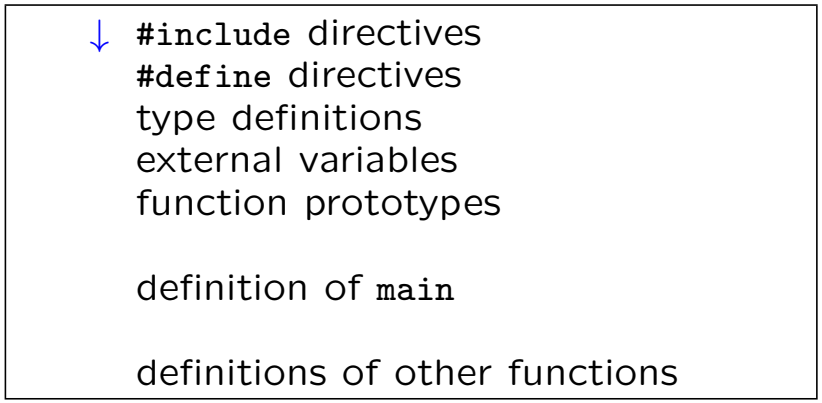
        i = 3;
    }
    i = 4;
}

void h(void) {
    i = 5;
}
```

- When a new `int i` is declared, it "hides" the old meaning
- When a newly declared `int i` expires, it regains the old meaning

Organizing a C program:

- (For now,) fit into a single file (later for multiple files, and `makefile`)
- A few simple, intuitive rules
  - a directive does not take effect until the line
  - a type name cannot be used until it's defined
  - a variable cannot be used until it's declared
  - a function cannot be called until it's declared
- Likely order:



- Classifying a poker hand (Page 230–236)

Instructions (during all exams):

- Read these instructions and wait for the signal to turn this cover-sheet over.
- Use space below/beside the questions to write your solutions legibly.
- Closed book;
  - no electronic devices (make sure your cellphone is OFF),
  - no calculators,
  - no conversations.
- In general, no questions will be answered during the quiz;
  - if unsure, state your best assumptions clearly and proceed;
- We will provide an exam clock.