

# Imperative programming

## 1. Introduction

Zoltán Porkoláb

gsd@inf.elte.hu

<https://gsd.web.elte.hu>

# Goals of the subject

- Generic programming language concepts
- Learning the terminology
- Understanding programming paradigms
- Conscious use of languages
- Some programming skills
- Surviving in operating systems (especially in Linux)

# Why C?

## Mother Tongues

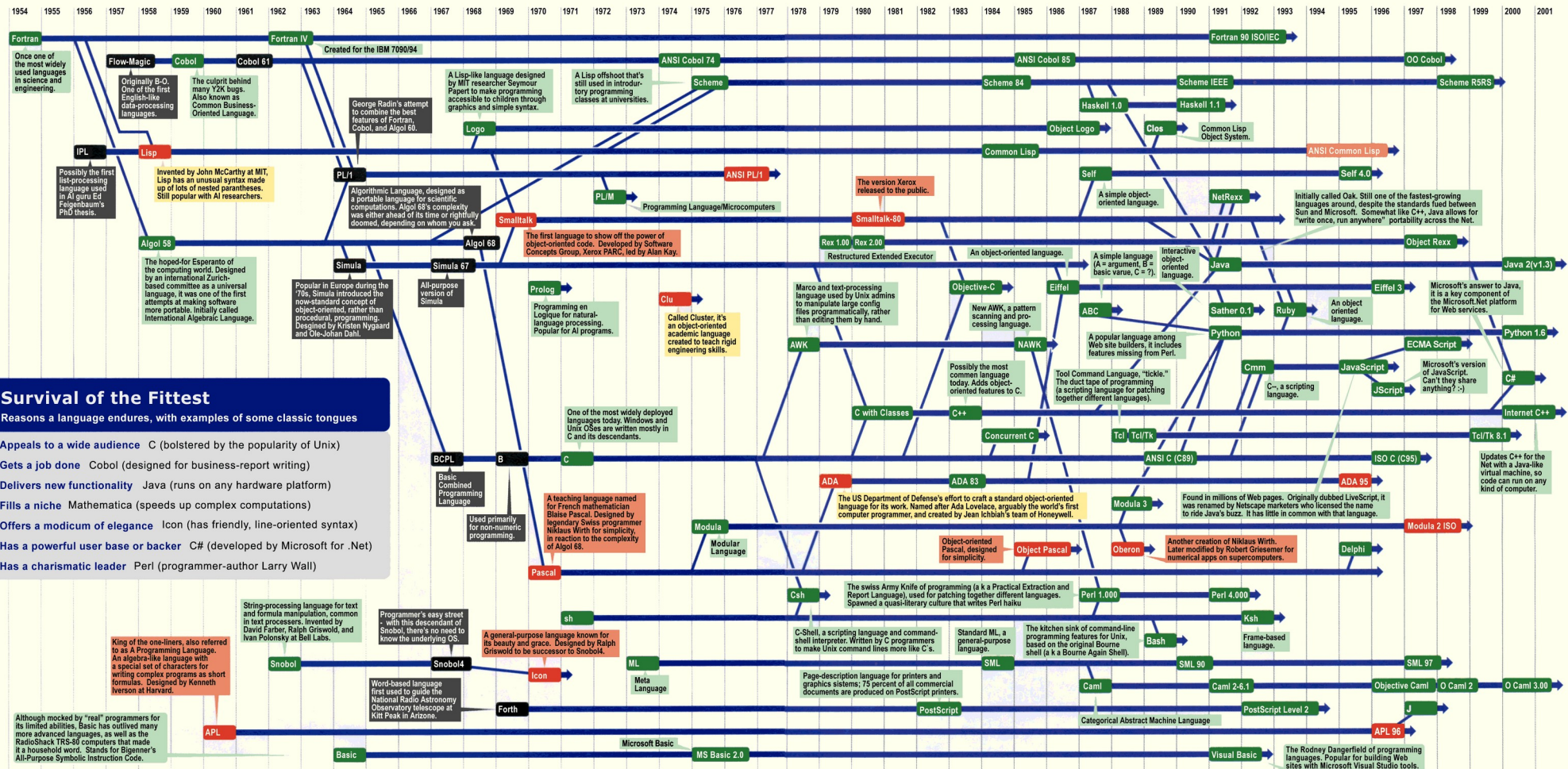
Tracing the roots of computer languages through the ages

Just like half of the world's spoken tongues, most of the 2,300-plus computer programming languages are either endangered or extinct. As powerhouses C/C++, Visual Basic, Cobol, Java and other modern source codes dominate our systems, hundreds of older languages are running out of life.

