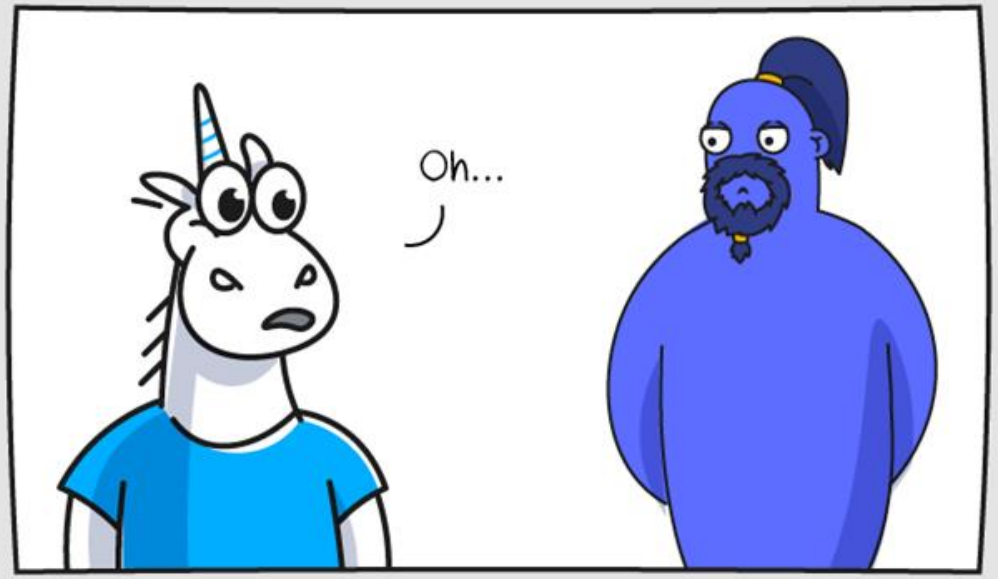
A Hitchhiker's Guide to

UNDEFINED BEHAVIOR

00001101 00001010 00001101 00001010 00001101 00001010


I want to know
everything about
undefined behavior


Done


Oh...


Oh, Lord!

Chapter I.
The Standard

2025. 06. 24.

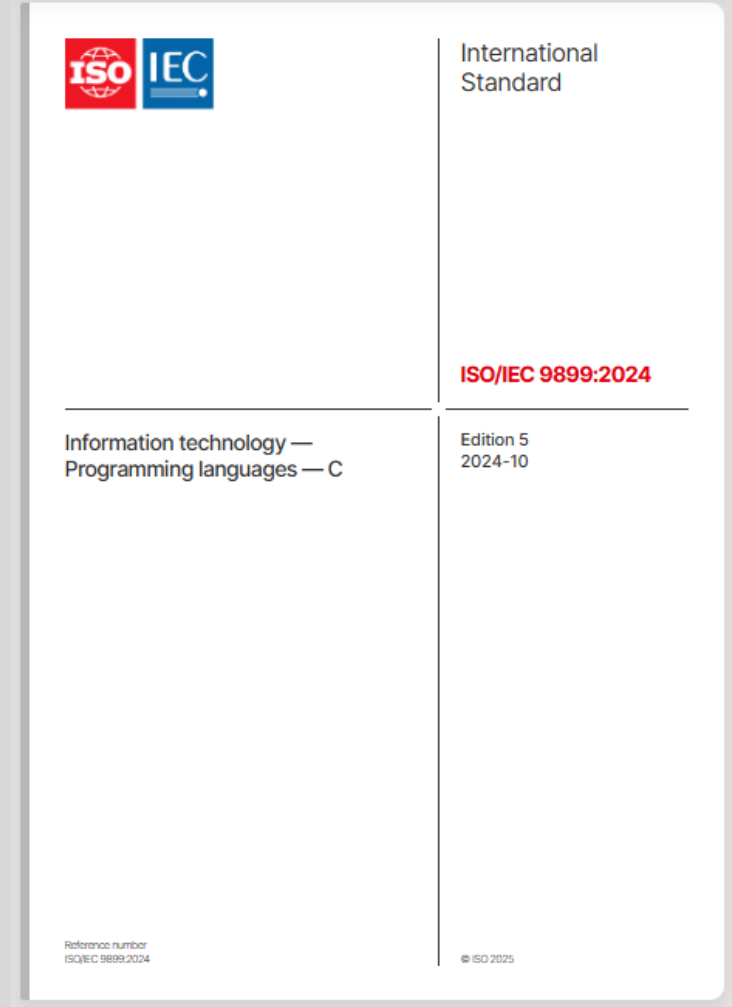
The C standard

- C language invented by: Dennis Ritchie (1970s)
- Became popular → first attempt at standardization in 1989 (ANSI)
- Currently ISO C 23
  - International Organization for Standardization

6.4.9 Comments

1 Except within a character constant, a string literal, or a comment, the characters `/*` introduce a comment. The contents of such a comment are examined only to identify multibyte characters and to find the characters `*/` that terminate it.[83)]

2 Except within a character constant, a string literal, or a comment, the characters `//` introduce a comment that includes all multibyte characters up to, but not including, the next new-line character. The contents of such a comment are examined only to identify multibyte characters and to find the terminating new-line character.



