CMPSC 16
Problem Solving with Computers I
Spring 2014

Instructor: Lucas Bang

Lecture 8 / 9: Functions, File IO
It is time to discuss functions!

• A very important part of programming is the use of functions!

• Functions are a very, very important mechanism in programming

• They allow us to modularize code!
  – Modularization is the key to developing large programs
  – So, you have to use functions if you are going to develop large programs, there is no other way
Modularity by C functions

- Idea: let modules solve problem *parts* – then combine the parts to solve whole problems
  - **Abstraction** benefits – *details are hidden* in a module to reduce complexity of overall solution
  - **Reusability** benefits – maybe can use it many times
  - **Testing** benefits – be confident the parts work

- A function is the simplest type of module
  
  ```c
  printf("Hello");
  ```
  - As a user of the function, we only care about **what** it does, **not how** it does it!
Functions

• To use functions in our programs we need to do two things:
  1. We need to **define** the functions: We need to write the code that defines how the function will execute
  2. We need to **call (invoke)** the functions: We use the functions by calling them

• We actually used/mentioned many functions so far (like printf, rand)
  – We did not have to define these functions because they are library functions that were defined (i.e., written/implemented) by other people. We just used (i.e., called) them!
Calling/Invoking functions

- Syntax:
  
  ```
  functionName(list of arguments);
  ```

- *Invoking* or *calling* a function. Arguments that we give when we invoke a function are called *actual parameters*.

- Effect – transfers control to the function
  - Along with *copies* of data passed as arguments

- When function completes – control returns to the point in the program where it was invoked
  - And returns a result if not a *void* function
Using functions to solve problems

- You might be able to directly translate an algorithm into a series of function calls
  
  ```
  mydata = getData();
  results = process(mydata);
  showResults(results);
  ```

- In turn, the function `process()` might do:
  ```
  intermediateResult = calculate(x, y);
  ```
  where `calculate` is another function, to perform a difficult calculation involving `x` and `y`.

- And so on: “top-down ... stepwise refinement”
Defining functions

• A function has two parts – a header and a body

```
type name (parameter declarations)//*header */
{ local declarations and statements } /*body */
```

– Header parentheses, and curly braces both required

• type – refers to the type of value returned by the function
  – If void – means function does not return any result
  – If not void, the statements in the body must execute a return statement

• name – any valid C identifier (same rules as variable names)

• parameter declarations: These declare the types and names of the formal parameters. This is how we access the values that are passed to the function as input via actual parameters
An example function

```c
int max(int a, int b) {
    if (a > b)
        return a;
    return b;
}
```

- Function header (interface) – says it will take two int values, and it will return an int value
  - i.e., how to use it, and what to expect in return

```c
    printf("%d\n", max(17, 4));
```

- Function body (implementation) – does the work
Function prototypes

• Also known as “forward declarations”
  – Just have to declare a function before it is used – so the compiler can know its interface
  – e.g., int max(int, int);

• Parameter names are optional in prototypes
  – The compiler doesn’t need the names to verify the function is being used correctly

• Prototypes insufficient for execution though
  – Must define function somewhere (linker looks for it)
  – i.e., need implementation – both a header and a body

• For example the stdio.h file we include uses the function prototypes for io functions such as printf, scanf, getchar, etc.
Actual vs. Formal parameters

- In each invocation of a function formal parameters have their own memory allocated for them!

- This fact has important implications

- For example, the following function changes the value of the (formal) parameter:
  ```c
  void foo(int x) {
    x = x * 5;
  }
  ```

- So what would the following code print?
  ```c
  int a = 1;
  foo(a);
  printf("a = %d\n", a);
  ```
Actual vs. Formal parameters

- Understanding the difference between the formal and actual parameters and how their values are related is the most important thing about understanding C functions.

- The style of parameter passing used in C is “call-by-value”
  - The value of the actual parameter is copied to the memory saved for the formal parameter when the function is called.
  - When the function returns the values of formal parameters are lost.
  - The return value is what the function returns.

- There are other ways of returning values from functions which involve pointers or global variables, which we will discuss later.
Actual vs. Formal parameters

- Assume a function which is declared as:
  ```
  int fie(int x) {
      ... 
  }
  ```
  - `x` is the formal parameter, it is a variable that can only be modified in the function body, except when the function is called its value is set (initialized) to the actual parameter.

- Now, assume some calls to this function:
  ```
  int t;
  int y = 4;
  fie(y);
  fie(y);
  t = fie(5);
  ```
  In the first call the actual parameter is the value of `y` (which is 4), in the second call the actual parameter is 5.
Scope of a variable

- Each variable has a scope
  - Scope identifies the visibility and duration of a variable in a program.
  - Scope of a variable changes based on where it is declared.

