C Programming
Introduction

Adam Sampson (based on material by Greg Michaelson)
School of Mathematical and Computer Sciences
Heriot-Watt University

Accessibility note

- If you're watching my lectures **without audio**...
- The PDF slides on Canvas are written as a visual-only version of the lecture content – I'll add extra text where necessary to capture anything that's only in audio here
  - Probably a better experience than the auto-captions!

- Please let me know about any accessibility difficulties

C overview

- Strict, strongly typed, imperative language for systems programming
- Combines high-level constructs with low level access to type representations and memory
- Reference: B. Kernighan & D. Ritchie, *The C Programming Language (2nd Ed)*, Prentice-Hall, 1988
- *The C Book*, second edition by Mike Banahan, Declan Brady and Mark Doran, Addison Wesley, 1991
  - https://publications.gbdirect.co.uk/c_book/

The history of C

- 1972-ish – K&R C – developed from BCPL and B, as a high-level language for the Unix operating system on PDP-11 computers; no formal standard
- 1989 – ANSI C – first international standard,
  with many extensions and new features
- 1999 – ISO C99; 2011 – ISO C11; 2018 – ISO C17
  Mostly backwards compatible with ANSI C
- ANSI C is still most widely used, but I'll mention changes from later standards as we go

Overview

- C looks like Java or JavaScript, but is **very** different
- Java and JavaScript have high-level objects
- C exposes low-level memory formats & addresses
- Must manage memory explicitly in C
- Relies on the programmer to avoid various kinds of errors, particularly to do with memory management and access

C lectures

- Compiling code, program layout, printing/reading data, expressions, arithmetic, memory addresses, control flow, precedence
- Functions, pointers, file IO, arrays
- Memory allocation, casting, masking, shifting
- Strings, structures, dynamic space allocation, field access
- Recursive structures, 2D arrays, union types

Compiling and running C programs

- We will use GCC – the GNU C Compiler
  - Free and open source software
  - Generates code for just about every conceivable platform

```
$ gcc -o name2 name1.c
```

  - Read C code from `name1.c`
  - Save compiled executable as `name2`

```
$ ./name2
```

  - Run the executable `name2` (`./` means current dir)

Separate compilation

```
$ gcc –c name1.c … nameN.c
```

- **Compile** object files `name1.o` … `nameN.o` only

```
$ gcc –o name name1.o … nameN.o
```

- **Link** object files `name1.o` … `nameN.o` and put executable in `name`

More GCC options

```
$ gcc -O2 …
```

- Generate optimised code (`-O0 -O1 -O2 -O3...` levels)

```
$ gcc -std=c99 …
```

- Specify C standard version to use (default will vary)

```
$ gcc -Wall …
```

- Enable all possible compiler warnings

```
$ gcc -g …
```

- Build code with extra information for debugging

- I would always recommend:

  ```
  gcc -g -O2 -std=c99 -Wall …
  ```

Running C programs

```
$ gcc name.c
```

- Forgot `-o name`? Default output name is `a.out`

```
$ man gcc
```

- View the **manual page** for the `gcc` command
- Detailed GCC manual is here: https://gcc.gnu.org/onlinedocs/

- Can often use `cc` instead of `gcc`
  - May be the proprietary C compiler for host OS/platform

Typical program layout

- `#include ...`
- `#define ...`
- `extern ...`
- declarations
- function declarations
- `int main(int argc, char *argv[])`
- `{ ... }`

Program layout

- Include **header files** that define reusable code – `name.h`

- `#include "..."`
  Looks in current directory

- `#include <...>`
  Looks in system directories
  - e.g. `<stdio.h>` for the standard library I/O definitions

- **`#include ...`**
- `#define ...`
- `extern ...`
- declarations
- function declarations
- `int main(...)`
- `{ ... }`

Program layout

- Macro definitions, e.g. for constants

- Names and types of variables/functions used in this file, but declared in other files that will be linked with this one

- `#include ...`
- **`#define ...`**
- **`extern ...`**
- declarations
- function declarations
- `int main(...)`
- `{ ... }`

Program layout

- Usually declare variables before functions
- `main` function with command-line argument count and array
- Return value indicates success (0) or failure (non-0)
- Declarations and statements terminated with a `;`

- `#include ...`
- `#define ...`
- `extern ...`
- **declarations**
- **function declarations**
- **`int main(...)`**
- **`{ ... }`**

`printf`

```
printf("text");
```

- A **standard library** function,
  which sends `text` to the display
- C strings can contain **escape characters**, e.g.
  - \n == newline
  - \t == tab

This is an example program:
you can download the full code
from the Canvas page for this lecture