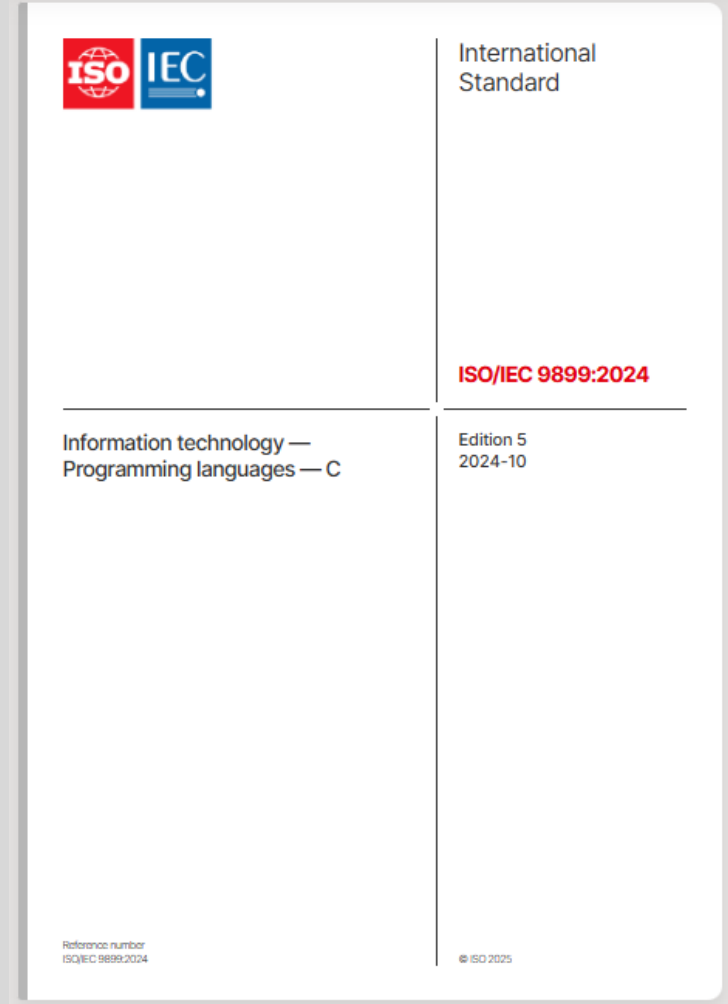
The C standard

- C language invented by: Dennis Ritchie (1970s)
- Became popular → first attempt at standardization in 1989 (ANSI)
- Currently ISO C 23
  - International Organization for Standardization

6.4.9 Comments

1 Except within a character constant, a string literal, or a comment, the characters `/*` introduce a comment. The contents of such a comment are examined only to identify multibyte characters and to find the characters `*/` that terminate it.[83)]

2 Except within a character constant, a string literal, or a comment, the characters `//` introduce a comment that includes all multibyte characters up to, but not including, the next new-line character. The contents of such a comment are examined only to identify multibyte characters and to find the terminating new-line character.

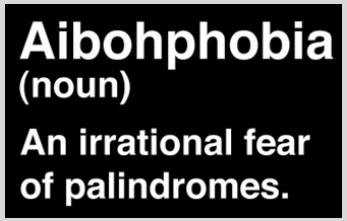


Question: who needs to know this standard the best?

3.4 – behavior: external appearance or action

- **3.4.1 implementation-defined behavior**
  - unspecified behavior where each implementation **documents** how the choice is made
  - **EXAMPLE** An example of implementation-defined behavior is the propagation of the high-order bit when a signed integer is shifted right.
    - arithmetic vs. logical vs. rotation
- **3.4.2 locale-specific behavior**
  - behavior that depends on local conventions of nationality, culture, and language that each implementation **documents**
  - **EXAMPLE** An example of locale-specific behavior is whether the *islower* function returns *true* for characters other than the 26 lowercase Latin letters.
- **3.4.4 unspecified behavior**
  - use of an unspecified value, or other behavior where this International Standard **provides two or more possibilities** and imposes no further requirements on which is chosen in any Instance
  - **EXAMPLE** An example of unspecified behavior is the order in which the arguments to a function are evaluated.
    - write(1, string(), len());

3.4.3 undefined behavior



- behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard **imposes no requirements.**
- **NOTE** Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).
- **EXAMPLE** An example of undefined behavior is the behavior on **integer overflow**.

Surprisingly, *what* counts as UB is quite well defined

Some examples (C11 annex J.2)

