Introduction to C

Peter Schoener

Contents

Background — Why did C arise?

- Developed in the late 60s – early 70s
- Was first used to reimplement Unix
- Designed to be [more] human-readable [than Assembly]
- Uses syntax that maps simply to machine instructions
- Gives low-level memory access

Background — Why do we want to use C?

- Does most tasks as fast as they can be done
- Grants a lot of control over what commands are executed
- Easy to compile/write a compiler for
- Portable to almost any system
- Follows Unix conventions
- Relevant for legacy code maintenance
- Allows use of system libraries

Background — What is C?

- Statically, sort of strongly typed
  - static type checking encourages you to keep track of and use only compatible types
  - however, type coercion is allowed because sometimes it is efficient
- Procedural — separates code into functions
- Imperative — executes commands sequentially
- Low-level — relatively little syntactic sugar
- Compiled — designed to be compiled and potentially optimized before execution

Crash Course

- Types
- Functions
- Operators
- Flow Control
- Preprocessor
- Standard Library

Types — Primitive Data Types

- Originally, C had only `chars`, `ints`, `floats`, and `doubles`
- `char` is at least 8 bits wide, meaning it can store any ASCII character
- `int` is at least 16 bits wide
- `float` is at least 16 bits wide
- `double` is at least 32 bits wide
- `char` and `int` can be interchanged for the most part
  - allows easy conversion to/from character/control codes
  - on some systems, allows EOF to be represented

Types — Primitive Data Types

- The first standardized version added `unsigned`, `short`, and `long` modifiers
- Since casting between character and integral types is guaranteed, this means that `unsigned chars` exist, though they are usually not used
- `unsigned char` is, however, a common substitute for `bool`; `bool` is not a C type (sort of)
- everything that is 0 when interpreted as an integral type of equal width is `false`, everything else is `true` (one caveat later)
- variables can have `static` and `const` attributes

Types — Primitive Data Types

- Some types are guaranteed starting with C99 (in standard libraries, not the language itself)
  - specific width integral types
  - wide characters
  - booleans
- Some additional types might be defined depending on the platform
  - even wider arithmetic types
  - Unicode characters
  - OS resource handles

Types — User-defined Types

- C is not an object-oriented language, though many languages have extended C syntax to allow objects
- The same functionality is achieved with `structs`
- `structs` simply associate a group of variables with each other
- `structs` can be anonymous
- the `typedef` keyword allows the user to create aliases of any type
- Some syntactic sugar: `enums` generate constants that can be used in place of magic numbers

Types — Pointers

- Pointers are locations of data
  - primitives
  - structs
  - functions
  - other pointers
- Pointers can by typed according to what they point to
- Arrays in C are just the pointers to the beginnings of the data
- Every array is a pointer, but not every pointer is an array

Types — Pointers

- However, there is also a special generic (`void`) pointer type
  - `void` pointers are guaranteed to be large enough to point to anything in the address space
- `NULL` is a pointer that is defined in the standard library
  - stands for a memory address guaranteed to be invalid
  - in practice, usually 0x0
  - falsey

Types — Strings

- There is no built-in or library-defined string type
- Strings are just character arrays terminated by a `NUL` character
- Strings are not automatically sized; you are in charge of making sure your strings are wide enough
- This is where most C bugs come from!
  - missing terminator
  - out of bounds access/segmentation faults

(Types) — Functions

- C does not allow function overloading, however there are some tricks that can be used to simulate it
  - preprocessor macros, though this can get messy quickly
  - variadics, which basically pass a linked list as the argument
- Functions must be declared before use, just like variables
- Functions can be pointed to, and thus passed as arguments to other functions
- A function's type includes its return type and the types of all its arguments

(Types) — Functions

- The entry point is the `main` function, which has a few possible signatures in the standard
- However, implementations are allowed to define additional possible signatures
- Some platforms do not adhere to the standard, or do hacky things to get around it! cf. `WinMain`

Operators — Logic

- Logical AND (`&&`)
- Logical OR (`||`)
- Logical NOT (`!`)
- Equality (==, <=, >=)
- Inequality (!=, <, >)

Operators — Arithmetic and Assignment

- Arithmetic (+, -, /, *, %)
  - division on integral types is floor division
  - % is commonly called the modulo operator, but is technically the remainder operator
  - `ldiv` standard function returns floor division and remainder at the same time, potentially using only one instruction
  - ++ and -- shorthand
- Assignment (=)
  - assignment can be combined with an arithmetic or bitwise operation
  - assignment is an expression which returns the value assigned

Operators — Bitwise

- Bitwise AND (&)
- Bitwise OR (|)
- Bitwise XOR (^)
- Bitwise COMPL (~)
- Shift left (<<)
  - moves bits to the left, padding with zeroes
- Shift right (>>)
  - on signed types with a set high bit, the “new” bits are implementation-defined

