COMPSCI 1XC3 - Computer Science Practice and Experience: Development Basics

Topic 6 - Functions and Arrays in C

Zheng Zheng

McMaster University

Winter 2023

Abstraction Satisfaction

Functions are the basic unit of abstraction in C.

- They are highly similar to how Python implements functions, however there are some important differences.
  - The return type of the function must be indicated where Python uses the `def` keyword.
    - In the absence of a return type declaration, gcc will give a warning, and the return type will default to `int`.
  - Curly braces are used instead of a colon and indentation.
  - The arguments must also have their types declared.
    - Again, the default is `int`.
  - While a return statement is not required, gcc will complain if one isn't present...

For Example...

Python:

```python
def max (x,y) :
  if (x > y) :
    return x
  else
    return y
```

C:

```c
int max (int x, int y) {
  if (x > y) {
    return x;
  } else {
    return y;
  }
}
```

Some Function-Related Warnings

Let's take a look at the following poorly written function:

```
1 max (x, y) {
2   if (x > y) { // return x;
3 } else { // return y;
4 } }
```

while this will *technically* compile, the following warnings are found:

```
warning: return type defaults to 'int'
 max (x, y) {
 ^~~
In function 'max':
warning: type of 'x' defaults to 'int'
warning: type of 'y' defaults to 'int'
warning: control reaches end of non-void function
 }
```

Function Prototypes

In C, as in Python, in order for a function to be in scope, it must be defined before it is used.

- If the function `max` were defined after its use in `main`, gcc produces the following warning.
- Early versions of C did not check for correct function usage at compile time, so the error would show up at runtime.
- Function prototypes solve the issue of having function typing information available, while maintaining C's back-compatability.
- That is why this is a warning and not an error:

```
warning: implicit declaration of function 'max';
      did you mean 'main'?
  int q = max(4,5);
          ^~~
          main
```

Function Prototypes (cont.)

To solve this problem, C has borrowed **function prototypes** from C++.

```
1 int max (int x, int y);
```

- A function prototype is syntactically the first line of a function declaration, terminated by `;`
- It is good practice to put function prototypes immediately after your preprocessor commands (i.e., `#include`'s)
- This "declares" the function up front, so it's definition may now occur anywhere in the file without generating any warnings.

Library Architecture

Computer Architecture *is a set of rules and methods that describe the functionality, organization, and implementation of computer systems.*

– Wikipedia –

Header Files vs Libraries

Things that are useful:

- The ability to call functions defined outside of a source code file.
- The ability to pre-compile these functions
- The ability to share library functions among many programs.

These purposes are served by the library system.

- **Header Files** - files containing the prototypes of functions made available by a specific library.
- **Library** - files in which the functions specified in the header file are implemented. There are two main types: **Static** and **Dynamic** (see the next slide).

Because their functionality is so closely linked, when a person says "a library", they usually mean both the library and header file taken in combination.

Static vs Dynamic Libraries

- **Static** - Derived from the Greek "statikos", meaning "causing to stand"
  - Contain object code linked with a program, which then is integrated into the executable.
  - Included at compile time (i.e., up front)
  - Consequentially, the compiler needs to be able to find it.
- **Dynamic** - Derived from the Greek "dunamikos", meaning "powerful"
  - Linked to the executable at compile time, but the implementation of the linked function is loaded at *run-time*.
  - Linking resolves undefined references in the source file
  - May be shared with many programs.

Find it in Your Local Library

There are two ways to include files.

```
1 #include <stdio.h>
2 #include "myheader.c"
```

- If the filename is enclosed in quotation marks, gcc will search both the first relative directory of the current file and a preconfigured list of standard system directories.
- If angle braces are used, *only* the standard system directories are searched.
- Technically this means you could use quotes for everything, but by convention, if it's a standard library header, use angle braces.

Header files for miles!

From the gcc documentation:

- Header files serve two purposes.
  - **System header files** declare the interfaces to parts of the operating system. You include them in your program to supply the definitions and declarations you need to invoke system calls and libraries.
  - Your own header files contain declarations for interfaces between the source files of your program. Each time you have a group of related declarations and macro definitions all or most of which are needed in several different source files, it is a good idea to create a header file for them.

By convention, header files have the *.h file extension.