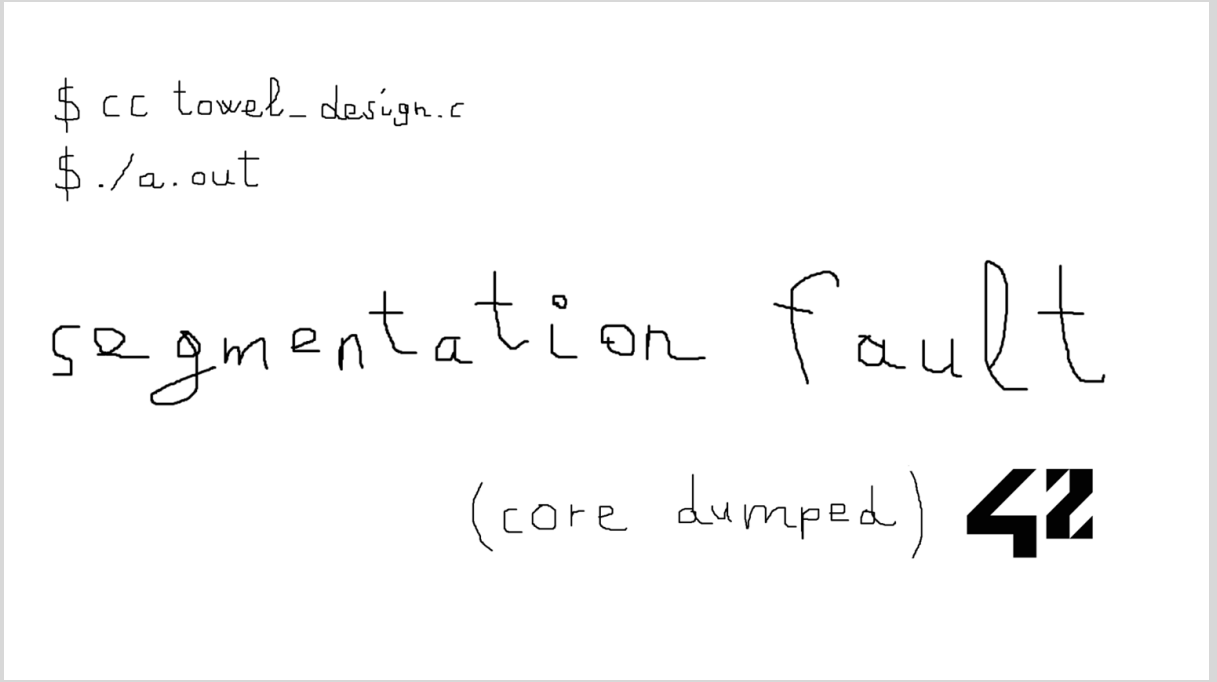
- The operand of the unary *** operator** has **an invalid value**
  - Among the invalid values for dereferencing a pointer by the unary * operator are a **null pointer**, an address inappropriately aligned for the type of object pointed to, and the address of an object after the end of its lifetime. (6.5.3.2.4/102)
- The value of a pointer that refers to space **deallocated** by a call to the *free* or *realloc* function is **used**
- The value of the second operand of the **/ or % operator** is **zero**
- The program attempts to **modify a string literal**
- Conversion to or from an integer type produces a value **outside the range** that can be represented
- A signal handler called in response to SIGFPE, SIGILL, SIGSEGV, or any other implementation-defined value corresponding to a computational exception **returns**
- A nonempty source file **does not end in a new-line character** which is not immediately preceded by a backslash character or ends in a partial preprocessing token or comment
- The execution of a program contains a **data race**

What is the definition of UB?

Chapter II.
Is UB the same as `Segmentation Fault`?

The Real Question

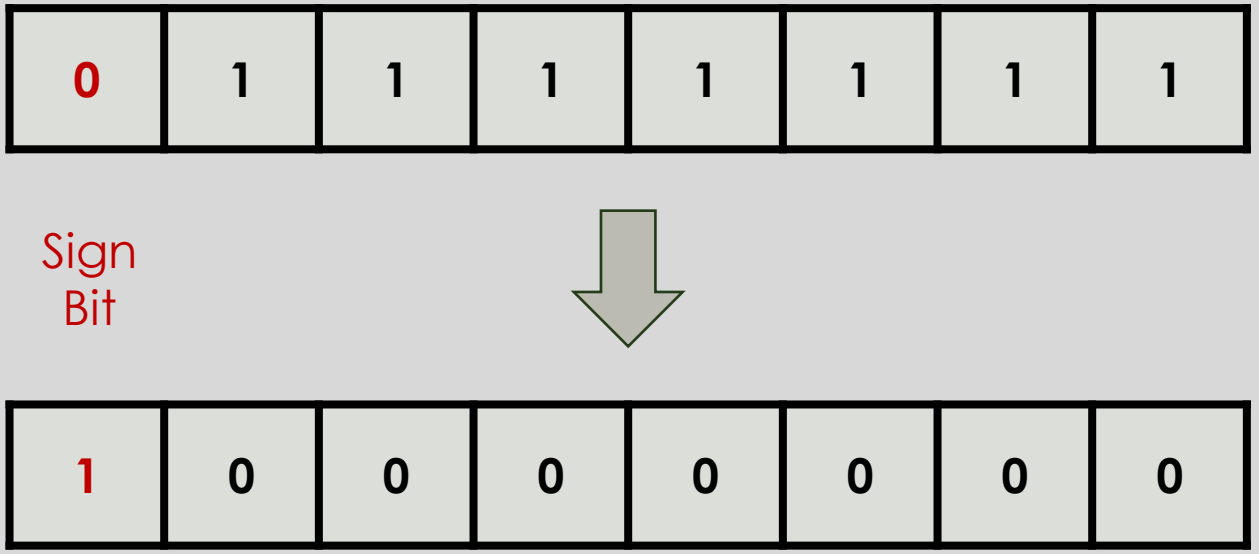
- **Can architecture *define* something that is undefined by the Standard?**
  - The theory: NULL ptr dereference is undefined
  - The practice:



Example: what happens on signed integer overflow?

- **C Piscine – Rush 00**

```c
void    rush(int x, int y)
{
    int max_y;
    int current_y;

    max_y = y;
    current_y = 1;
    while (current_y <= max_y)
    {
        if (current_y == 1)
        {
            first_line(x);
            ft_putchar('\n');
        }
        else if (current_y == max_y)
        {
```



(this is a signed char but the same thing happens)

Do rules of architecture override the C Standard?

