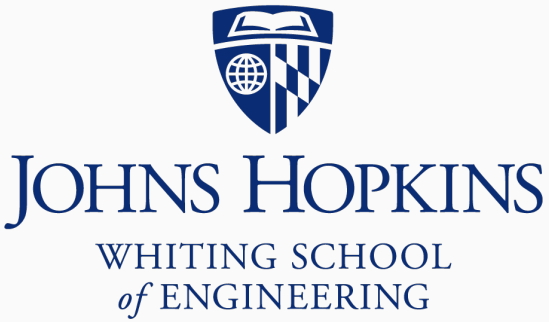
Functions

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Functions

So far we have written our programs so that all the functionality is in the `main` function

Real-world programs are spread over many functions; this:

- Helps the programmer focus on smaller problems, one at a time
- Improves readability
- Helps with testing
    - Can test *functions* one by one, as we'll see
- Easier to collaborate
    - "Alice will write function X, Bob will write function Y, Carol will write Z assuming X and Y exist."

Functions

Putting a chunk of code in a new function is appropriate when:

- The code has a clear, distinct goal
- The number of variables used is not large
- Has one result (or a few)

Function examples

```c
int add(int lhs, int rhs) {
    int sum = lhs + rhs;
    return sum; // 2. returns from add
}

int main() {
    printf("%d\n", add(4, 5)); // 1. calls add
    return 0; // 3. returns from main, exiting program
}
```

Here: `main` is the *caller*, add is the *callee*

`return` exits add, sending the return value to callee

`return` from `main` is special: exits the program

Functions

```
float fahrenheit_to_celcius(float fahrenheit) {
    float scale = 5.0 / 9.0;
    return scale * (fahrenheit - 32);
}
```

Function can have its own *local* variables (`scale`)

`fahrenheit` is a *parameter*, and also a local variable

When function returns, local variables "disappear"

Functions

```c
#include <stdio.h>

int sum(int a, int b) {
    int c = a + b;
    // c "disappears" when function returns
    return c;
}

int main() {
    sum(7, 5);
    // error: there's nothing called "c"
    printf("%d\n", c);
}
```

Functions

```
$ gcc function_eg2.c -std=c99 -pedantic -Wall -Wextra
function_eg2.c: In function 'main':
function_eg2.c:12:20: error: 'c' undeclared (first use in this function)
     printf("%d\n", c);
                    ^
function_eg2.c:12:20: note: each undeclared identifier is reported only once for
each function it appears in
```

Functions

There are two variables called c here; changing one doesn't affect other

```c
#include <stdio.h>

int sum(int a, int b) {
    int c = a + b; // doesn't affect 'c' in main
    return c;
}

int main() {
    int c = 0;
    sum(7, 5);
    printf("%d\n", c);
    return 0;
}
```

```
$ gcc function_eg3.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
0
```

Functions

```c
int sum(int a, int b) {
    int c = a + b;
    return c;
}
```

In C, parameters are passed *by value*

Callee's parameter receives a *copy* of the value

In general, changes to local variables (including parameters) in the callee are *not* reflected in the caller

Functions

```c
#include <stdio.h>

void assign_7(int a) { // different `a` from the one in main
    a = 7;
}

int main() {
    int a = 0;
    assign_7(a);
    printf("%d\n", a); // main's `a` is unchanged
    return 0;
}
```

Functions

Compiler warns us in this case:

```
$ gcc function_eg4.c -std=c99 -pedantic -Wall -Wextra
function_eg4.c: In function 'assign_7':
function_eg4.c:3:19: warning: parameter 'a' set but not used [-Wunused-but-set-
parameter]
 void assign_7(int a) { // different `a` from the one in main
                   ^
$ ./a.out
0
```

Functions

An alternative to passing by value is to pass *by pointer*, which allows callee to modify a variable local to the caller

