C Functions

Kurt Schmidt
Skipjack Solutions
March 11, 2022
Intro
Intended audience: Student who has working knowledge of Python

Target compiler: I’ll try to center the discussion on C99 using gcc 7

- Code examples might have an accompanying link
  - Follow link to step through example at pythontutor.com
  - Does a nice job of graphically showing variables in memory, the heap, and the stack
Functions
Functions

- Can *not* be defined inside another function (must be global)\(^1\)
- Return type part of function declaration
  - Can be `void`
  - `void` is a place-holder type
  - As a return type, it indicates that the function doesn’t return a value
- Arguments are passed *by value*\(^2\)
- Must be declared before use
  - A *prototype* serves
- May not be overloaded
- C supports recursion

\(^1\) GCC will allow it, but, it’s not C
\(^2\) Except for arrays
#include <string.h>
#include <stdio.h>

/* prototype */
int foo( int k ) ;

int main( void ) {
  int j = 12, k = 13, r ;
  r = foo( j ) ;
  printf( "foo returned %d\n", r ) ;
  if( j != 12 )
    printf( "j changed\n" ) ;
  if( k != 13 )
    printf( "k changed\n" ) ;
  return 0 ;
}

int foo( int k )
{
  /* j is local, and
     k behaves so */
  int j = 23 ;
  int rv = k * j ;
  k = 928357 ;
  return rv ;
}

• https://goo.gl/v4rZgR
main.c

```c
#include <string.h>
#include <stdio.h>
#include "foo.h"

int main( void ) {
    int j = 12,
        k = 13,
        r ;
    r = foo( j ) ;
    printf( "foo returned %d\n", r ) ;
    return 0 ;
}
```

foo.h

```c
#ifndef __MY_FOO_H_
#define __MY_FOO_H_
int foo( int k ) ;
#endif /* __MY_FOO_H_ */
```

foo.c

```c
#include "foo.h"

int foo( int k )
{
    int j = 23 ;
    int rv = k * j ;
    k = 928357 ;
    return rv ;
}
```

```
$ gcc main.c foo.c -o foo #$
```
Scoping
Scoping in C

- C uses *lexical scoping* (or *static binding*)
- A symbol (variable name) is bound to a location at compile time
- A variable is either
  - Local to the function it appears in (if it’s declared there, or is a formal parameter), or
  - Global (declared outside of all functions, possibly in another file)
- Pretty straightforward in C, since we can’t define functions inside of other functions
Array Arguments
Passing Arrays to Functions

- Arrays are passed by reference
  - The name of the array is a pointer
  - The *pointer* is copied in
  - Semantics are preserved

- Array can be modified by the function
- Size (number of elements) must also be passed to function
  - Unless a suitable sentinel value exists (e.g., ‘\0’)
```c
#include <stdio.h>

void arrPrint( float a[], int n, FILE* ofile )
{
    for( int i=0; i<n; ++i )
        fprintf( ofile, "%f ", a[i] ) ; /* access by index */
}

int main()
{
    int a[20] = { 5, 12, 13, 23, 42, 67 } ;

    arrPrint( a, 6, stdout ) ;

    return 0 ;
}
```
Function Header Comments
Side-effects are some evidence the function was called. The state of the machine is different.
Some examples:

- A global variable was changed
- A reference argument was changed
- Read from a file (moved the input pointer)
- Wrote to the screen
Any function header comment should include:

- The purpose of the function
- A description of the inputs
  - Type, and/or units, as applicable
- A description of the output (return value)
- Disclosure of any side-effects

```c
/* sin - computes sin(t)
 * Inputs: t, in radians
 * Output: sin(t)
 * Side-Effects: Removes 100g of dark chocolate */

double sin( double t ) ;
```
It is helpful to think in terms of **preconditions** and **postconditions** when crafting an interface.

**Preconditions**  Anything the function assumes to be true in order to do its job, including:

- Types or units of arguments
- Existence or condition of global variables or reference arguments

**Postconditions**  What the function guarantees to be true, assuming the preconditions were met, including:

- Types or units of returned variable
- State of global or referenced variables (side-effects)