Chapter III.
The Abstract Machine

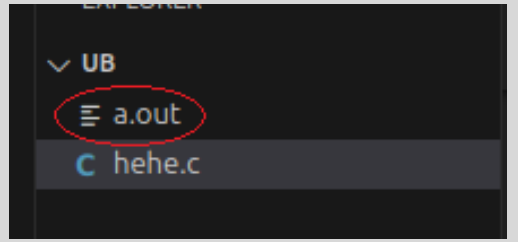
2025. 06. 24.

What happens when you „cc” a file?

```c
#include <unistd.h>

int main(void)
{
    int *ptr = NULL;
    *ptr = 0;
    while (*ptr < 3)
    {
        write(1, "hehe\n", 5);
        (*ptr)++;
    }
    return (42);
}
```

hehe.c

$ cc hehe.c

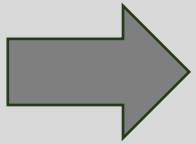


???

$ hexdump a.out

```c
#include <unistd.h>

int main(void)
{
    int *ptr = NULL;
    *ptr = 0;
    while (*ptr < 3)
    {
        write(1, "hehe\n", 5);
        (*ptr)++;
    }
    return (42);
}
```

hehe.c



```
00010f0 e0ff 0f66 441f 0000 0fc3 801f 0000 0000
0001100 0ff3 fa1e 3d80 2f05 0000 7500 552b 8348
0001110 e23d 002e 0000 8948 74e5 480c 3d8b 2ee6
0001120 0000 19e8 ffff e8ff ff64 ffff 05c6 2edd
0001130 0000 5d01 0fc3 001f 0fc3 801f 0000 0000
0001140 0ff3 fa1e 77e9 ffff f3ff 1e0f 55fa 8948
0001150 48e5 ec83 4810 45c7 00f8 0000 4800 458b
0001160 c7f8 0000 0000 eb00 ba28 0005 0000 8d48
0001170 8f05 000e 4800 c689 01bf 0000 e800 fece
0001180 ffff 8b48 f845 008b 508d 4801 458b 89f8
0001190 4810 458b 8bf8 8300 02f8 cd7e 2ab8 0000
00011a0 c900 00c3 0ff3 fa1e 8348 08ec 8348 08c4
00011b0 00c3 0000 0000 0000 0000 0000 0000 0000
00011c0 0000 0000 0000 0000 0000 0000 0000 0000
*
0002000 0001 0002 6568 6568 000a 0000 1b01 3b03
0002010 0030 0000 0005 0000 f014 ffff 0064 0000
0002020 f034 ffff 008c 0000 f044 ffff 00a4 0000
0002030 f054 ffff 004c 0000 f13d ffff 00bc 0000
```

a.out

hehe.c → intermediate representations (+ optimizations) → assembly → binary

```
int main ()
{
  int D.3649;

  {
    int * ptr;

    ptr = 0B;
    *ptr = 0;
    goto <D.3646>;
    <D.3647>:
    write (1, "hehe\n", 5);
    _1 = *ptr;
    _2 = _1 + 1;
    *ptr = _2;
    <D.3646>:
    _3 = *ptr;
    if (_3 <= 2) goto <D.3647>; else goto <D.3645>;
    <D.3645>:
    D.3649 = 42;
    return D.3649;
  }
  D.3649 = 0;
  return D.3649;
}
```

GIMPLE

```
0000000000001149 <main>:
    1149:   f3 0f 1e fa             endbr64
    114d:   55                      push   %rbp
    114e:   48 89 e5                mov    %rsp,%rbp
    1151:   48 83 ec 10             sub    $0x10,%rsp
    1155:   48 c7 45 f8 00 00 00    movq   $0x0,-0x8(%rbp)
    115c:   00
    115d:   48 8b 45 f8             mov    -0x8(%rbp),%rax
    1161:   c7 00 00 00 00 00       movl   $0x0,(%rax)
    1167:   eb 28                   jmp    1191 <main+0x48>
    1169:   ba 05 00 00 00          mov    $0x5,%edx
    116e:   48 8d 05 8f 0e 00 00    lea    0xe8f(%rip),%rax
    1175:   48 89 c6                mov    %rax,%rsi
    1178:   bf 01 00 00 00          mov    $0x1,%edi
    117d:   e8 ce fe ff ff          call   1050 <write@plt>
    1182:   48 8b 45 f8             mov    -0x8(%rbp),%rax
    1186:   8b 00                   mov    (%rax),%eax
    1188:   8d 50 01                lea    0x1(%rax),%edx
    118b:   48 8b 45 f8             mov    -0x8(%rbp),%rax
    118f:   89 10                   mov    %edx,(%rax)
    1191:   48 8b 45 f8             mov    -0x8(%rbp),%rax
    1195:   8b 00                   mov    (%rax),%eax
    1197:   83 f8 02                cmp    $0x2,%eax
    119a:   7e cd                   jle    1169 <main+0x20>
    119c:   b8 2a 00 00 00          mov    $0x2a,%eax
    11a1:   c9                      leave
    11a2:   c3                      ret
```