An ad hoc collection of engineers-electronic lexicographers, if you will-aim to save, or at least document the lingo of classic software. They're combing the globe's 9 million developers in search of coders still fluent in these nearly forgotten lingua frangas. Among the most endangered are Ada, APL, B (the predecessor of C), Lsp, Oberon, Smalltalk, and Simula.





















Code-raker Grady Booch, Rational Software's chief scientist, is working with the Computer History Museum in Silicon Valley to record and, in some cases, maintain languages by writing new compilers so our ever-changing hardware can grok the code. Why bother? "They tell us about the state of software practice, the minds of their inventors, and the technical, social, and economic forces that shaped history at the time," Booch explains. "They'll provide the raw material for software archaeologists, historians, and developers to learn what worked, what was brilliant, and what was an utter failure." Here's a peek at the strongest branches of programming's family tree. For a nearly exhaustive rundown, check out the Language List at [HTTP://www.informatik.uni-freiburg.de/Java/misc/lang\\_list.html](http://www.informatik.uni-freiburg.de/Java/misc/lang_list.html). - Michael Mendeno

1954	Year Introduced
Active: thousands of users	
Protected: taught at universities; compilers available	
Endangered: usage dropping off	
Extinct: no known active users or up-to-date compilers	
Lineage continues	



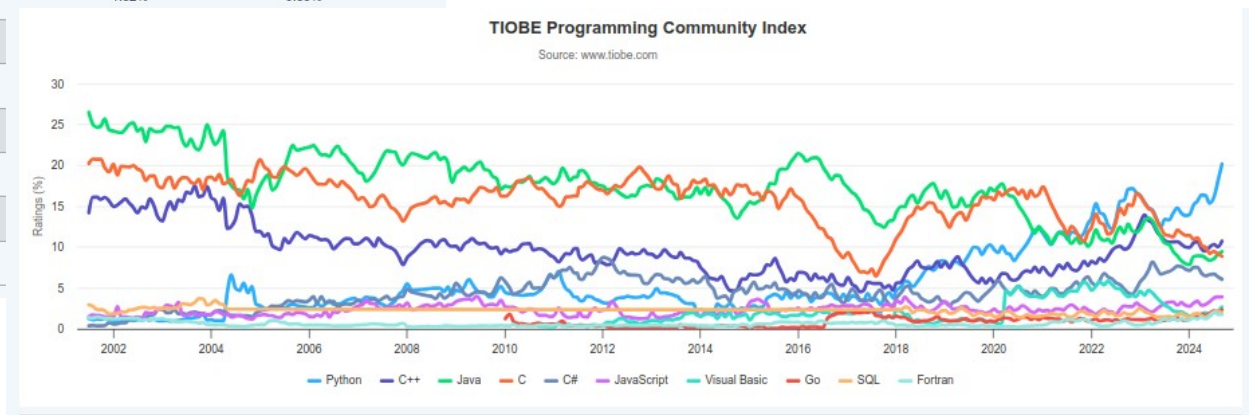
Sources: Paul Boutin; Brent Hailpern, associate director of computer science at IBM Research; The Retrocomputing Museum; Todd Proebsting, senior researcher at Microsoft; Gio Wiederhold, computer scientist, Stanford University

# Why C?

Sep 2024	Sep 2023	Change	Programming Language	Ratings	Change
1	1		 Python	20.17%	+6.01%
2	3	▲	 C++	10.75%	+0.09%
3	4	▲	 Java	9.45%	-0.04%
4	2	▼	 C	8.89%	-2.38%
5	5		 C#	6.08%	-1.22%
6	6		 JavaScript	3.92%	+0.62%
7	7		 Visual Basic	2.70%	+0.48%
8	12	▲	 Go	2.35%	+1.16%
9	10	▲	 SQL	1.94%	+0.50%
10	11	▲	 Fortran	1.78%	+0.49%
11	15	▲	 Delphi/Object Pascal	1.77%	+0.75%
12	13	▲	 MATLAB	1.47%	+0.28%
13	8	▼	 PHP	1.46%	-0.09%
14	17	▲	 Rust	1.32%	+0.35%
15	18	▲	 R		
16	19	▲	 Ruby		
17	14	▼	 Scratch		
18	20	▲	 Kotlin		
19	21	▲	 COBOL		
20	16	▼	 Swift		

