Specifications

Part 3: Correctness

We are here

- Part 1: How computers works
  - Boolean logic, integers
  - Instructions
  - Memory
- Part 2: Software development
  - Compiling (clang, make, …)
  - Architectures, portability (ABIs, …)
  - Code management (regex, git)
- Part 3: Correctness
  - Specifications ← TODAY
  - Documentation, testing
  - Static & dynamic analysis, debugging
- Part 4: Performance
  - CPU pipelines, caches
  - Data structures
  - Parallel computation

A note about C

Why C?

- The C language has deep flaws

- but the C ABI is everywhere:
  - CPU and OS vendors define the ABI for C function calls
  - OS services are typically provided via C functions:
    - Win32 anxd WinRT (even though WinRT is C++)
    - MacOS’s Cocoa uses the Objective-C ABI (a superset of the C ABI)
    - Linux kernel ABI
  - almost all other languages support calling into C code

Why the C ABI?

The C ABI is simple:

- just functions and simple types: integer, pointer, `struct`
- no objects or methods:
    - What names do we give the symbols for the following?

      ```
      MyClass::myFunction(int type);
      MyClass::myFunction(OtherClass &c);
      ```

    - This?

      ```
      MyClass__method__int__myFunction
      MyClass__method__OtherClass_ref_myFunction
      ```

    - How do we call them? Like this?

      ```
      MyClass__method__int__myFunction(MyClass *self, int type);
      MyClass__method__OtherClass_ref_myFunction(MyClass *self, OtherClass *c);
      ```

- no exceptions

Other ABIs

- There are multiple C++ ABI specifications
    - but they change over time (no “stable” ABI)
    - even across versions of the same compiler
- There is no Rust ABI specification

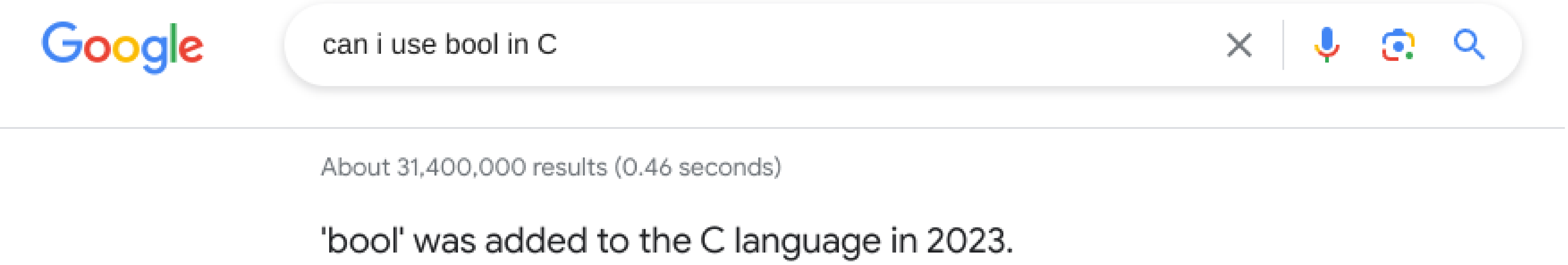
Specifications

What is even the C language?

```c
bool is_zero(int i)
{
    return i == 0;
}
```

```
clang -O3 -c -o is_zero.o is_zero.c
```

```
is_zero.c:1:1: error: unknown type name 'bool'
bool is_zero(int i)
^
1 error generated.
```



can i use bool in C

About 31,400,000 results (0.46 seconds)

'bool' was added to the C language in 2023.

```c
bool is_zero(int i)
{
    return i == 0;
}
```

```
clang -O3 -c -std=c2x -o is_zero.o is_zero.c
```