ASSEMBLY

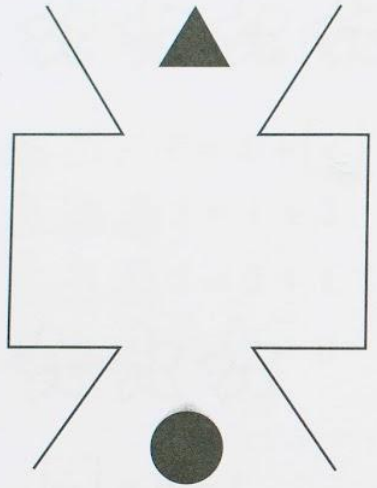
```
00010f0 e0ff 0f66 441f 0000 0fc3 801f 000
0001100 0ff3 fa1e 3d80 2f05 0000 7500 552
0001110 e23d 002e 0000 8948 74e5 480c 3d8
0001120 0000 19e8 ffff e8ff ff64 ffff 05c
0001130 0000 5d01 0fc3 001f 0fc3 801f 000
0001140 0ff3 fa1e 77e9 ffff f3ff 1e0f 55f
0001150 48e5 ec83 4810 45c7 00f8 0000 480
0001160 c7f8 0000 0000 eb00 ba28 0005 000
0001170 8f05 000e 4800 c689 01bf 0000 e80
0001180 ffff 8b48 f845 008b 508d 4801 458
0001190 4810 458b 8bf8 8300 02f8 cd7e 2ab
00011a0 c900 00c3 0ff3 fa1e 8348 08ec 834
00011b0 00c3
00011c0 0000
*
0002000 0001
0002010 0030
0002020 f034 ffff 008c 0000 f044 ffff 00a
0002030 f054 ffff 004c 0000 f13d ffff 00b
```

83 → ALU 8imm
F8 → 11111000
REG – CMP – EAX

BINARY

The Abstract Machine

3. Mit csinál a gép? Pótold a hiányzó számokat!



▲	2	5			3	4		
●	0	3	2	0			3	1

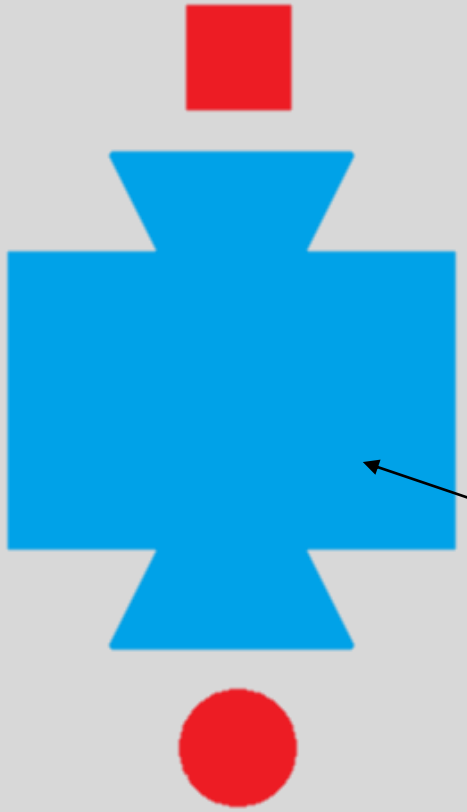
The Abstract Machine

The Standard introduces this concept because compilers might need to translate the exact same C code to **very different** architectures

The implementation itself does not matter as long as the **observable behavior** of the program remains the same

The Abstract Machine

The Standard introduces this concept because compilers might need to translate the exact same C code to **very different** architectures



The implementation itself does not matter as long as the **observable behavior** of the program remains the same

C code itself is an abstraction. You are basically describing **this** with what the standard calls **abstract semantics**

a.out, on the other hand, contains **actual semantics** (the mechanism inside the machine)

The Abstract Machine

5.1.2.3.6. At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced.

5.1.2.3.9. „An implementation might define a one-to-one correspondence between abstract and actual semantics: at every sequence point, the values of the actual objects would agree with those specified by the abstract semantics. The keyword *volatile* would then be redundant.
5.1.2.3.10. Alternatively, an implementation might perform **various optimizations** within each translation unit, such that the actual semantics would agree with the abstract semantics only when making function calls across translation unit boundaries."

The Abstract Machine

5.1.2.3.6. At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced.

6.4.9 Comments

1 Except within a character constant, a string literal, or a comment, the characters `/*` introduce a comment. The contents of such a comment are examined only to identify multibyte characters and to find the characters `*/` that terminate it.[83)]

2 Except within a character constant, a string literal, or a comment, the characters `//` introduce a comment that includes all multibyte characters up to, but not including, the next new-line character. The contents of such a comment are examined only to identify multibyte characters and to find the terminating new-line character.

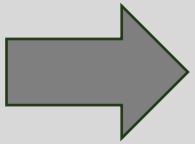
REMEMBER!!


