600.120 Intermediate Programming, Spring 2017*

Misha Kazhdan

**Much of the code in these examples is not commented because it would otherwise not fit on the slides. This is bad coding practice in general and you should not follow my lead on this.*

Announcements

- HW1 due Wednesday
  - Make sure you know how to tar your file and scp/pscp it to your local machine so you can submit to Blackboard!
  - Submit a compiling version of it soon, even if it is not complete
  - Resubmit as many times as you like before 11pm (11:30pm if taking a late penalty) – we will only grade your last submission

Outline

- Writing functions
  - Function definitions and function calls
  - Parameter passing
  - Function declarations
  - Header files (`.h`)

Functions (definitions and calls)

- A function definition generally looks like:

```
return-type function-name( parameter-list )
{
        function-body
}
```

- A function call generally looks like:

```
function-name( argument values );
```

Functions (definitions and calls)

- A function definition generally looks

```
return-type function-
{
        function-body
}
```

- A function call generally looks like:

```
function-name( argum
```

```
#include <stdio.h>

float func( int x , float y )
{
        return x+y;
}

int main( void )
{
        int a = 7;
        float b = 2.5f;
        float c = func( a , b );
        float d = func( a , 3.5f );
        float e = func( a+2 , func( 3 , b ) );
        return 0;
}
```

Argument passing

- For functions to communicate, they take in a (possibly empty) set of parameters and return (at most) one value

- Argument values in C are passed by value
  - Function is given argument values that are copied into temporary variables

  ⇒ A called function cannot directly modify an argument in a way that will be visible to the the calling function

  ⇒ If we need to do that, we must provide the address of the variable to be altered

Argument passing

- For functions to communicate, they take in parameters and return (at most) one value

- Argument values in C are passed by value
  - Function is given argument values that are cop

  ⇒ A called function cannot directly modify an ar visible to the the calling function

  ⇒ If we need to do that, we must provide the ad

```
#include <stdio.h>

void func( int x )
{
        x += 5;
}

int main( void )
{
        int a = 7;
        func( a );
        printf( "%d\n" , a );
        return 0;
}
```

```
>> ./a.out
7
>>
```

Argument passing

- For functions to communicate, they take in parameters and return (at most) one value

- Argument values in C are passed by value
  - Function is given argument values that are cop

  ⇒ A called function cannot directly modify an ar visible to the the calling function

  ⇒ If we need to do that, we must provide the ad

- Recall: An array name is the address of first element, so if a function is passed an array argument it already has access to the original values

```
#include <stdio.h>

void trim( char* str )
{
        str[0] = '\0';
}

int main( void )
{
        char hi[] = "hello";
        trim( hi );
        printf( "%s\n" , hi );
        return 0;
}
```

```
>> ./a.out

>>
```

Argument passing

- For functions to communicate, they take parameters and return (at most) one va

- Argument values in C are <u>passed by valu</u>
  - Function is given argument values that are

  ⇒ A called function cannot directly modify a visible to the the calling function

  ⇒ If we need to do that, we must provide th

- <u>Recall</u>: An array name is the address of f passed an array argument it already has access to t es

```
#include <stdio.h>

void swap( char* str1 , char* str2 )
{
        char* temp = str1;
        str1 = str2;
        str2 = temp;
}

int main( void )
{
        char hi[] = “hello”;
        char bye[] = “goodbye”;
        swap( hi , bye );
        printf( “%s : %s\n” , hi , bye );
        return 0;
}
```

```
>> ./a.out
hello : goodbye
>>
```

Argument passing

- For functions to communicate, they take parameters and return (at most) one va
- Argument values in C are <u>passed by valu</u>
  - Function is given argument values that are

  ⇒ A called function cannot directly modify a visible to the the calling function

  ⇒ If we need to do that, we must provide th
- <u>Recall</u>: An array name is the address of f passed an array argument it already has access to t ues

```
#include <stdio.h>

void swap( char** str1 , char** str2 )
{
        char* temp = *str1;
        *str1 = *str2;
        *str2 = temp;
}

int main( void )
{
        char hi[] = "hello";
        char bye[] = "goodbye";
        swap( &hi , &bye );
        printf( "%s : %s\n" , hi , bye );
        return 0;
}
```

```
>> ./a.out
goodbye: hello
>>
```

Variable Scope

- The scope of a variable is the region of code in which that variable is known and accessible
  - Variables declared inside a function cannot be (directly) accessed from other functions
  - Each local variable declared in a function
    - Comes into existence when function is called
    - Must be initialized, otherwise has garbage value
    - Disappears when the function returns
    - Does not retain value from one call to the next
  - Function parameters are like local variables, except they are initialized by the argument value passed in during function call

```
#include <stdio.h>

int foo( int x )
{
        int y = 5;
        return x+y;
}

int main( void )
{
        int a = 7;
        a = foo( foo( a ) );
        printf( "%d\n" , a );
        return 0;
}
```

```
>>  ./a.out
17
>>
```

Variable Scope

- The scope of a variable is the region of code in which that variable is known and accessible
- External (global) variables are another option
  - The use of these is generally discouraged

```
#include <stdio.h>

int a = 7;

void foo( void )
{
        int y = 5;
        a += y;
}

int main( void )
{
        foo( );
        foo( );
        printf( "%d\n" , a );
        return 0;
}
```

Variable Scope

- The scope of a variable is the region c
  in which that variable is known and a

Q: What’s wrong with this code?

