C

- created in the early 70’s
- on PDP11 with 24 KB of RAM (early UNIX: 12 KB)
- for the development of UNIX

Outcome:

- minimalist → easy to learn and to implement
- pragmatic → trumps aesthetics
- portable and close to the hardware
- core language has no I/O and no dynamic memory management

... in many ways ideal for embedded systems.
Programming in C

C will put you in full control and not get in your way—no matter how big the mistake is you are about to make.

- For historical reasons there are some subtle pitfalls
- For efficiency reasons there are no safety nets at runtime
- It has a terse syntax that is not very restrictive—but has little redundancy!

Very likely the compiler will find an interpretation for your code.
C will put you in full control and not get in your way—no matter how big the mistake is you are about to make.

▶ For historical reasons there are some subtle pitfalls
▶ For efficiency reasons there are no safety nets at runtime
▶ It has a terse syntax that is not very restrictive—but has little redundancy!

Very likely the compiler will find an interpretation for your code.
What does this do?

- Recognizes every printable ASCII character of handwritten text in an image

Comments from the author:

- “Newcomers to C find it hard to learn all those different ways to control flow: for, while, if, do, goto, continue, break and heaven knows what else! So, in this program we only use for, so absolute beginners can get into the code straight away.”

- “main() is the most useful function in all of C—so it is a mystery to the author why most programs use it only once. Here we use it over and over for maximum benefit.”

http://www.ioccc.org/2013/cable2/hint.html
Looks like $A$ but does $B$

Ideal to smuggle malicious code past a review! $\rightarrow$ backdoors

If bugs make it past the review... why would it stop some carefully crafted code?

The Underhanded C Contest:

“C is an ideal language for this... C lets you overwrite stack entries, screw up function pointers, and poison all data at the bit level. C nods encouragingly as you attempt to execute a floating point array.

In terms of enforcing program correctness, your typical C compiler is basically the two guards from Swamp Castle in Monty Python and the Holy Grail.”

http://www.underhanded-c.org/
Is something like this still useful?

IEEE Spectrum: The 2017 Top Programming Languages

<table>
<thead>
<tr>
<th>Language Rank</th>
<th>Types</th>
<th>Spectrum Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Python</td>
<td><img src="image1" alt="world" /> <img src="image2" alt="desktop" /></td>
<td>100.0</td>
</tr>
<tr>
<td>2. C</td>
<td><img src="image3" alt="mobile" /> <img src="image2" alt="desktop" /> <img src="image4" alt="chip" /></td>
<td>99.7</td>
</tr>
<tr>
<td>3. Java</td>
<td><img src="image1" alt="world" /> <img src="image3" alt="mobile" /> <img src="image2" alt="desktop" /></td>
<td>99.5</td>
</tr>
<tr>
<td>4. C++</td>
<td><img src="image3" alt="mobile" /> <img src="image2" alt="desktop" /> <img src="image4" alt="chip" /></td>
<td>97.1</td>
</tr>
<tr>
<td>5. C#</td>
<td><img src="image1" alt="world" /> <img src="image3" alt="mobile" /> <img src="image2" alt="desktop" /></td>
<td>87.7</td>
</tr>
</tbody>
</table>

“...the languages currently in demand by recruiting companies, C comes out ahead of Python by a good margin.”

IEEE: Top Languages
Is something like this still useful?

January 2018

C Language of Year 2017

<table>
<thead>
<tr>
<th>Programming Language</th>
<th>Ratings</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>14.215%</td>
<td>-3.06%</td>
</tr>
<tr>
<td>C</td>
<td>11.037%</td>
<td>+1.69%</td>
</tr>
<tr>
<td>C++</td>
<td>5.603%</td>
<td>-0.70%</td>
</tr>
<tr>
<td>Python</td>
<td>4.678%</td>
<td>+1.21%</td>
</tr>
<tr>
<td>C#</td>
<td>3.754%</td>
<td>-0.29%</td>
</tr>
</tbody>
</table>

https://www.tiobe.com/tiobe-index
When to use C

Where C is used, there is typically no alternative:

- if low-level hardware access is required
  - driver, privileged CPU mode, configuration of hardware (e.g. memory management unit)
- when runtime resources are critical
- when realtime behaviour is critical
When to use C

- for very low capability systems
  - C can run on systems without RAM!
- software for new hardware
  - C is often the first language available
- highly portable programs
  - under the right restrictions (ANSI89, POSIX)
  - command line only
When to use C

C is also great for short bits of code that benefit from optimisation:

1. Write everything in Python
2. Profile
3. Replace the hotspots with C
C in Embedded Systems

- Typically a small amount of code
- Often large number of deployments

⇒ C is a worthwhile trade-off
C in Practice
Programming in which C?

