APT Session 4: C

Laurence
Tratt

KING'S College LONDON

Software Development Team
2018-11-09

What to expect from this session

1. C.

Prerequisites

1. Install either GCC or LLVM/clang onto your computer. Most Unixes will have either/both installed by default, or as easy-to-install packages.

Prerequisites (Windows)

Windows users may find these instructions (courtesy of Sam White and Lukas Diekmann) useful:

1. Download the MinGW web installer.
2. Launch the installer, hit 'Install', then 'Continue' (leave the installation directory as the default, `C:\MinGW`). The installer will now download the files necessary. Once complete, hit 'Continue'.
3. The MinGW Installation Manager will now launch. Right-click 'mingw32-base-bin' and select 'Mark for Installation'. Now, select 'Apply Changes' from the 'Installation' menu.
4. Hit 'Apply'. MinGW will now download and install the base package. This may take 5–10 minutes, depending on your download speed. Once finished, you can close both windows. MinGW is now installed.
5. You can now find gcc in `C:\MinGW\bin`. You should add this directory to your PATH to make development easier (go to Windows Settings; search for 'env'; select 'Edit environment variables for your account'; under 'System Variables', edit 'PATH'; click 'New' and add `C:\MinGW\bin` into the text box).

Prerequisites (OS X)

On recent OS X, executing `xcode-select --install` at the command-line should install GCC and clang.

- C is a low-level programming language initially designed as the implementation language of Unix.

C

- C is a low-level programming language initially designed as the implementation language of Unix.
- Often described as a ‘high-level assembler’: it’s a way of writing code that’s fairly close to the machine while still being fairly readable for humans.

C

- C is a low-level programming language initially designed as the implementation language of Unix.
- Often described as a ‘high-level assembler’: it’s a way of writing code that’s fairly close to the machine while still being fairly readable for humans.
- The most practical way to get an insight into what the CPU is ‘really doing’; and still nearly always the fastest programming language.

C

- C is a low-level programming language initially designed as the implementation language of Unix.
- Often described as a ‘high-level assembler’: it’s a way of writing code that’s fairly close to the machine while still being fairly readable for humans.
- The most practical way to get an insight into what the CPU is ‘really doing’; and still nearly always the fastest programming language.
- First usable version debuted in 1973.

C

- C is a low-level programming language initially designed as the implementation language of Unix.
- Often described as a ‘high-level assembler’: it’s a way of writing code that’s fairly close to the machine while still being fairly readable for humans.
- The most practical way to get an insight into what the CPU is ‘really doing’; and still nearly always the fastest programming language.
- First usable version debuted in 1973.
- Still used for writing operating systems, programming languages critical utilities (e.g. I wrote extsmail in C), embedded systems etc. etc.

Versions of C

- C++ is more widely used for new software. Is it still worth learning C?

Versions of C

- C++ is more widely used for new software. Is it still worth learning C?
- Yes, because C++ is a superset of C: nearly everything that's valid C is valid C++ (but not the other way around).

Versions of C

- C++ is more widely used for new software. Is it still worth learning C?
- Yes, because C++ is a superset of C: nearly everything that's valid C is valid C++ (but not the other way around).
- As that suggests, C++ is a separate language that adds many extra things to C; too complex for my tastes.

Versions of C

- C++ is more widely used for new software. Is it still worth learning C?
- Yes, because C++ is a superset of C: nearly everything that's valid C is valid C++ (but not the other way around).
- As that suggests, C++ is a separate language that adds many extra things to C; too complex for my tastes.
- C is still actively (if slowly) developed: new versions in '99 and '11. Few changes in C11 of interest to us, so we'll use C99.

The basics

C is syntactically *very* similar to Java. Major immediate differences:

- No classes or objects, just top-level functions and `structs`.
- Memory must be allocated *and* freed manually.

The basics

C is syntactically *very* similar to Java. Major immediate differences:

- No classes or objects, just top-level functions and `structs`.
- Memory must be allocated *and* freed manually.