A: out_str only exists within the
scope of function trim5
The value at address out_str is not
defined after the function returns

```
#include <stdio.h>

char* trim5( char* in_str )
{
        char out_str[128];
        strcpy( out_str , in_str );
        out_str[5] = ‘\0’;
        return out_str;
}
int main( void )
{
        char* in = “goodbye”;
        char* out = trim5( in );
        printf( “%s\n” , out );
        return 0;
}
```

```
>> ./a.out
Segmentation fault (core dumped)
>>
```

Variable Scope

- The scope of a variable is the region of code in which that variable is known and accessible

```c
#include <stdio.h>

int foo( int x )
{
        int y = 5;
        return x+y;
}

int main( void )
{
        int a = 7;
        a = foo( foo( a ) );
        printf( “%d\n” , a );
        return 0;
}
```

```
>> ./a.out
17
>>
```

Variable Scope

- The scope of a variable is the region of code in which that variable is known and accessible

```
#include <stdio.h>

int main( void )
{
        int a = 7;
        a = foo( foo( a ) );
        printf( “%d\n” , a );
        return 0;
}

int foo( int x )
{
        int y = 5;
        return x+y;
}
```

```
>> gcc -std=c99 -pedantic -Wall -Wextra foo.c
foo.c: In function main:
foo.c:6:7: warning: implicit declaration of function foo [-Wimplicit-function-declaration]
   a = foo( foo( a ) );
```

Functions (declaration)

- For a function call, compiler is satisfied if it knows the (input) parameter list types and the (output) return type
  - It doesn’t need full definition to check if a call is legal (e.g. the types match)
  - The definition is required to execute the call. The linker’s job is to locate the definition when it is time to create the executable

Functions (declaration)

- We can declare a function before it is called and only define it after.
  - Note semicolon after parameter list
  - Declaration should appear before the first call to the function
  - A function declaration is also known as a *function prototype*
  - Names of parameters (e.g., x) are optional, but can be illuminating
    - It is recommended that you have them (and that they are meaningful) in your declarations so it's easier to understand what the function does and what to pass in.
  - Code tends to be more readable if functions are declared before main and defined after.

```
#include <stdio.h>

int foo( int );

int main( void )
{
        int a = 7;
        int b = foo( foo( a ) );
        printf( "%d\n" , b );
        return 0;
}

int foo( int x )
{
        int y = 5;
        return x+y;
}
```

Functions

Q: Why not just put everything in the main function?

A: Modularizing code:

- ✓Makes it easier to debug
- ✓Makes it easier to read
- ✓Makes it easer to work in a team
- ✓Makes it possible to reuse code
- ✘ May result in non-optimial code
- ✘ Function overhead is inefficient

```
#include <stdio.h>

int foo( int x , int y )
{
        // A lot of expensive calculation on x
        return x * y;
}

int main( void )
{
        printf( "%d\n" , foo( 1 , 0 ) );
        return 0;
}
```

Header files

When the pre-processor sees a #include directive it inserts the contents of the specified file at that location in the code

- The included file contains the declarations of the functions that are used

```
#include <stdio.h>

int main( void )
{
        printf( "hello world!\n" );
        return 0;
}
```

```
...
/* Write formatted output to STREAM.
   This function is a possible cancellation point and therefore not
   marked with __THROW.  */
extern int fprintf (FILE *__restrict __stream, const char *__restrict __format, ...);

/* Write formatted output to stdout.
   This function is a possible cancellation point and therefore not
   marked with __THROW.  */
extern int printf (const char *__restrict __format, ...);

/* Write formatted output to S.  */
extern int sprintf (char *__restrict __s, const char *__restrict __format, ...)
__THROWNL;
...
```

/usr/include/stdio.h

```
int main( void )
{
        printf( "hello world!\n" );
        return 0;
}
```

Header files

When the pre-processor sees a #include directive it inserts the contents of the specified file at that location in the code

- The included file contains the declarations of the functions that are used
- (Inclusion has to happen before the function is called)
- Two closely-related ways to include files:
  - #include <stdio.h>
  - #include “stdio.h”

```
#include <stdio.h>

int main( void )
{
        printf( “hello world!\n” );
        return 0;
}
```

In-Class Exercises

- On Piazza, find Resources section, then click Resources tab
- Scroll down to section for this course section
- Find link for Exercise 3-1 and follow it
- Follow the instructions; raise your hand if you get stuck
- Make sure you check in with a course staff member sometime during this session