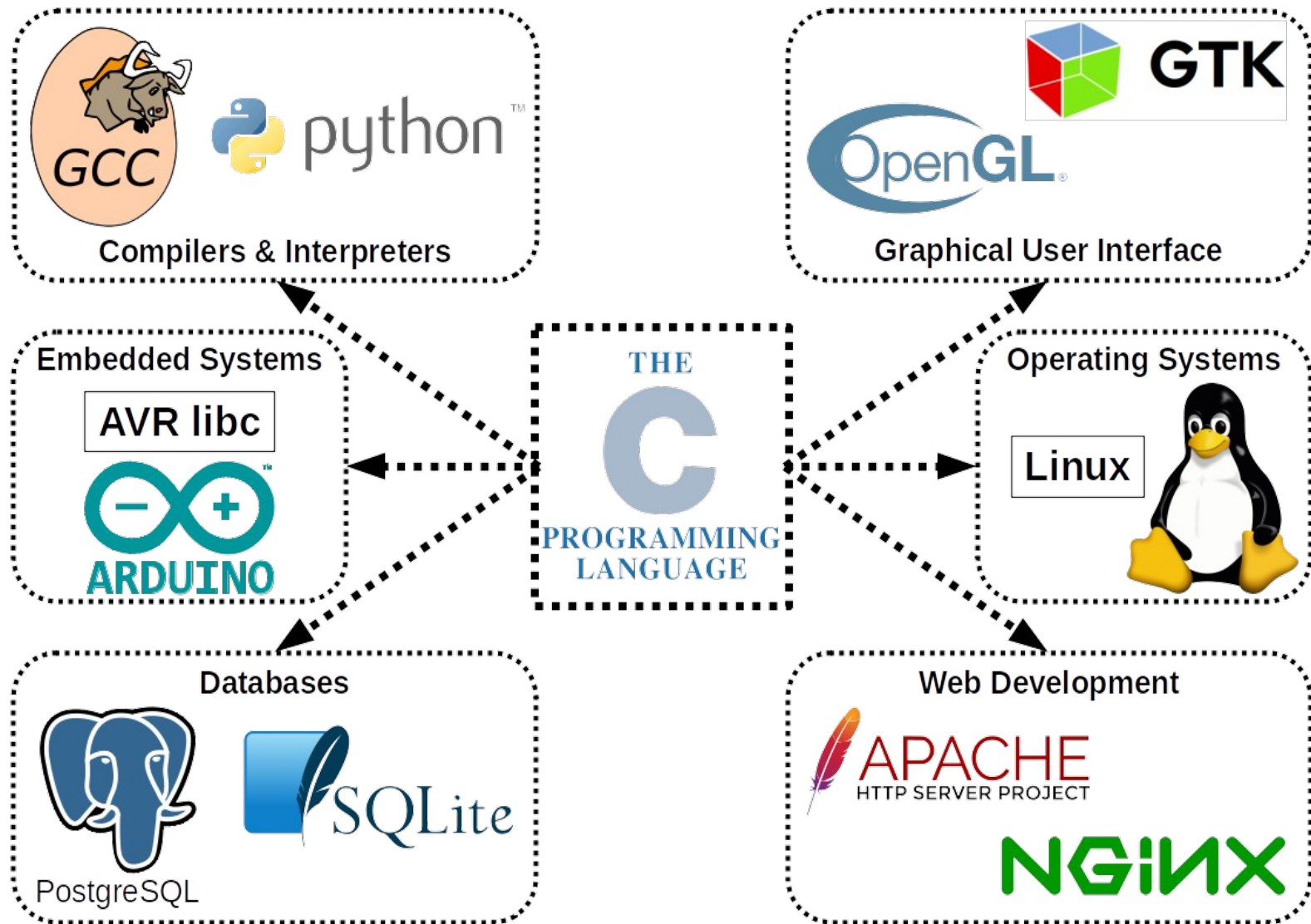
- TIOBE index started in 2001
- C was always in the top 4
- Currently the 4th most popular