Some Standard Library Headers

Header	Explanation
`<assert.h>`	Contains information for adding diagnostics that aid program debugging.
`<ctype.h>`	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.
`<errno.h>`	Defines macros that are useful for reporting error conditions.
`<float.h>`	Contains the floating-point size limits of the system.
`<limits.h>`	Contains the integral size limits of the system.
`<locale.h>`	Contains function prototypes and other information that enables a program to be modified for the current locale on which it's running. The notion of locale enables the computer system to handle different conventions for expressing data such as dates, times, currency amounts and large numbers throughout the world.
`<math.h>`	Contains function prototypes for math library functions.
`<setjmp.h>`	Contains function prototypes for functions that allow bypassing of the usual function call and return sequence.
`<signal.h>`	Contains function prototypes and macros to handle various conditions that may arise during program execution.

Fig. 5.10 | Some of the standard library headers. (Part 1 of 2.)

Some Standard Library Headers

Header	Explanation
`<stdarg.h>`	Defines macros for dealing with a list of arguments to a function whose number and types are unknown.
`<stddef.h>`	Contains common type definitions used by C for performing calculations.
`<stdio.h>`	Contains function prototypes for the standard input/output library functions, and information used by them.
`<stdlib.h>`	Contains function prototypes for conversions of numbers to text and text to numbers, memory allocation, random numbers and other utility functions.
`<string.h>`	Contains function prototypes for string-processing functions.
`<time.h>`	Contains function prototypes and types for manipulating the time and date.

Fig. 5.10 | Some of the standard library headers. (Part 2 of 2.)

Library Example

```c
// >>>>>>> top.c >>>>>>>
#include <stdio.h>
#include "mylib.h"
int main () {
  int q = max(4,5);
  printf("The maximum is %d\n", q);
}
```

```c
// >>>>>>> mylib.h >>>>>>>
int max (int x, int y);
```

```c
// >>>>>>> mylib.c >>>>>>>
int max (int x, int y) {
  if (x > y) { return x;
  } else { return y;
} }
```

Static Compilation

- Note that `lib.c` does not contain a `main` function!

All three of these files are compiled and linked with a single invokation of gcc:

```
gcc -Wall top.c mylib.c -o top
```

- Runnning gcc on only `top.c` (omitting `mylib.c`) will result in an undefined reference, which is an **error**!

An Alternative Procedure

It is a stylistic recommendation to write header files for your C files below the top-level. Especially if they are **Large!**

- This emulates the way other programming languages use **packages** and **modules**.

However, the header file may be successfully omitted. In the previous example:

- Change `mylib.h` to `mylib.c` on line 3 of `top.c`
- Omit `mylib.c` from the gcc invokation.

Dynamic Libraries

- The procedure for creating a dynamic library is operating system dependent, but the goal is to create:
  - A shared object file (`*.so`) [Mac/Linux]
  - Or a dynamically linked library file (`*.dll`) [Windows]
- If set up properly, a dynamic library can be used by programs written in *languages other than C!*
- **BE WARNED!!** Messing around in your operating system's hidden folders can have **SERIOUS CONSEQUENCES**!
  - Never mess around with a computer like this if you don't have your data backed up
  - Don't say your professor didn't warn you!

One of the lab activities in this course will require you to create and install a shared object library