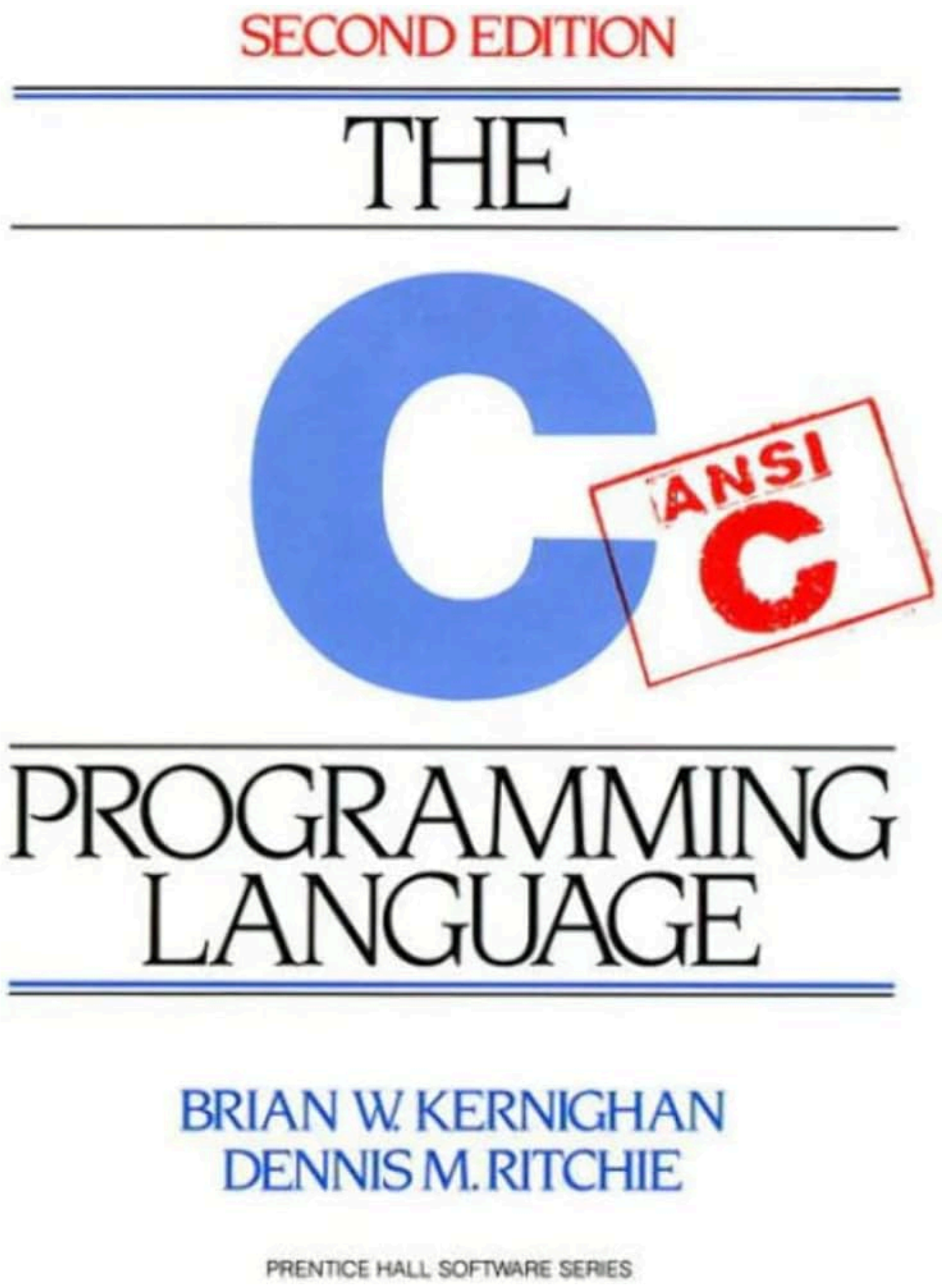
↑ Works!

Questions

- What is (and is not) valid C?
- Who defines the C language?
- What does `-std=c2x` mean?

What is valid C?

- Pragmatically, C code is valid if your compiler builds an executable that does what you want
- However, there are many compilers
- It would be convenient if they all agreed on a definition for the C language



SECOND EDITION

THE

C

ANSI C

PROGRAMMING LANGUAGE

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES

In the beginning, there was K&R C (1978)

- 1978: Kernighan and Ritchie publish their book
- 1983: The American National Standards Institute (ANSI) forms a committee to standardize C
- 1989: The commitee publishes the standard, “ANSI C” / “C89”
- 1990: The International Organization for Standardization (ISO) adopts the standard
- 1999: ISO updates the standard (ANSI adopts it): “C99”
- 2011: ISO update: “C11”
- 2017: ISO update: “C17”
- ISO working on update “C23”, provisionally “C2x” (publication expected October 2024)

Hence `-std=c2x` (`-std=c23` may work with recent compilers)

Who defines the C language nowadays?

- A “working group” within ISO: “WG14”
  - Compiler writers
  - Hardware vendor representatives
  - OS maintainers
  - Academics

> C23 draft

(742 pages)

Behaviors

Locale-specific behavior

Behavior that depends on local conventions (nationality, culture, and language) that each implementation **documents**.

