Chapter Six

Functions and Pointers

Chapter 6

Chapter 6 – Concepts

- Storage classes and scope.
- Interactions with functions.
- Functions without arguments, functions without values.
- Pointers.
- Pointer arguments to functions.

6.1 The main function

Every C program is a collection of functions.

Every program must have exactly one `main` function.

Function `main` is called by the shell and returns a value to the shell. Arguments `argc` and `argv` are described in Chapter 7.

If `exit` is called, all pending calls are aborted, and all stack frames associated with the program are popped.

6.2 Use of void

An argument list of `void` indicates that there are no arguments.

A return type of `void` indicates no return value.

Any function (or expression) can be used as a stand-alone statement, with the value discarded.

- `void.c`

6.3 Scope

Variables declared in a function are local to the function, and housed in the function's stack frame.

They are destroyed when the function returns.

If the function is called again they are created afresh. They do not retain values between calls.

6.3 Scope

Argument variables are also housed in the stack frame.

Changes made to argument variables do not cause change the calling context.

- `scope1.c`

6.4 Global variables

Variables declared outside any function are global, and may be accessed by any function within that same file.

- `scope2.c`

All of the functions declared in a file are also global.

6.4 Global variables

If a local variable is declared with the same name as a global variable, then it shadows the global variable. Shadowing happens even if the two objects are of different types.

Local variable names should be chosen to avoid conflicts with library-defined names such as `sqrt`.

6.4 Global variables

Global variables have significant drawbacks.

They make functions less general, and oppose abstraction, by binding particular variable names into the function when it is compiled.

Best is to just avoid them.

6.5 Static variables

A local variable with the storage class `static` is allocated with the global variables, not in the stack frame.

A static variable is only initialized once, and only one copy of it is created for the program. It does retain its values between calls.

- `scope3.c`

Never mix recursive functions and static variables.

6.5 Exercise 1

Study the program. For each function, describe what variables are in scope. What is the output?

- `scope4.c`

6.6 Pointers and pointer operations

Each variable is stored in some memory location, and accessed via its memory address.

Each address references a byte of memory. Each byte contains eight bits, and each bit can store one binary digit, a zero or a one.

The compiler converts symbolic names into memory addresses (or offsets within the current stack frame) that are used when the program is executing.

6.6 Pointers and pointer operations

An `int` variable typically requires 32 bits, or 4 bytes. A `double` variable typically requires 8 bytes; and a `char` typically requires 1 byte.

Current computers typically have four billion bytes or more of memory enough to store the complete contents of something like 8,000 books.

6.6 Pointers and pointer operations

A pointer is a variable that can store an address.

Pointer values are addresses, and so make sense only while a program is running.

Each time a program executes, it might be loaded into a different section of memory.

Pointer values must be regarded as transient – used while the program is active, then forgotten about.

6.6 Pointers and pointer operations

There are two operations that make pointers useful.

The operator "&" means "the address of". It can only be applied to variables (not expressions). When applied, it generates the numeric address of that variable:

- pointer1.c

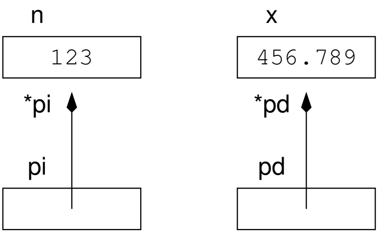
6.6 Pointers and pointer operations

The operator "*" means "use this address to access the underlying variable stored at the indicated memory location". It undoes the action of "&", and `*(&x)` is the same as `x`.

Pointer variables must have an underlying type declared. For example, a "pointer to `int`" variable is a different type to either an "`int`" variable or a "pointer to `double`" variable.

Like all variables, pointers must initialized before use.

- `pointer2.c`



The dereferencing operator $*$ can be used to alter the value of the variable whose address is stored in the pointer.

6.6 Pointers and pointer operations

If an uninitialized pointer is used, one of a range of things can happen:

- Immediate "segmentation fault", and program abort.
- Mysterious failure later in the execution of the program.
- Incorrect results, but no obvious failure.
- Correct results, but maybe not always, and maybe not when executed on another machine, or when executed on another day.

The first is the most desirable, but cannot be relied on.

You need to be very careful!

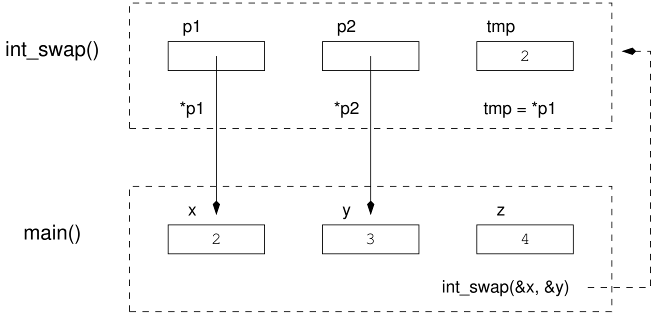
6.7 Pointers as arguments

So what? If pointers are dangerous, why do we need them? Because we can pass pointer arguments to functions...

- `pointer3.c`

6.7 Pointers as arguments

After `t=*p1` is executed in the first call to `int_swap`:



6.7 Pointers as arguments

It looks like the argument is being altered, even though the real argument (the pointer) is unchanged.

When a function needs to alter its arguments, it should be designed to receive pointers, and when it is called, pointers should be passed.

The function is then able to use those pointers to alter the variables underlying them.

6.8 Case study

Write a function that reads integers until it obtains one in the range given by its first two arguments. When a suitable value is read, it stores that value using its third argument, and returns the predefined constant `READ_OK`. If no suitable value is located, the predefined constant `READ_ERROR` should be returned.

- `readnum.c`

Chapter 6 – Summary

- Variables can be local to a function, or global across a program; the rules of scope determine which variables can be accessed.
- Static local variables retain their values between calls.
- All data is stored in memory, and accessed via addresses.
- Pointer variables store addresses.
- Pointer arguments allow functions to “reach out” and alter variables outside their scope.