<https://www.tiobe.com/tiobe-index/>





# Why C?



From Wikipedia

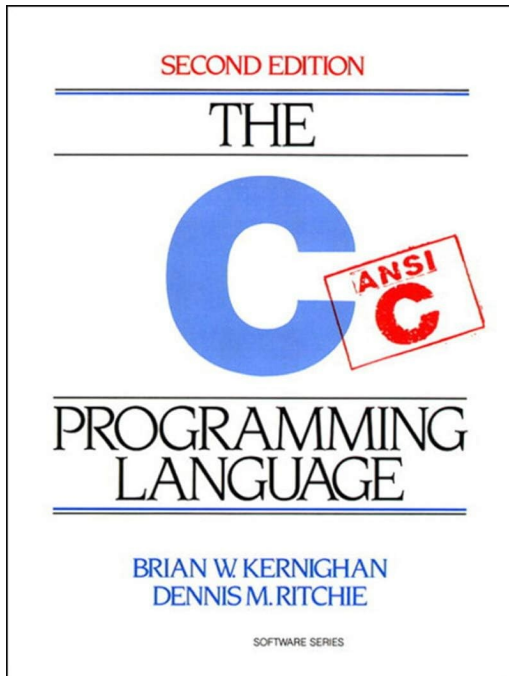
# Why C?

- (Relatively) high level programming language
- Small feature set
- Easy to learn – fits for one semester
- Good abstraction of hardware (pointers, memory handling)
- Portable programs
- Very efficient code generation
- The most popular language for embedded systems
- Compiler friendly – easy to port, available on all platforms

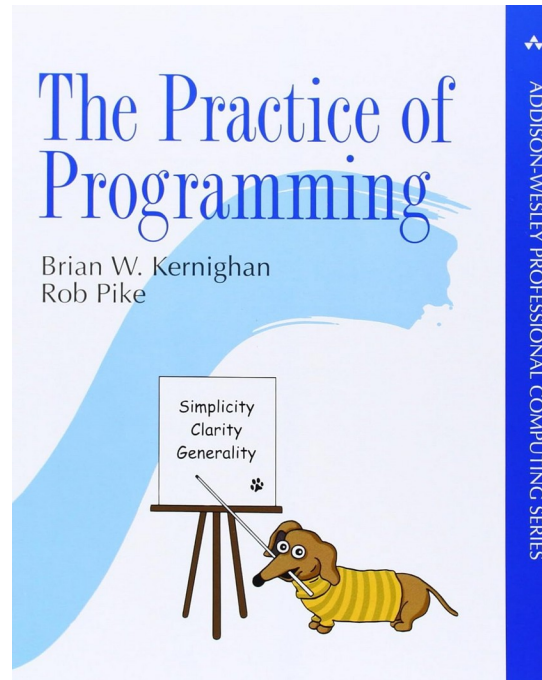
# Plan for the semester

- Introduction, programming paradigms, imperative programming
- Structure of C programs, static type system,
- Basic data types
- Operators, expressions, expression evaluation
- Statements
- Memory handling, arrays and pointers, pointer arithmetic
- Declarations, life and scope rules
- Functions and parameters
- Enum, struct, union and their usage
- Dynamic memory handling
- Encapsulation: separation of interface and implementation
- Practice, practice, practice ...

# Literature



Brian W. Kernighan, Dennis Ritchie:  
The C Programming Language, 2nd Edition.  
Prentice Hall (April 1, 1988).  
ISBN-13: 978-0131103627.



Brian W. Kernighan, Rob Pike:  
The Practice of Programming, 1st Edition.  
Addison-Wesley Professional Computing Series (April 1, 1999).  
ISBN-13: 978-0201615869.

<https://en.cppreference.com/w/c>



# Syntax/Semantics/Pragmatics

- Syntax
  - The proper grammar of the language
  - valid operators, how to separate statements, ...
- Semantics
  - The meaning of the syntactically correct programs
  - type system, conversion, copy of objects, overload resolution, ...
- Pragmatics
  - How to use the language in the best (safest) way
  - while/do-while/for, if-else/switch, use of pointers

# Programming paradigms

- High-level concept to conceptualize and structure the implementation
  - How to split the problem to smaller parts
  - How a programming language supports paradigm(s)
- Imperative
  - Procedural (Fortran, C, Pascal, Go, ...)
  - Object-oriented (Simula, Smalltalk, Java, Scala, Rust, ...)
- Declarative
  - Functional (Lisp, Haskell, Clean, Ocaml, ...)
  - Logic (Prolog, ...)
- Many other categories
  - Concurrent, generative, metaprogramming, visual, dataflow, ...
  - Multiparadigm

# Method of execution

- Interpreter (bash, Python, Perl, ...)
  - Processing the source code by statements
    - + Flexible, REPL (Read-Evaluate-Print-Loop), fast prototyping
    - Run-time errors harder to avoid, less optimization, slower execution
- Compilation + linking (Fortran, C, C++, Go, Rust, ...)
  - Compile to machine code, link with other machine code
    - + Many errors can be detected in compilation time, optimal executable
    - Slower development, harder debugging, executable is platform specific
- Mixed (Java, C#, Scala, ...)
  - Compile to some intermediate language and interpret that
    - + Intermediate code is portable
    - Requires pre-installed run-time system