- The most widely supported: C89
- At present in common use: a subset of C99

Portability

- from one processor type to another
  - same series: ATmega644P → AT90USB1286
  - different architecture: AT90USB1286 → ARM CortexM0
- from board to board: Arduino → LaFortuna
- from compiler to compiler: AVRstudio → gcc → clang
Programming in which C?

- The most widely supported: C89
- At present in common use: a subset of C99

Portability

- from one processor type to another
  - same series ATmega644P → AT90USB1286
  - different architecture AT90USB1286 → ARM CortexM0
- from board to board Arduino → LaFortuna
- from compiler to compiler AVRstudio → gcc → clang
Translation Units

Recall cross-compilation lecture:

- **Scopes:** program, file, function/block
- Programs are organised in files
- Files can be compiled independently (.c → .o)

This was important for C's success; Linux kernel ≈ 8 M lines of C in ≈16 k files.
Header files: *.h

- Contains declarations for exposed functions and variables
- Included with preprocessor commands
  - #include <stdio.h>, #include "my.h"
  - in source file that holds corresponding definitions
  - in source files that make use of the declared functions

...serves also as documentation.
Things to be aware of from the start

- Unspecified Behaviours
- Undefined Behaviours
- Unexpected Behaviours
Things to be aware of from the start

- Unspecified Behaviours
- Undefined Behaviours
- Unexpected Behaviours

Programmer makes incorrect assumptions.

⇒ Verify your assumptions!
Values or behaviours unspecified

- E.g., adapt to hardware, e.g. size of int
  - Know what your compiler does: limits.h
  - Explicitly direct it: stdint.h
- Freedom for the compiler design
  → Implementation-Defined Behaviour
  see https://gcc.gnu.org/wiki/avr-gcc

⇒ Know what happens!

Behaviours undefined

- error for which the standard does not prescribe any action
- it is not clear what will happen
- arbitrary behaviour at arbitrary time

⇒ Don’t go there!
- Values or behaviours unspecified
  - E.g., adapt to hardware, e.g. size of int
    - Know what your compiler does: limits.h
    - Explicitly direct it: stdint.h
  - Freedom for the compiler design
    → Implementation-Defined Behaviour
    see https://gcc.gnu.org/wiki/avr-gcc

  ⇒ Know what happens!

- Behaviours undefined
  - error for which the standard does not prescribe any action
  - it is not clear what will happen
  - arbitrary behaviour at arbitrary time

  ⇒ Don’t go there!
Things to be aware of from the start II

- In some contexts Zero is special:
  - Any value that is not 0 is true
  - A zero byte (’\0’) marks the end of a string
  - A pointer to the address 0 (NULL) is invalid

Testing for zero is generally fast on any hardware.
Things to be aware of from the start III

- The rules for type promotions and conversions
  - see links on the notes page under “Reading”
- The operator precedence
↑ higher precedence

<table>
<thead>
<tr>
<th>()</th>
<th>[ ]</th>
<th>.</th>
<th>-&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>+</td>
<td>-</td>
<td>~</td>
</tr>
<tr>
<td>++</td>
<td>--</td>
<td>*</td>
<td>&amp;</td>
</tr>
<tr>
<td>(type)</td>
<td>sizeof</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>/</td>
<td>%</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>&gt;&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;=</td>
<td>&gt;=</td>
</tr>
<tr>
<td>==</td>
<td>!=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

right-to-left associative

↓ lower precedence

| =   | &   | = | |= | <<= | >>= | += | -= | *= | /= | %= |
|-----|-----|---|----|-----|-----|----|----|----|----|----|
Operator Precedence/Associativity

- You need to know this for programming in C
- You do not need to know it by heart for the exam
- Look it up when you need it
- But: do not sprinkle around parenthesis because you are too lazy to look it up

Parenthesis should indicate that the programmer deliberately overwrites the precedence rules—rather than indicating ignorance of the rules.