We've done this: `scanf("%f%f", &a, &b)`

We'll return to this when we talk about pointers

Passing arrays & strings is a special topic that we cover a little later

Functions

Functions can call other functions, which can call other functions. . .

```
#include <stdio.h>

int add2(int a, int b) {
    return a + b;
}

int add3(int a, int b, int c) {
    return add2(a, add2(b, c));
}

int main() {
    printf("%d\n", add3(10, 20, 5));
    return 0;
}
```

```
$ gcc function_eg5.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
35
```

Functions

A *recursive* function is a function that calls itself

```
// Euclidean algorithm for greatest common divisor
int gcd(int a, int b) {
    if(a == 0) {
        return b;
    }
    return gcd(b % a, a);
}
```

Be careful; it's easy to accidentally write a recursive function that never exits, causing a *stack overflow*

Functions

Saw we have a program that calculates compound interest that is structured like this:

```
float compound(float p, float r, int n) {
    ...
}

int main() {
    ...
    float monthly = compound(p, r, 12);
    ...
}
```

compound is above `main`. What if we put it below?

Functions

```c
#include <stdio.h>
#include <math.h>

int main() {
    printf("%f\n", compound(1000.0, 0.05, 12));
    return 0;
}

float compound(float p, float r, int n) {
    return p * pow(1 + r/n, n);
}
```

Functions

```
$ gcc function_eg6.c -std=c99 -pedantic -Wall -Wextra -lm
function_eg6.c: In function 'main':
function_eg6.c:5:20: warning: implicit declaration of function
    'compound' [-Wimplicit-function-declaration]

     printf("%f\n", compound(1000.0, 0.05, 12));
                    ^~~~~~~~
function_eg6.c:5:14: warning: format '%f' expects argument of
    type 'double', but argument 2 has type 'int' [-Wformat=]

     printf("%f\n", compound(1000.0, 0.05, 12));
             ~^     ~~~~~~~~~~~~~~~~~~~~~~~~~~
             %d
function_eg6.c: At top level:
function_eg6.c:9:7: error: conflicting types for 'compound'
 float compound(float p, float r, int n) {
       ^~~~~~~~
function_eg6.c:5:20: note: previous implicit declaration of
    'compound' was here

     printf("%f\n", compound(1000.0, 0.05, 12));
                    ^~~~~~~~
```

Functions

If the compiler finds a call to a function that hasn't been defined or declared, it tries to *infer* the parameter and return types

- If it infers correctly, you may only see a warning like the first one (which you should fix anyway)
- If it infers incorrectly, you'll see *more* warnings and errors; above, compiler incorrectly infers return type `int`

Functions

Compiler needs to know about the function before we call it

Callee must be defined or declared (*prototyped*) before caller

`implicit declaration of function` warning means this hasn't been done properly; you should fix it, regardless of what compiler says after

Functions

This is a function *definition*

```
int add2(int a, int b) {
    return a + b;
}
```

Specifies name, return type, parameters (type and name for each), and implementation

Functions

If implementation is omitted, function is *declared* but not defined. This is a *function prototype*.

```
int add2(int a, int b);
```

Definition can be given later

When writing function `A`, you can call function `B` as long as `B` has been defined or prototyped *previously*

Functions

```c
#include <stdio.h>
#include <math.h>

// function declaration (prototype)
float compound(float p, float r, int n);

int main() {
    printf("%f\n", compound(1000.0, 0.05, 12));
    return 0;
}

// function definition
float compound(float p, float r, int n) {
    return p * pow(1 + r/n, n);
}
```

```
$ gcc function_eg7.c -std=c99 -pedantic -Wall -Wextra -lm
$ ./a.out
1051.162598
```

Functions

Parameter names can be omitted in prototypes

```
float compound(float p, float r, int n); // OK

float compound(float, float, int); // also OK
```

Functions

Functions cannot be “nested” – unlike Python

```
// This WON'T compile
int add3(int a, int b, int c) {
    int add2(int d, int e) {
        return d + e;
    }
    return add2(a, add2(b, c));
}
```

C++ *lambda functions* accomplish something like this; but not discussed here