a-hehe.c.030t.local-fnsummary1
a-hehe.c.031t.einline
a-hehe.c.032t.early_optimizations
a-hehe.c.033t.early_objsz
a-hehe.c.034t.ccp1
a-hehe.c.035t.forwprop1
a-hehe.c.036t.ethread
a-hehe.c.037t.esra
a-hehe.c.038t.ealias
a-hehe.c.039t.fre1
a-hehe.c.040t.evrp
a-hehe.c.041t.mergephi1
a-hehe.c.042t.dse1
a-hehe.c.043t.cddce1
a-hehe.c.044t.phiopt1
a-hehe.c.045t.tail
a-hehe.c.046t.iftoswitch
a-hehe.c.047t.switchconv
a-hehe.c.049t.profile_estimate
a-hehe.c.050t.local-pure-const1
a-hehe.c.051t.modref1
a-hehe.c.052t.fnsplit
a-hehe.c.053t.release_ssa
a-hehe.c.054t.local-fnsummary2
a-hehe.c.094t.fixup_cfg3
a-hehe.c.101t.adjust_alignment
a-hehe.c.102t.ccp2
a-hehe.c.103t.objsz1
a-hehe.c.104t.post_ipa_warn1
a-hehe.c.105t.waccess2
a-hehe.c.106t.cunrolli
a-hehe.c.107t.backop
a-hehe.c.108t.phiprop
a-hehe.c.109t.forwprop2
a-hehe.c.110t.alias
a-hehe.c.111t.retslot
a-hehe.c.112t.fre3
a-hehe.c.113t.mergephi2
a-hehe.c.114t.threadfull1
a-hehe.c.115t.vrp1
a-hehe.c.116t.dse2
a-hehe.c.117t.dce2
a-hehe.c.118t.stdarg
a-hehe.c.119t.cdce
a-hehe.c.120t.cselim
a-hehe.c.121t.copyprop1
a-hehe.c.122t.ifcombine
a-hehe.c.123t.merphi3
a-hehe.c.124t.phiopt2
a-hehe.c.125t.tailr2
a-hehe.c.126t.ch2
a-hehe.c.127t.cplxlower1
a-hehe.c.128t.sra
a-hehe.c.129t.thread1
a-hehe.c.130t.dom2
a-hehe.c.131t.copyprop2
a-hehe.c.132t.isolate-paths
a-hehe.c.133t.reassoc1
a-hehe.c.134t.dce3
a-hehe.c.135t.forwprop3
a-hehe.c.136t.phiopt3
a-hehe.c.137t.ccp3
a-hehe.c.138t.powcabs
a-hehe.c.139t.bswa
a-hehe.c.140t.laddress
a-hehe.c.141t.lim2
a-hehe.c.142t.walloca2
a-hehe.c.143t.pre
a-hehe.c.144t.sink1
a-hehe.c.148t.dse3
a-hehe.c.149t.dce4
a-hehe.c.150t.fix_loops
a-hehe.c.151t.loop
a-hehe.c.152t.loopinit
a-hehe.c.153t.unswitch
a-hehe.c.154t.sccp
a-hehe.c.155t.lsplit
a-hehe.c.156t.lversion
a-hehe.c.157t.unrolljam
a-hehe.c.158t.cddc
a-hehe.c.159t.ivcanon
a-hehe.c.160t.ldist
a-hehe.c.161t.linterchange
a-hehe.c.162t.copyprop3
a-hehe.c.170t.ch_vect
a-hehe.c.171t.ifcvt
a-hehe.c.172t.vect
a-hehe.c.173t.dce6
a-hehe.c.174t.pcom
a-hehe.c.175t.cunroll
a-hehe.c.178t.slp1
a-hehe.c.180t.ivopts
a-hehe.c.181t.lim4
a-hehe.c.182t.loopdone
a-hehe.c.186t.veclower21
a-hehe.c.187t.switchlower1
a-hehe.c.188t.sincos
a-hehe.c.190t.reassoc2
a-hehe.c.191t.slsr
a-hehe.c.192t.split-paths
a-hehe.c.194t.fre5
a-hehe.c.195t.thread2
a-hehe.c.196t.dom3
a-hehe.c.197t.strlen1
a-hehe.c.198t.threadfull2
a-hehe.c.199t.vrp2
a-hehe.c.200t.ccp4
a-hehe.c.201t.wrestrict
a-hehe.c.202t.dse5
a-hehe.c.203t.dce7
a-hehe.c.204t.forwprop4
a-hehe.c.205t.sink

In short

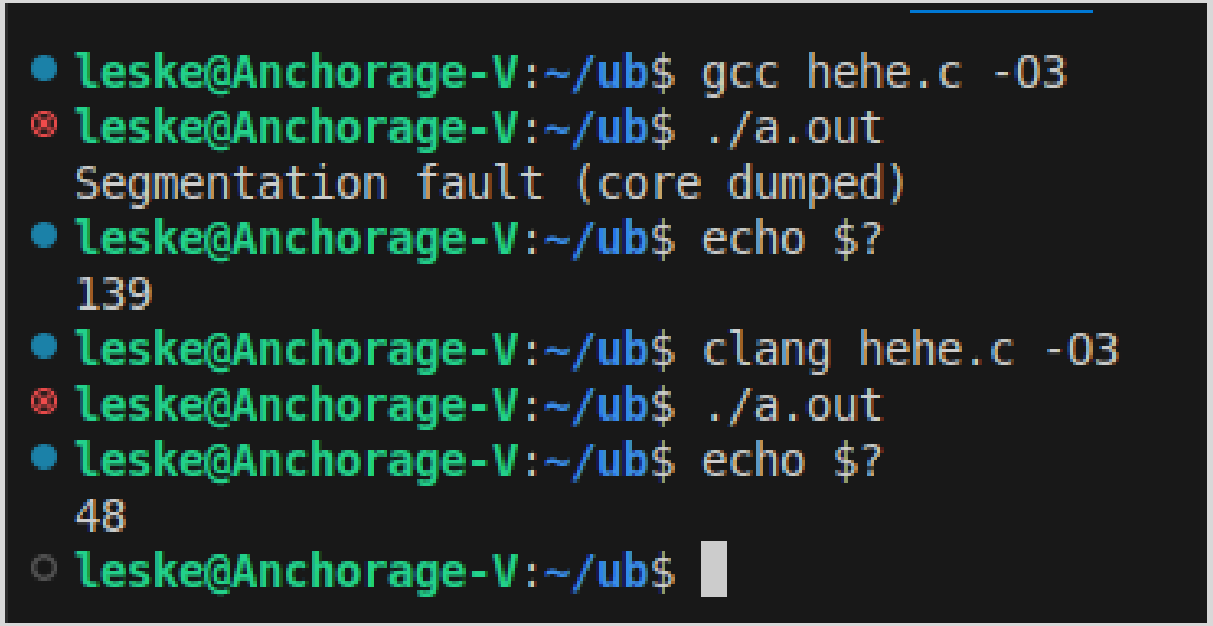
The code *you* write will NOT be the same code that gets executed!

```c
#include <unistd.h>

int main(void)
{
    int *ptr = NULL;
    *ptr = 0;
    while (*ptr < 3)
    {
        write(1, "hehe\n", 5);
        (*ptr)++;
    }
    return (42);
}
```