Other useful things to know:

- The main method in C is `int main(int argc, char **argv)`; the return value is returned to the shell. `return 0;` means 'I finished successfully'.
- `#include <stdio.h>` is similar to a Java import. It brings the `printf` function which prints a string to screen into scope.

The basics

C is syntactically *very* similar to Java. Major immediate differences:

- No classes or objects, just top-level functions and `structs`.
- Memory must be allocated *and* freed manually.

Other useful things to know:

- The main method in C is `int main(int argc, char **argv)`; the return value is returned to the shell. `return 0;` means 'I finished successfully'.
- `#include <stdio.h>` is similar to a Java import. It brings the `printf` function which prints a string to screen into scope.

Exercises:

1. Write a program which prints out `Hello world!` in C. Put it in a file `hello.c` and compile it with `gcc -Wall --std=c99 hello.c`. This will produce an `a.out` (Unix) or `a.exe` (Windows) file which can then be run.

Basic types

- C has some similar basic types to Java: `int` and `char`.
- For any type, one can make a *pointer type* with '`*`'.
- `char *` is a pointer to a sequence of characters of unknown length (i.e. roughly equivalent to a Java string).

Basic types

- C has some similar basic types to Java: `int` and `char`.
- For any type, one can make a *pointer type* with '`*`'.
- `char *` is a pointer to a sequence of characters of unknown length (i.e. roughly equivalent to a Java string).

Other useful things to know:

- C can't concatenate strings with `+`. But `printf` takes a *format string* as its first argument. '`%s`' in the format string is replaced with the matching parameter with strings passed as parameters. For example, `printf("%s %s!", "hello", "world")` prints out `hello world!`

Basic types

- C has some similar basic types to Java: `int` and `char`.
- For any type, one can make a *pointer type* with '`*`'.
- `char *` is a pointer to a sequence of characters of unknown length (i.e. roughly equivalent to a Java string).

Other useful things to know:

- C can't concatenate strings with `+`. But `printf` takes a *format string* as its first argument. '`%s`' in the format string is replaced with the matching parameter with strings passed as parameters. For example, `printf("%s %s!", "hello", "world")` prints out `hello world!`

Exercises:

1 Assign the string `Hello world!` to a variable of type `char *` then print out the contents of the variable.

Pointers

- Pointers are everything to C.

Pointers

- Pointers are everything to C.
- A pointer is an integer which references an *address* in memory.
- Pointers can be changed to other memory addresses.
- Think of a piece of string (the pointer) tied to a balloon (the bit of memory we're interested in). Multiple pieces of string can point to the same balloon. We can untie our string and tie it to another balloon if we want.

Pointers

- Pointers are everything to C.
- A pointer is an integer which references an *address* in memory.
- Pointers can be changed to other memory addresses.
- Think of a piece of string (the pointer) tied to a balloon (the bit of memory we're interested in). Multiple pieces of string can point to the same balloon. We can untie our string and tie it to another balloon if we want.
- e.g. if we have a variable `v` pointing to a `char *` array, we can access the first character either by explicit *dereferencing* with `*v` or using array syntax `v[0]` (the two are equivalent).
- `strlen` returns the length of a string. `#include <string.h>`

Pointers

- Pointers are everything to C.
- A pointer is an integer which references an *address* in memory.
- Pointers can be changed to other memory addresses.
- Think of a piece of string (the pointer) tied to a balloon (the bit of memory we're interested in). Multiple pieces of string can point to the same balloon. We can untie our string and tie it to another balloon if we want.
- e.g. if we have a variable `v` pointing to a `char *` array, we can access the first character either by explicit *dereferencing* with `*v` or using array syntax `v[0]` (the two are equivalent).
- `strlen` returns the length of a string. `#include <string.h>`

Exercises:

1 Assign the string `Hello world!` to a variable of type `char *` then print out each character of the string on a new line.