# History of C

- The original machine
  - DEC PDP-11
  - 24K RAM
  - 12K used for OS



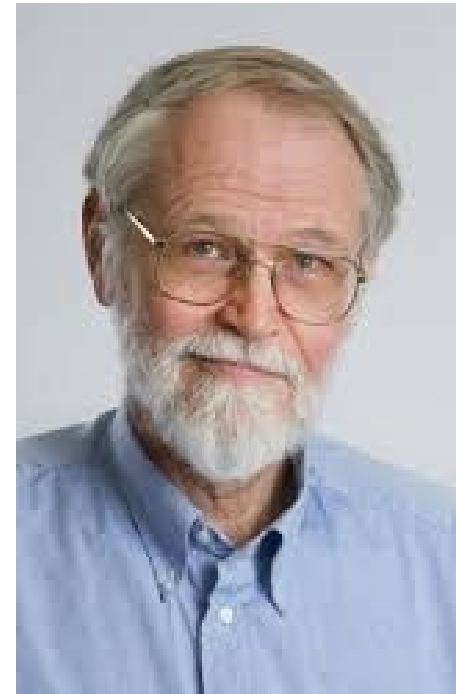
# History of C



Dennis Ritchie  
1941 - 2011



Ken Thompson  
1943 -



Brian Kernighan  
1942 -

# History of C



Bjarne Stroustrup  
1950 -

<http://www.wired.com/2012/06/beard-gallery/>



# History of C

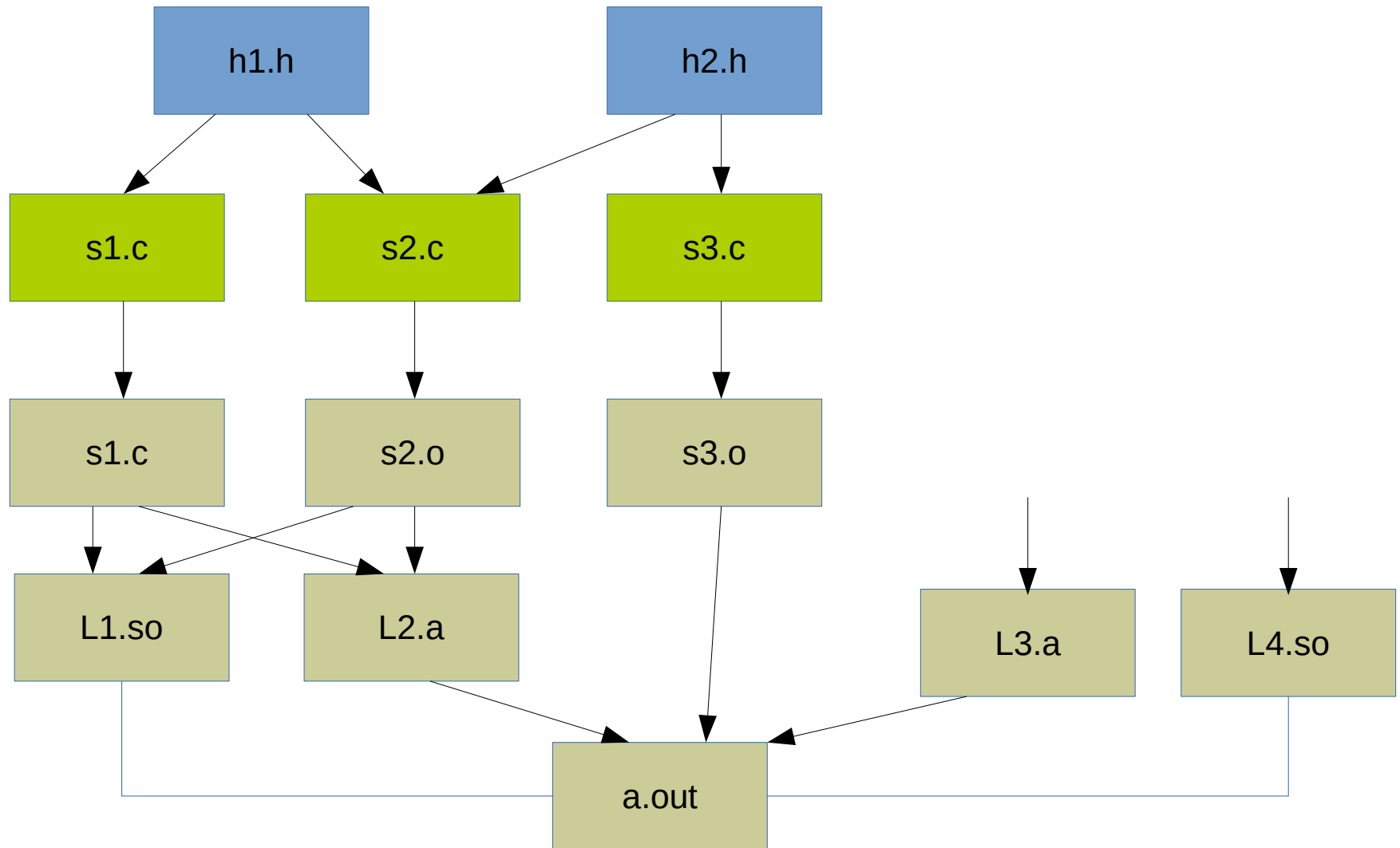
- 1967 Martin Richards develops BCPL in Cambridge
- 1969 Ken Thompson develops B (a simplified BCPL)
- 1969- Ken Thompson, Dennis Ritchie and others work on UNIX
- 1972 Dennis Ritchie develops C
- 1972-73 UNIX kernel is rewritten in C
- 1977 Johnsons Portable C Compiler 1977
- 1978 B. Kernighan & D. Ritchie: The C Programming Language book
- 1989 ANSI C standard (C90) (32 keywords)
- 1999 ANSI C99 standard (+5 keywords)
- 2011 ANSI C11 standard (+7 keywords)