Example:

Whether `islower()` returns true for characters other than the 26 lowercase Latin letters.

```
int a = islower('è');
```

Unspecified behavior

- “Behavior upon which this document provides two or more possibilities and imposes no further requirements on which is chosen in any instance”
- “Behavior that results from the use of an unspecified value”

Examples

- The order in which the arguments to a function are evaluated.
- Value of padding bytes:

```
struct s {
    char a;    // 1 byte
               // 3 padding bytes
    int b;     // 4 bytes
};
```

Implementation-defined behavior

Unspecified behavior where each implementation (compiler / platform / OS) **documents** how the choice is made

Example

The propagation of the high-order bit when a signed integer is shifted right.

```
int a = -8;
int b = a >> 1;
```

On x86_64 and AArch64: sign-extend

Undefined behavior

“Behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this document imposes **no requirements**”.

Possibly:

- the situation is completely ignored with unpredictable results,
- implementation-defined behavior
- compilation yields error message
- execution yields error message
- compilation crashes
- execution crashes
- **anything else**

Undefined behavior

Example

```
int *a = NULL;
int b = *a;
```

Undefined behavior

Easy UB: division by zero

> *“The result of the / operator is the quotient from the division of the first operand by the second; the result of the % operator is the remainder.*
> *In both operations, if the value of the second operand is zero, the behavior is undefined.”* (p83)

```
int main(int argc)
{
    return 5 / (argc - 1);
}
```

```
./main
Floating point exception (core dumped)
```

Easy UB? (division by zero)

```c
#include <stdio.h>

int main()
{
    printf("%d\n", 5 / 0);
    return 0;
}
```

```
clang -O3 -std=c2x -o main main.c
main.c:3:11: warning: division by zero is undefined [-Wdivision-by-zero]
        return 5 / 0;
                 ^ ~
```

```
./main
-882586408
```

```
./main
1687000168
```

```
./main
-1071941800
```

```
./main
-60110776
```

```
0000000000401130 <main>:
  401130:       50                      push   rax
  401131:       bf 10 20 40 00          mov    edi,0x402010
  401136:       31 c0                   xor    eax,eax
  401138:       e8 f3 fe ff ff          call   401030 <printf@plt>
  40113d:       31 c0                   xor    eax,eax
  40113f:       59                      pop    rcx
  401140:       c3                      ret
```

Easy UB: division overflow

> “*When integers are divided, the result of the / operator is the algebraic quotient with any fractional part discarded* (*”truncation toward zero”*).
>
> *If the quotient `a/b` is representable, the expression `(a/b)*b + a%b` shall equal `a`; otherwise, the behavior of both `a/b` and `a%b` is undefined.*” (p83)

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

void print_if_negative(int a)
{
    if (a >= 0)
        return;

    printf("a = %d\n", a);
    printf("a / -1 = %d\n", a / -1);
}

int main()
{
    print_if_negative(-5);

    return 0;
}
```

```
a = -5
a / -1 = 5
```

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

void print_if_negative(int a)
{
    if (a >= 0)
        return;

    printf("a = %d\n", a);
    printf("a / -1 = %d\n", a / -1);
}

int main()
{
    print_if_negative(INT_MIN);

    return 0;
}
```

Reminder: `int` can represent $\{-2147483648, \ldots, 2147483647\}$.

```
a = -2147483648
a / -1 = -2147483648
```

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

void print_if_negative(int a)
{
    if (a >= 0)
        return;

    printf("a = %d\n", a);
    printf("a / -1 = %d\n", a / -1);

    if (a / -1 > 0)
        printf("a / -1 = %d is positive\n", a / -1);
}

int main()
{
    print_if_negative(INT_MIN);

    return 0;
}
```

```
a = -2147483648
a / -1 = -2147483648
a / -1 = -2147483648 is positive
```

Integer overflow

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

int main()
{
    uint8_t a = 0;

    for (int i = 0; i < 1000; i++) {
        printf("%012b\n", a);
        a = a + 1;
    }

    return 0;
}
```

Note: $1000 > 2^8 = 256$.

```
000000000000
000000000001
000000000010
000000000011
000000000100
...
000011111101
000011111110
000011111111
000000000000
000000000001
```

Unsigned integer overflow

- Unsigned overflow is **not** undefined behavior
- Unsigned overflow has wrap-around behavior:
    - if i, j are n-bit unsigned integers
        - then `i + j` yields $(i + j) \bmod 2^n$
    - for any operation on unsigned n-bit integers,
        - the result is the bottom n bits of the true arithmetic value
- x86_64 and AArch64 instruction work in this same way

Signed integer overflow

- x86_64 and AArch64 instruction have wrap-around behavior
- But in C, signed overflow **is** undefined behavior!!!