Introducing the **Call Stack!**

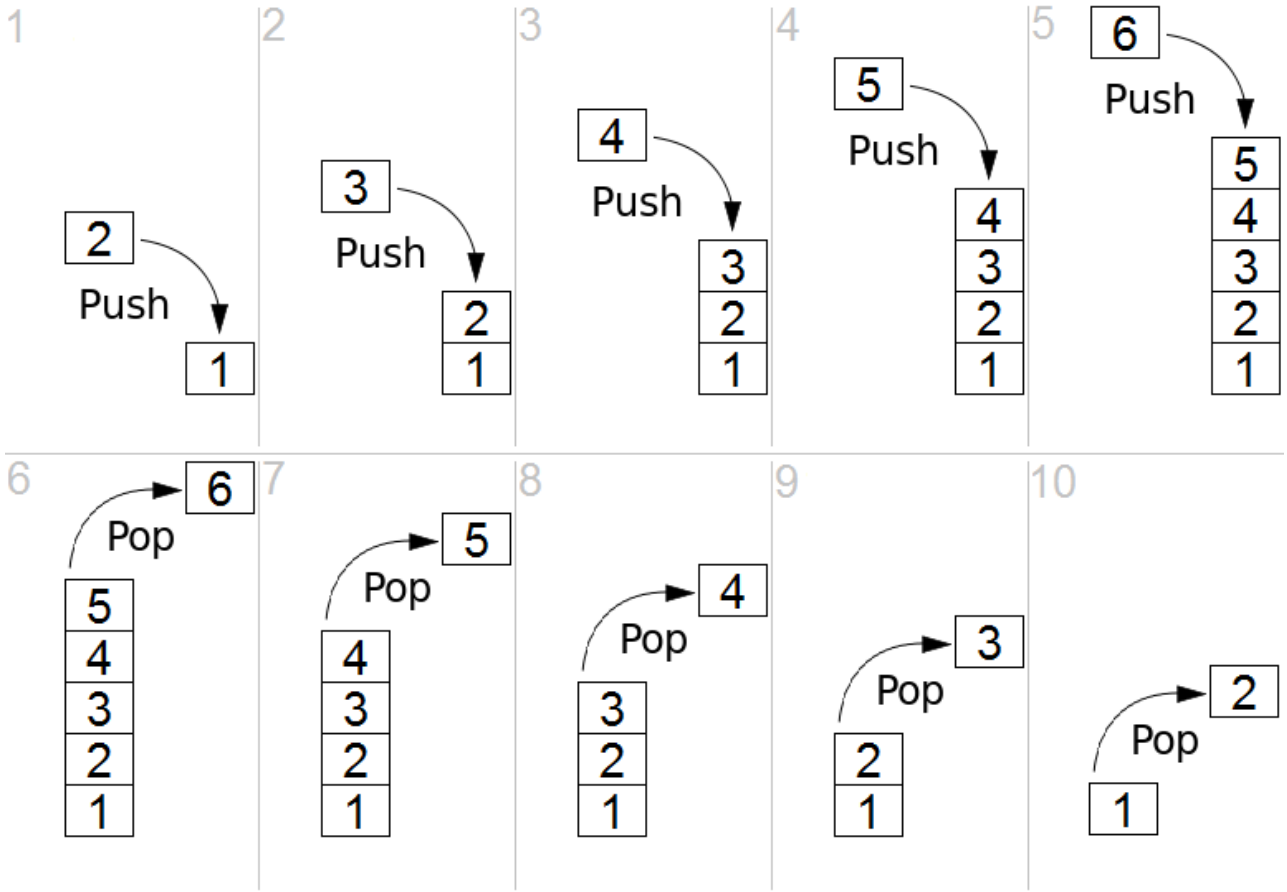
Have you ever wondered how your computer keeps track of all those function calls?

- function calls are stored in your system's **function call stack** or just **call stack**, or, reverently, **The Stack**.
- The call stack has a fixed, or *static* size.
- If too many items are added to the call stack, this can result in an error called **stack overflow** (which what `https://stackoverflow.com/` is referencing!)
- In order to understand the call stack, we need to know some rudimentary data structures!

Stacks of Stacks

- A **stack** is a data structure that always returns the most recently added element.
- This is also known as LIFO (last-in, first-out)
- There are two primary operations on stacks:
  - **Push** - An element is added to the stack
  - **Pop** - The element in the stack that was added most recently is removed from the stack and returned.
- The kitchen metaphor is a stack of plates. How you use a stack of dishes in your cupboard is how your computer uses stacks for data.
- Undo/redo actions in many programs are also stored in a stack.

A Stack in Operation



Call Stack Functionalities

The call stack:

- enables programs to "step into" functions while keeping their place in the program that called the function.
  - Functions are properly thought of as *sub-programs*
- supports the creation, maintenance, and destruction of each function's local variables. This is also known as **namespace**.
- keeps track of the return addresses that each function needs to return control to the invoking function.

Think of the previous stack example, but replace the numbers with all of the data necessary to execute a function, including variables. These are known as **stack frames**.

Pass By Reference vs Pass By Value

There are two main ways that a programming language will pass data into its functions:

- **By Reference** - Perl, VB, Fortran
  - Variables are passed into functions as references to data, rather than the data itself.
  - Modifying the function argument inside the function will modify the variable passed into the function, within the context of the calling program.
- **By Value** - C, Java, Python*
  - The bit values of the variables are passed into functions, rather than references to the memory contained in them.
  - Modifying an argument does not effect anything outside the function's local context.

C can pass by reference using pointers, but you have to do the legwork yourself ;-)

Creating Our Own Types!

The `typedef` keyword allows us to define type synonyms:

```
1 typedef unsigned char BYTE;
```

This allows us to use the new type `BYTE` anywhere a type must be declared:

```
1 ...
2   BYTE foo = 0;
3 ...
```

The compiler will swap in the type's synonym at compile time. Type synonyms can be very useful for improving the readability of your code, as they allow for more descriptive type information!
By convention, data types are capitalized!

enum With Terror!