# Hello.c

```
#include <stdio.h>

int main()
{
    printf( "hello world\n" );
    return 0;    /* not strictly necessary since C99 */
}
```

# Building the C program



# Hello.c

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    printf( "hello world\n" );
```

```
    return 0;    /* not strictly necessary since C99 */
```

```
}
```

```
$ gcc hello.c
```

```
$ ./a.out
```

```
hello world
```

```
$
```

# Hello.c

```
#include <stdio.h>
```

```
int main()  
{  
    printf( "hello world\n" );  
    return 0;    /* not strictly necessary since C99 */  
}
```

```
$ gcc hello.c
```

```
$ ./a.out
```

```
hello world
```

```
$
```

```
$ gcc -std=c11 -ansi -pedantic -Wextra hello.c -o hello.exe
```

```
$ ./hello.exe
```

```
hello world
```

```
$
```

# Compiling, linking

```
$ gcc hello.c
$ gcc -ansi -pedantic -Wall -W hello.c
$ gcc -std=c11 -ansi -pedantic -Wall -W hello.c
$ gcc -std=c11 -ansi -pedantic -Wall -W hello.c -o a.exe
$ gcc -s hello.c          # print assembly
$ gcc -E hello.c          # precompile only
$ gcc -c hello.c          # compile only
$ gcc hello.o             # link
$ gcc a.c b.o c.a         # compile a.c + link a.o b.o c.a
```



# Errors, warnings

```
#include <stdio.h>
int main()
{
    printf( "hello world\n" ) /* ; */
    return 0;
}
```

```
$ gcc hello.c
```

# Errors, warnings

```
#include <stdio.h>
int main()
{
    printf( "hello world\n" ) /* ; */
    return 0;
}
```

```
$ gcc hello.c
m.c: In function 'main':
m.c:6:28: error: expected expression before '/' token
$
```

# Errors, warnings

```
/* #include <stdio.h> */  
int main()  
{  
    printf( "hello world\n" );  
    return 0;  
}
```

```
$ gcc hello.c
```

# Errors, warnings

```
#include <stdio.h>
int main()
{
    printf( "hello world\n" ) /* ; */
    return 0;
}
```

```
$ gcc hello.c
hello2.c: In function 'main':
hello2.c:6:3: warning: implicit declaration of function
'printf'
$
```

# Program behavior

- Observable behavior
  - C language standard defines the behavior of the program as a black box
- Undefined behavior
  - Compiler can diagnose and give warning, but not always `ptr=NULL; *ptr;`
- Unspecified behavior
  - Multiple possible behavior, not necessary documented `f(a,b)`
- Implementation-defined behavior
  - Documented platform-specific unspecified behavior `sizeof(int)`
- Locale-specific behavior
  - Locale-specific `islower('á')`
- Strictly conforming programs:
  - Not undefined, unspecified or implementation-defined