`hello.c`

Printing a string

Printing a string

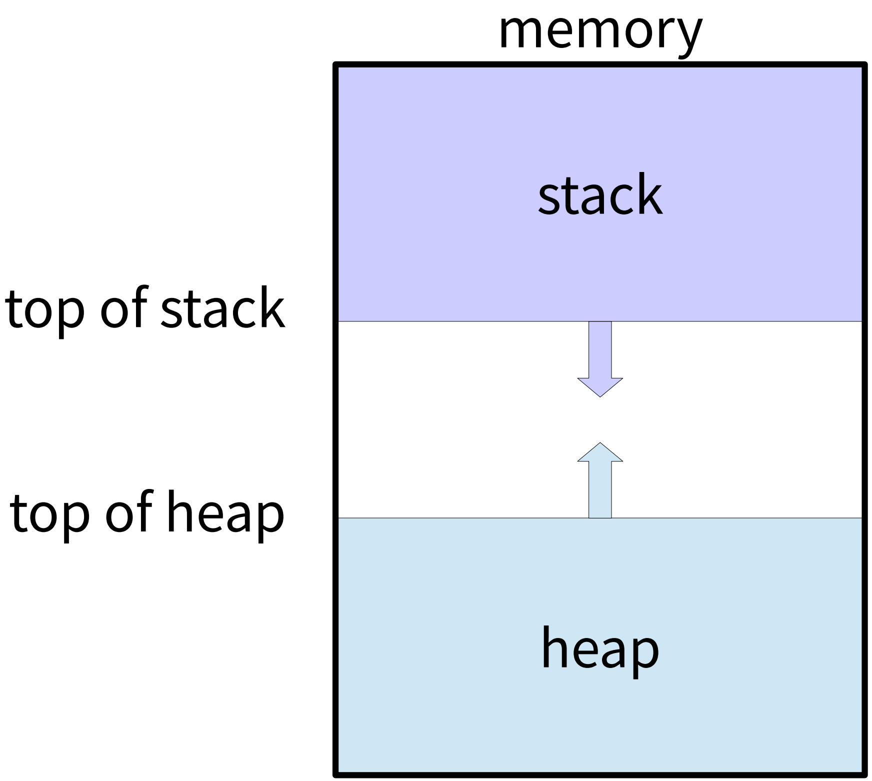
```
#include <stdio.h>

int main(int argc, char *argv[]) {
   printf("hello\n");
   return 0;
}


...
$ gcc -o hello hello.c
$ ./hello
hello
$
```

Memory organisation

- BSS
  - Allocated by compiler
  - Global declarations
- Stack
  - Allocated automatically
  - Local declarations
  - (Some) function parameters
- Heap
  - Allocated by program at runtime
  - Similar to Java's new keyword
  - No garbage collection in C!



Declarations

- C's **primitive** (built-in) types include:
  - `char` – character
  - `int` – integer
  - `short` – short integer
  - `long` – long integer
  - `float` – single-precision floating point number
  - `double` – double-precision floating point number

- Actual sizes of these will depend on the platform

Declarations

- `type name;`
  - Allocates space for new variable of `type` called `name`
  - e.g. `int count;` or `float height;`
- Names can contain letters, digits, and _
  - … but must start with a letter
- By convention:
  - `lower_case` = variable name
  - `UPPER_CASE` = symbolic constant

Declarations

- Can group several declarations with the same type

```
type name1;
type name2;
type name3;
→
type name1, name2, name3;
```

Expressions may contain...

- Constants (number, string...)
- Variable `name` → value of variable from memory
  - value for that name may differ depending on what type context name is used in, because of automatic type conversion
- Unary/binary operators
- Function calls
  - No objects in C – so we have **functions**, not ~~**methods**~~

Constants

- Signed integer
  - `4231 -2579` (decimal)
  - `0x12AB34` (hexadecimal – leading `0x`)
  - `0755` (octal – leading `0`)
- Signed floating-point
  - `886.754`
  - `-3.9e11` (means -3.9 x 10^{11})
- Character: `'letter'`
  - `'a' '\n'` (ASCII value of character)

Operator expressions

- Unary operators
  - *op exp* (e.g. `−42`)
    - evaluate *exp*
    - apply *op* to value
- Binary infix operators
  - *exp1 op exp2* (e.g. `score + 42`)
    - evaluate *exp1* and *exp2* (in either order)
    - apply *op* to values

Arithmetic

- Unary minus (negate): −
- Binary infix
  - + == add
  - − == subtract
  - ⋆ == multiply
  - / == divide
  - % == integer modulo/remainder
    - Different behaviour with negative arguments than some other languages

Arithmetic

- (. . .) – brackets
- Order of precedence:
  - (. . .) before...
  - unary – before...
  - * or / or % before...
  - + or – before ...
  - function call
- Operators associate from left to right

Arithmetic

- Mixed-mode arithmetic permitted, working at the maximum precision needed automatically
- For a binary operator:
  - `char` and `short` are **promoted** to `int`
  - `float` is converted to `double`
  - If either operand is `double` then the other converts to `double`
  - If either operand is `long` then the other converts to `long`
- (Simplified a bit – see the book!)

Function calls

- A function called as:
  `name(exp1, exp2 ... expN)`

- Evaluates actual parameters `exp1` to `expN`
  - Evaluated in **arbitrary** order
  - Values are passed to the function via CPU registers and/or stack (depending on platform)

- Result of function execution is returned

`printf` again

```
printf("format", exp1, exp2 ... expN);
```

- Prints a string, expanding **format characters** in the string into the values of expressions exp1...expN in order
  - `%d` == decimal integer
  - `%f` == floating point
  - `%x` == hexadecimal
  - `%s` == string

`printf` again

- `printf` has a variable number of arguments
  - Must have one format character for each argument
  - Any non-format information in string is displayed as text

```
int answer = 42;
printf("Answer is %d decimal, %x hex\n",
       answer, answer);
```