Signed integer overflow

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

void print_if_positive(int a)
{
    if (a <= 0)
        return;

    printf("a = %d\n", a);
    printf("a + 1 = %d\n", a + 1);

    if (a + 1 > 0)
        printf("a + 1 = %d is positive\n", a + 1);
}

int main()
{
    print_if_positive(INT_MAX);

    return 0;
}
```

```
a = 2147483647
a + 1 = -2147483648
a + 1 = -2147483648 is positive
```

Pointers in C

What pointers are

A pointers contains a (virtual) address in memory.

Pointer declaration

- Pointers are declared using the * symbol.

- They are usually accompanied by the **type** pointed to.

Pointer declaration examples

```
char *pa;
```

pa contains the memory address of a character.

```
int *pb;
```

pb contains the memory address of an `int`.

```
struct vec {
    int l;
    float x, y, z;
} *pc;
```

pc contains the memory address of a `struct vec`.

```c
char *pa;
int *pb;
struct vec {
    int l;
    float x, y, z;
} *pc;
```

…	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91
…		char			int								int l				float x				float y				float z			
…		↑			↑								↑															
…		pa			pb								pc															

Pointer notation

When a variable is declared as a pointer, its value is the value of the pointer:

```
char *pa;
// ...
printf("address = %p\n", pa);  // Prints memory address contained by pa in hex
```

When we want to access the memory that the pointer points to, we use *:

```
char *pa;
// ...
printf("address = %p, memory content: '%c'\n", pa, *pa);
```

^ this is called *dereferencing* the pointer.

To take the address of an object in memory, use &:

```
char *pa;
char a = 'K';
pa = &a;
printf("address = %p, memory content: '%c'\n", pa, *pa); // Prints memory address of a, then 'K'
```

Pointer arithmetic

The compiler uses the pointed type when performing arithmetic on the pointer:

```
char *pa;
int *pb;
struct vec {
    int l;
    float x, y, z;
} *pc;
```

…	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92
…		char			int								int l				float x				float y				float z				
…		↑	↑		↑				↑				↑																↑
…		pa	pa + 1		pb				pb + 1				pc																pc + 1

Array notation

The array indexing notation in C

```c
int *pb;
// ...
int i = pb[10];
```

is equivalent to pointer dereferencing:

```c
int *pb;
// ...
int i = *(pb + 10);
```

In particular, the following to lines are equivalent:

```c
int i = *pb;
int i = pb[0];
```

Warning: In a **declaration**, the pointer and array notations are not synonymous:

```c
int *pb;        // <--- this declares a pointer; its initial value is uninitialized
int pb2[40];    // <--- this declarse 40 integers; pb2 is the address of the first one
```

Type casting

In C, **explicit** type conversion ("casting") is indicated by preceding an object by the new type, between parenthesis:

```c
char letter = 'A';
int UTF8_code_for_letter = (int)letter;
```

Note in C, many type conversions are performed **implicitly**:

```c
int UTF8_code_for_letter = 'A';
```

```c
int x = 'x';
int lower_case_x = x - 'A' + 'a';
```

Pointer casting

```
int b;
int *pb = &b;
char *pa = (char *)&b;
```

…	64	65	66	67	68	69	70	71	72	73	74	75
…					int b;							
…					↑							
…					pb							
…					pa							

```
printf("%d\n", *pa);
// ^ prints the first 8 bits of b:
//   - least significant 8 bits of b if little-endian
//   -  most significant 8 bits of b if big-endian
```

What if a variable is not in memory

```c
for (int i = 0; i < 10; i++) {
    printf("%d\n", i);
}
```

```c
for (int i = 0; i < 10; i++) {
    printf("%p. %d\n", &i, i);
}
```

In the second case, the compiler must act “as if” `i` was in memory.

NULL pointer

The C language specifies that the NULL pointer (address zero) is never valid.

Easy UB: invalid pointers

*“If an invalid value has been assigned to the pointer,
the behavior of the unary * operator is undefined.”* (p81)

```
int int_at(int *pointer)
{
    int r = *pointer;

    return r;
}

int main()
{
    printf("%d", int_at((int *)1));
    return 0;
}
```

```
./main
Segmentation fault (core dumped)
```

Easy UB?!?? (invalid pointers)

```c
int int_at(int *pointer)
{
    int r = *pointer;

    if (pointer == NULL)
        return 0;

    return r;
}
```

```
0000000000401110 <int_at>:
  401110:       8b 07                   mov    eax,DWORD PTR [rdi]
  401112:       c3                      ret
```