Arrays

- Pointers can be used as arrays. e.g. `char **` is an array of strings.
- **But** C arrays don't know their length. That must always be passed around separately.

Arrays

- Pointers can be used as arrays. e.g. `char **` is an array of strings.
- **But** C arrays don't know their length. That must always be passed around separately.
- `int main(int argc, char **argv)` is a great example. `argv` will contain `argc` number of elements, each a `char *`.
- C 'strings' are a pointer to a NUL-terminated region of memory. i.e. a sequence (of unknown length) of characters finishing with a char of value `0`. `strlen` manually walks the string each time!

Arrays

- Pointers can be used as arrays. e.g. `char **` is an array of strings.
- **But** C arrays don't know their length. That must always be passed around separately.
- `int main(int argc, char **argv)` is a great example. `argv` will contain `argc` number of elements, each a `char *`.
- C 'strings' are a pointer to a NUL-terminated region of memory. i.e. a sequence (of unknown length) of characters finishing with a char of value `0`. `strlen` manually walks the string each time!

Exercises:

1. Print out all the command-line arguments passed to your program. What is the first parameter?
2. Print out all the command-line arguments passed to your program along with the length of the arguments.

Functions

- C functions have a return type and 0 or more parameters (similar to Java).

Functions

- C functions have a return type and 0 or more parameters (similar to Java).

Exercises:

1. Write a ROT13 function which takes in a single `char` and returns its ROT13 equivalent. Test it with these cases `rot13('a') ≡ 'n'` and `rot13('n') ≡ 'a'`. You may assume only lower and upper case characters a–zA–Z will be passed.
2. Print out all command line arguments passed to your program after being ROT13ed.

Memory

- Heap memory is allocated in `n` bytes with `malloc(n)`. This returns `void *`, which can be cast to any pointer type you want (e.g. `char * c = malloc(n)`).
- Free memory with `free(c)`.
- You're responsible for freeing memory you allocated.
- `strcat(dst, cpy)` appends `cpy` to `dst`.

Memory

- Heap memory is allocated in `n` bytes with `malloc(n)`. This returns `void *`, which can be cast to any pointer type you want (e.g. `char * c = malloc(n)`).
- Free memory with `free(c)`.
- You're responsible for freeing memory you allocated.
- `strcat(dst, cpy)` appends `cpy` to `dst`.

Exercises:

1. What file do you need to include for `malloc`?
2. Concatenate all the command line parameters passed to the program into one string in memory. Print out a ROT13 version of the string, then the original string afterwards. Make sure you account for line endings when allocating memory!

IO

- Reading / writing to a file in C can be fiddly—need to do your own error handling.

IO

- Reading / writing to a file in C can be fiddly—need to do your own error handling.
- Can read input from `stdin` with `read(STDIN_FILENO, buf, len)` (defined in `<unistd.h>`) where: `STDIN_FILENO` is a magic number (on Windows you might need to explicitly change this to `0`); `buf` is a pointer to a buffer of `len` bytes.

IO

- Reading / writing to a file in C can be fiddly—need to do your own error handling.
- Can read input from `stdin` with `read(STDIN_FILENO, buf, len)` (defined in `<unistd.h>`) where: `STDIN_FILENO` is a magic number (on Windows you might need to explicitly change this to `0`); `buf` is a pointer to a buffer of `len` bytes.

Exercises:

1. Change your `rot13` function so that it leaves spaces, newlines (etc.) untouched (i.e. it only applies rot13 to a-zA-Z).
2. Read input from `stdin`, rot13 it, and print it to `stdout`.
3. What happens if you chain your program twice? i.e. `cat file | rot13_stdin | rot13_stdin`?

Post-session exercises

Try these (no particular order):

- You might find this 'C for Java programmer guide' useful.
- Writing insecure programs in C is easy: read a guide to secure programming in C (e.g. this).
- Some of the best written – despite, oddly, having few comments – C code can be found in Unix kernels.
  e.g. OpenBSD's kernel is a work of art.