Another way we can define custom types in C is with **enumerations**

```
1 enum PKMN_STATUS {FNT, SLP, PRZ, PSN, FRZ, BRN, None};
```

Not only is the type available, but the enumerats may also be used as literals:

```
1 ...
2   PKMN_STATUS stat = None;
3   if (stat == PSN) {
4     hp = hp - hp / 16;
5   }
6   if (hp <= 0) {
7     stat = FNT;
8   }
9 ...
```

enum With Terror! (cont.)

In reality, each of the enumerats is an alias for a literal bit value. We can expose these with the following program:

```
1 #include <stdio.h>
2 enum PKMN_STATUS {FNT, SLP, PRZ, PSN, FRZ, BRN, None};
3 int main() {
4   printf("FNT = %d\n", FNT);
5   printf("SLP = %d\n", SLP);
6   printf("PRZ = %d\n", PRZ);
7 }
```

Just like typedefs, the main purpose of enumerations is *code readability!*

Arrays: The Simplest Data Structure

In Python, there are many data structures to choose from:

- Tuples
- Lists
- Dictionaries
- Sets
- Strings

In C, the only data structure supported natively is the **Array**. If you need something more complex than an Array, you have to either find a library defining it or define it yourself.

So what is an `Array`?

An array is a contiguous segment of memory, which may be accessed via linear indexing.

- All elements of an array have the same type.
- Arrays use **zero-indexing**.
- Arrays are *static*
  - The size of an array does not change during execution
  - This means that the size of the array *must* be known at the time of declaration.
  - There are ways of getting around this, but not without pointers (which we'll be talking about soon...)

Syntax, my friends!

An array is declared in the following manner:

```
1 int c[x];
```

- Note, that x must be either an integer literal or an expression which evaluates to an integer.
  - This rule applies to array indexing as well as declaration.
- On declaration, an array is filled with *junk data!*

Arrays may be indexed using the array index operator:

```
1 c[7] = 5;
```

- Binds at the same level as function calls (so very tightly)
- As we'll see later, the index operator is syntactic sugar for some pointer operations.

Pretty Printing Arrays

```
1  #include <stdio.h>
2  int main () {
3    int c[(6+6)];
4    float bar[] = {0.0, 0.1, 0.2, 0.3};
5    printf("%d\n", c);
6    printf("%p\n", c);
7    printf("[");
8    for (int i = 0; i < 12; i++) {
9      if (i < 11) {
10       printf("%d, ", c[i]);
11     } else {
12       printf("%d]\n", c[i]);
13     }
14   }
15 }
```

Pretty Printing Arrays (Visualization)

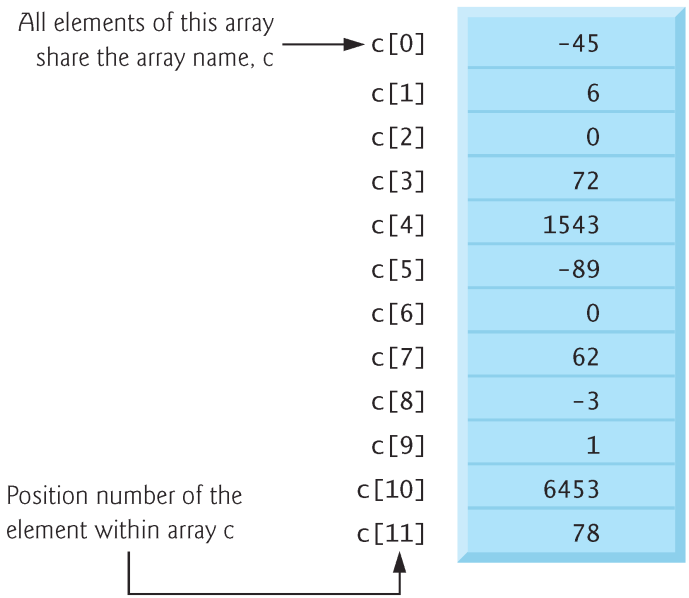


Fig. 6.1 | 12-element array.

Pretty Printing Arrays (output)

Compiler Warnings:

```
In function 'main':
warning: format '%d' expects argument of type 'int',
  but argument 2 has type 'int *'
  printf("%d\n", c);
          ~^     ~~~
          %ls
```

Output:

```
-1506989472
0x7ffca62d2a60
[1, 0, 56797197, 22077, 997394928, 32685, 0, 0,
  56797120, 22077, 56796656, 22077]
```

Pretty Printing Arrays (Analysis)

Just like with a lot of things in C, there are no free lunches!

- You can't print a whole array just by passing the array's name to `printf` as the array identifier is really a **memory address**, or **pointer**.

Initialization!