- A variable declared within a function is only visible within that function, it is not visible by other functions.
Scope of variables in C

• Local variables
  – These are variables that are defined within a block ({}), and their scope is only that block
    • They are not visible outside of the block they are declared
    – Note that function body is a block and it defines a scope. Scope of formal parameters is that block.
  – Duration: only as long as the block executes

• Global variables
  – These are variables declared outside any block and all C functions
  – Scope: everywhere in that compilation unit
  – Duration: as long as the program is executing
Functions and scope

Each function creates its own scope
• Variables can be declared locally
• Local variables hide identical but non-local names
• Local variables cannot be seen outside the function

C has global, static, local, and block scopes
• Blocks can be nested, functions cannot be nested
Scoping rules for C

- A variable can have global scope, all declarations of the same global name refer to a single instance.

- A variable can have file-wide scope (static), such names are visible to every function in the file containing the declaration.

- A variable can be declared locally in a function, then the scope of the name is the function body.
  - Formal parameters are like local variables.

- A variable can be declared within a block (denoted by a pair of curly braces). Such a variable is only visible inside the declaring block.
The function abstraction

There are 3 major things that functions provide
• Control abstraction
  – Well defined entries and exits
  – Mechanism to return control to caller
  – Assigning values of parameters (arguments) from caller’s scope to callee’s scope

• Scope
  – Declare locally visible names
  – Local names can hide identical, non-local names
  – Local names cannot be seen outside

• External Interface
  – Access is by function name and parameters
  – Function prototypes declare the external interface
Coercion of arguments

• Normally should pass the correct types – e.g., pass two int values to imax(int,int)
  – See ...demos/imax.chf (prints parameters and returns max)

• If you pass other types, they will be converted (coerced) to the formal parameter’s type

```ch>
ch> double largest = (double)imax(5.2,6.9999)
in imax: a = 5, b = 6
ch> largest
6.0000
```

• Coercion rules same as assignment (chapter 2)
  – Initializing parameter by assigning argument value
Recursion

• A recursive function is one that calls itself (directly or indirectly)
  – It is kind of like iterating. So what’s the big deal?

• Just that sometimes it is easier to solve a problem recursively than it is to solve it iteratively

• Key: Can we solve our problem by solving a smaller version of the same problem?
  – And then can we solve the smaller one by solving a smaller version of it, and so on?
  – e.g., \( n! = n \cdot (n-1)! \) (where \( n \geq 0 \), and \( 0! = 1 \))
3 essentials of a recursive solution

• Recursive factorial, for instance:
  
  ```java
  long factorial(int n) {
    // 1. Must have a base case:
    if (n <= 1) return 1;
    // 2. Recursive calls converge on base case:
    else return n * factorial(n-1);
  }
  
  // 3. Must actually solve the problem!
  //    – And should not unduly consume resources
  ```
Fibonacci numbers

• Fibonacci sequence is defined as
  \[ f(0) = f(1) = 1 \]
  \[ f(n) = f(n-1) + f(n-2) \text{ for } n > 1 \]

• The definition of the problem is itself recursive.
  – So, it is easy to solve this problem using recursion
• However, the recursive solution (see pp. 202-3) is not as efficient as iterative solution since it wastes intermediate results!
• Learn about “cost” of algorithms in CS 24 and later
  – Sometimes recursive solution can cost more than iteration as in Fibonacci numbers
• But some problems are best solved recursively
File input-output in C
File input-output in C

• First note these equivalent function calls:
  – `scanf(...)` ↔ `fscanf(stdin, ...)`
  – `printf(...)` ↔ `fprintf(stdout, ...)`
  – `getchar()` ↔ `getc(stdin)`
  – `putchar(...)` ↔ `putc(..., stdout)`

• Q. What are `stdin` and `stdout`?
  – A. File pointers (type is `FILE *`) – they “point” to standard input and standard output

• Of course you can point to a disk file too!
  – `fopen(...)` associates a file pointer with a file
# Opening and closing files

- Opening and closing a file means associating and disassociating a file pointer with that file

- e.g., `FILE *fp = fopen("mydata", "r");`
  - "r" means to read the file
  - Or use "w" to write, or "a" to append

- Best check `fp != NULL` before using though

- When done with `fp`, do `fclose(fp);
  - Especially important to close output files
Reading, writing and appending

- If we want to open a file to read from it, the file must exist before we open it
  - `FILE *fp = fopen("mydata", "r");`
  - If the file “mydata” does not exist, then `fp == NULL` after the call to `fopen`
- If we open a file to write to it, then it may or may not exist before. If it existed before, it will be overwritten!
  - `FILE *fp = fopen("mydata", "w");`
  - If there was a file called `mydata` its contents will be overwritten and lost
- We can open a file in the append mode
  - `FILE *fp = fopen("mydata", "a");`
  - The values we write will be appended to the end of the file. If the file does not exist a new file will be created.
Techniques for reading data files

- First find out how the file is organized
- Learn layout of logical and physical records
- A special first record might say how many
  See week05 demos data.txt and num_readfile.c
- Or read up to end of file
  See mydata.txt and eof_readfile.c
  (assumes there is at least one data)
  Need a way to detect EOF (end of file)
  fscanf() returns -1 when EOF reached.