Flow Control

- `if` and `while` work as you would expect
- C does not have iterators, so `for` works with index syntax only
- `do` can be combined with for and while
- `switch` allows you to match a variable to [integral] values
- `break` and `continue` work as you would expect in a loop, or allow you to leave a `switch` statement
- `goto` is present, but now generally discouraged

Preprocessor Directives

- Prefixed by #
- `include` copies headers and definitions from another file so that they are accessible
- `define` can be used for relatively naïve find and replace
  - often used for constants, though these are then untyped!
  - sometimes makes sense to use `typedef` or `const` instead
- `ifdef` and `ifndef` can be used to check for other macros

Standard Library — stdlib.h

- Defines functions for basic system interactions
  - heap memory management
  - process exiting
- Some common operations
  - basic integer arithmetic
  - random number generation
  - binary search
  - quicksort

Quick Aside — Heap Memory

- `malloc` allocates memory
- `calloc` allocates and clears memory
- `realloc` reallocates memory
- `free` frees memory
- `sizeof` gets the size of a variable or type

ALWAYS remember to free heap memory when you're done with it!

Standard Library — stdio.h

- C follows the POSIX paradigm of treating files and streams as the same thing
- Defines a file handle type (`FILE*`) and functions to manipulate it
- Because of the hardware that C was initially designed for (tapes), C file handling functions are those that would be efficient on tapes
  - those are generally also efficient on modern storage media, but do have some quirks
  - read operations generally work based on the number of bytes to be read — this goes hand in hand with manual string size management
  - the stream pointer can be moved by a given number of bytes, as on a tape

Standard Library — stdio.h

- Handle mode must be specified when opening the file — this is more a system limitation than a language one
- I/O is generally buffered, but several things are specified as flushing the buffers
- Just like with memory deallocation, file handles must be closed before the program exits
  - except standard streams: `stdin`, `stdout`, `stderr` (closing these is undefined behavior)

Standard Library — string.h

- String and array operations
  - remember, strings are nothing more than char arrays, and chars in C are just bytes!
  - operations for moving whole blocks of memory are therefore in the string library
- Standard string operations
  - concatenation
  - searching
  - comparison
  - …

Hello, World!

- I’ve bored you long enough — let’s get right into some coding examples!

Now it’s your turn!

- get the first command line argument that doesn’t begin with a dot
- attempt to open the file
- if successful, count the number of bytes in the file
- else create the file

on POSIX: compile with `gcc -o fsz main.c`

on Windows: probably best to use Visual Studio

Useful Libraries — posix.h, Windows.h

- System-level interactions
  - changing environment state
  - reading status of other programs
  - drivers
  - …
- System-specific functionality
  - threading
  - file system traversal
  - …

Useful Libraries — libcurl

- Internet access
- Bindings exist for other languages
- However, curl is written entirely in C
  - idiomatic C bindings
  - granular control
- Nearly as platform-agnostic as C itself

Useful Libraries — curses/ncurses/PDcurses

- Terminal graphics
- Windowed graphics are more easily handled with OOP (though it can be done with structs — just look at GTK+!), but terminal graphics only require simple function calls
- Again, bindings exist for other languages, but the paradigm fits C elegantly

(Useful Libraries) — Boost

- C++ library
  - makes extensive use of C++ language features, so does not work with pure C
  - however, it can be quite useful
- Cross-platform (modern Windows and most POSIX)
- Takes over a lot of functionality that would generally be in the system libraries
  - threading
  - file system operations
  - some graphics
- DSA that are usually covered by standard libraries
  - however, some of these are now covered by C++!

(Useful Libraries) — Tensorflow

- Again, not a pure C library
- Underlying code written in C++
  - sadly, only the Python bindings are considered stable
  - however, the C++ API is for the most part stable enough by now
  - again, granular control
  - some C/C++ paradigms are eschewed because the Python bindings are the main focus
- Machine learning
- GPU-based/parallel linear algebra

C++ — Additional Features

- Full-fledged object system (inheritance, polymorphism, generics)
- Built-in types expanded on and more narrowly standardized
- Function and operator overloading
- More strict type system
- More strict function signature enforcement
- More expansive standard library (while separately retaining C libs)
- Allows mixed source code via `extern`
- Some platform-dependent features now standard (e.g. threading)

Questions?

- Comments?
- Concerns?
- Wagers?
- Threats?

SfS Stammtisch starts immediately following this.

More Info

After years I still can't always remember the nuances of the standard library, and this is a good reference: http://www.cplusplus.com/reference/clibrary/

If you want to get into really pedantic stuff, free copies of the standards can be found here:

http://port70.net/~nsz/c/

The K&R book is still considered by many to be the best reference/guide to learning C, so if you're looking for a book that's a good place to start