- An uninitialized array is full of garbage data!
- To initialize an array, provide a comma separated series of type-correct literals or expressions, delimited with curly braces!
- If the initializer is smaller than the declared memory, C backfills with zeros!
- If you initialize the array, you may omit specifying the size (the size of the initializing array will be used).

Arrays and Functions

Consider the following function prototype:

```
1 int* myFun (const int input[]);
```

- `input` is declared as an array by the inclusion of `[]`
  - `int*` is also valid.
- `int*` tells us that this function returns an **integer pointer**
  - Arrays in C are syntactic sugar for pointers.
- the `const` term locks the pointer so that it can't be modified. This essentially makes it "read-only"
  - This will prevent any of the array's elements from being modified, as well as the value of the pointer.
  - If you want to be able to modify the array, just leave out the `const` keyword.
  - You can do this with any argument, not just arrays!

sizeof

In Python, we have the `len()` function to quickly and easily tell us how large our data structures are.

- The equivalent fuction in C is `sizeof()`, but as with most things in C, it's a bit more complicated.
- `sizeof()` returns the size *in bytes* that the provided argument was declared with.
- `sizeof()` may be used on variables of any type, not just arrays.
- Accordingly, the *declared* length of an array is equal to the size of the array (in bytes) divided by the size of any element of the array (in bytes):

```
1 int foo[] = {1,2,3,4,5,6}
2 length = sizeof(foo) / sizeof(foo[0]);
```

Accessing Higher Dimensions

Linear arrays are all well and good, but what if I need to represent a matrix?

```
1 // Declaring a 2D array
2 int foo[][] = {{1, 2}, {3, 4}};
3 // Declaring a 3D array
4 int bar[5][5][5];
```

- Dimensionality is indicated by the number of square braces following the identifier.
- These are correctly thought of as *arrays of arrays*.
- Providing $n - k$ indexes to an n dimensional array will produce a k dimensional array.

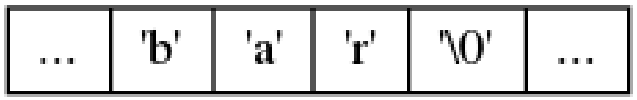
Strings are Character Arrays!

Character Arrays, not Character Sheets...

As it turns out, C handles strings as **character arrays.** Consider the following declarations:

```
1 char foo[] = "bar";
```

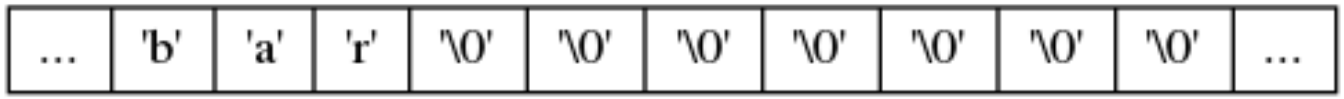
`foo` will be written into memory as:



If we specify a size of ten,

```
1 char foo[10] = "bar"
```

we get the following:



String Things

- `string` isn't even a keyword in C!
- Character arrays may be indexed, just like regular arrays.
- String literals are always terminated by the **null character** implicitly!
- All strings must be null terminated.
- We can receive strings directly from `scanf` using the`%s` format specifier.

```
1 scanf("%14s", foo);
```

- By inserting a number, the format specifier may even be used to limit the number of characters that get copied into the character array.
- `foo` is actually a **pointer**, so we don't need the address-of operator `&` in scanf. `foo` is already a memory address.

Overflow Attacks!

A common form of security vulnerablility in C and C++ programs is **array overflow**.

- Arrays are replaced with pointer arithmetic by the compiler, with no bounds checking!
- If you tell C to write 100 characters into a 50 character array, it will happily do so!
- The extra 50 characters will be written into the memory that comes after the character array.
- This can overwrite all kinds of useful things, like other variables. If the character array is stored in the stack, function call informaton can also be vulnerable.

The moral of the story is that you should always limit the things you write into an array to the size of the array MANUALLY!

Smash that Stack!

```c
#include <stdio.h>

int main() {
  char query[10];
  printf("Enter A Query : ");
  scanf("%s", query);
  printf("The query is %s\n", query);
}
```

```
Enter A Query : jjjjjjjjjjjjjjjjjjjjjjjjjjjjjj
The query is jjjjjjjjjjjjjjjjjjjjjjjjjjjjjj
*** stack smashing detected ***: <unknown> terminated
  Aborted (core dumped)
```

Acknowledge

The contents of these slides were liberally borrowed (with permission) from slides from the Summer 2021 offering of 1XC3 (by Dr. Nicholas Moore).