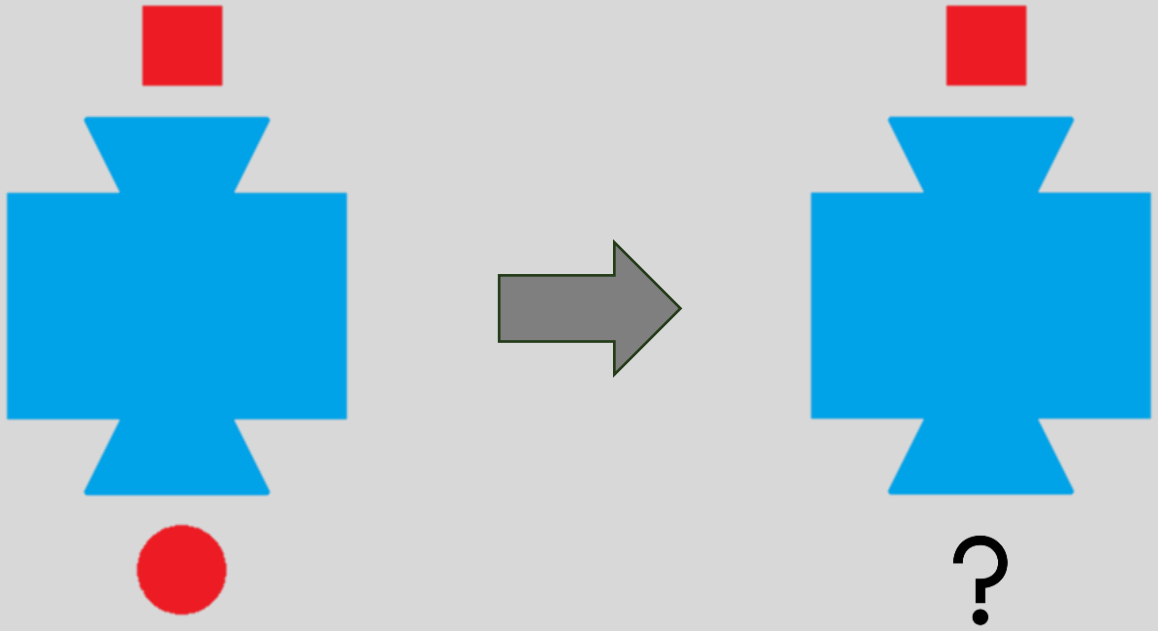
```
;; Function main (main, funcdef_no=12, d

int main ()
{
  <bb 2> [local count: 118111600]:
  MEM[(int *)0B] ={v} 0;
  __builtin_trap ();

}
```

But how is this relevant to UB?



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

„executing an erroneous operation causes the entire program to be **meaningless**”



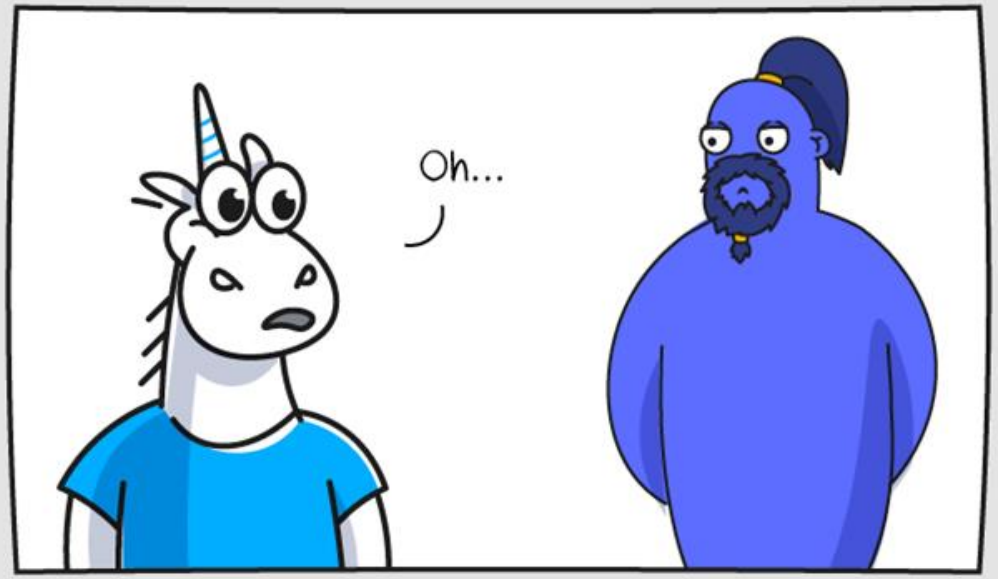
In **GCC 1.17**, when the compiler encountered specific forms of undefined behavior (unknown/not implemented #pragmas), here's the code it executed:

```
execl("/usr/games/hack", "#pragma", 0); // try to run the game NetHack

execl("/usr/games/rogue", "#pragma", 0); // try to run the game Rogue

// try to run the Tower's of Hanoi simulation in Emacs.
execl("/usr/new/emacs", "-f","hanoi","9","-kill",0);

execl("/usr/local/emacs","-f","hanoi","9","-kill",0); // same as above

fatal("You are in a maze of twisty compiler features, all different");[/c]
```

Compiler is free to do anything!


I want to know
everything about
undefined behavior


Done


Oh...


Oh, Lord!

But wait… it gets worse

```c
#include <unistd.h>

int main(void)
{
    int *ptr = NULL;
    *ptr = 0;
    while (*ptr < 3)
    {
        write(1, "hehe\n", 5);
        (*ptr)++;
    }
    return (42);
}
```

UB can happen **explicitly**

```c
void print_array(int *arr, int size)
{
    int i = 0;
    while (i < size)
    {
        printf("%d ", *(arr + i));
        i++;
    }
    printf("\n");
}
```

Or it can also happen **implicitly**

But wait… it gets worse

```c
#include <unistd.h>

int main(void)
{
    int *ptr = NULL;
    *ptr = 0;
    while (*ptr < 3)
    {
        write(1, "hehe\n", 5);
        (*ptr)++;
    }
    return (42);
}
```

UB can happen **explicitly**

Compiler can (and will) work with the assumption everything that is defined as UB will never happen in your code

That means this
→
will never receive NULL

```c
void print_array(int *arr, int size)
{
    int i = 0;
    while (i < size)
    {
        printf("%d ", *(arr + i));
        i++;
    }
    printf("\n");
}
```

Or it can also happen **implicitly**

```c
char    *handle_envp(char *str, t_data *node)
{
    char    *result;
    int     i;

    i = 0;
    result = malloc(sizeof(char) * (ft_strlen(str) + 1));
    if (!result)
        ft_exit(node, -1, "malloc in handle_envp failed");
    result[0] = '\0';
    while (str[i])
    {
```

that means this

```c
char    *handle_envp(char *str, t_data *node)
{
    char    *result;
    int     i;

    i = 0;
    result = malloc(sizeof(char) * (ft_strlen(str) + 1));
    result[0] = '\0';
    while (str[i])
    {
```

can be turned into this

2025. 06. 24.

So: *is it* okay to rely on UB if the architecture seems to guarantee behavior?

- While nothing forbids you from doing it, it will make your code horribly **unportable**.
- If you want consistent behavior, you will need to document:
  - Compiler
    - name (gcc)
    - version (13.3.0)
    - options (compiler flags) used (-fforward-propagate -finline-functions -floop-unroll-and-jam)
  - Architecture (x86-64)
- You can't really change anything in your code (even if it seems unrelated)
- It will be super hard to use it & people will not want to try
- BUT technically it is doable
- This is NOT the point of UB!!

Chapter V.
How to detect UB

(yes IV is missing)

2025. 06. 24.

Non comprehensive list of UB Fighting Methods

- Spread awareness on campus!!
- Compile with optimization flags
  - If your code has no UB, it should give the exact same output & behavior with O3 as with O0
  - Don't submit with O3 but use it to test your code before submission
- Static analyzers (for compile time UB)
  - Can detect some more obvious cases
- Use UBSan for runtime UB
  - But then again this will only cry if shit *happens* which is not guaranteed at all
  - By that time code might already be removed

Chapter IV.

2025. 06. 24.

Why can we not just define all behavior?

- Most undefined behavior is *explicitly named* in standard
- They even have a separate list of them at the end
- Could not they just all be defined? So both compiler writers and C coders have an easier time

Why do we even have UB?

Time to introduce an imaginary language called „D”

D is an actual language

Respecting the standard is NOT the responsibility of the coder – it's the reponsibility of the compiler

That's why we get errors and warnings

Undefined for C: The operand of the unary *** operator** has **an invalid value**. Among the invalid values for dereferencing a pointer by the unary * operator are a null pointer, an address inappropriately aligned for the type of object pointed to, and the address of an object after the end of its lifetime.

Let's define for D!
If the operand of the unary ***** **operator** has **an invalid value…**

?

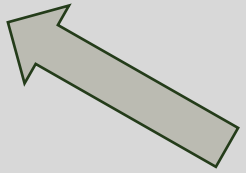
```
int *ptr = NULL;
*ptr = 0;
while (*ptr < 3)
```

Could compiler catch it with an error?

Invalid value 1: a null pointer

```c
void print_array(int *arr, int size)
{
    int i = 0;
    while (i < size)
    {
        printf("%d ", *(arr + i));
        i++;
    }
    printf("\n");
}
```

?



Cannot be caught at compile time!

Invalid value 1: a null pointer

```c
void print_array(int *arr, int size)
{
    int i = 0;
    while (i < size)
    {
        printf("%d ", *(arr + i));
        i++;
    }
    printf("\n");
}
```

```
  void print_array(int *arr, int size)
  {
+     if (arr == NULL)
+     {
+         dprintf(2, "Error: NULL pointer passed to print_array\n");
+         abort();
+     }
      int i = 0;
      while (i < size)
      {
          printf("%d ", *(arr + i));
          i++;
      }
      printf("\n");
  }
```

Compiler needs to add it! (not user)

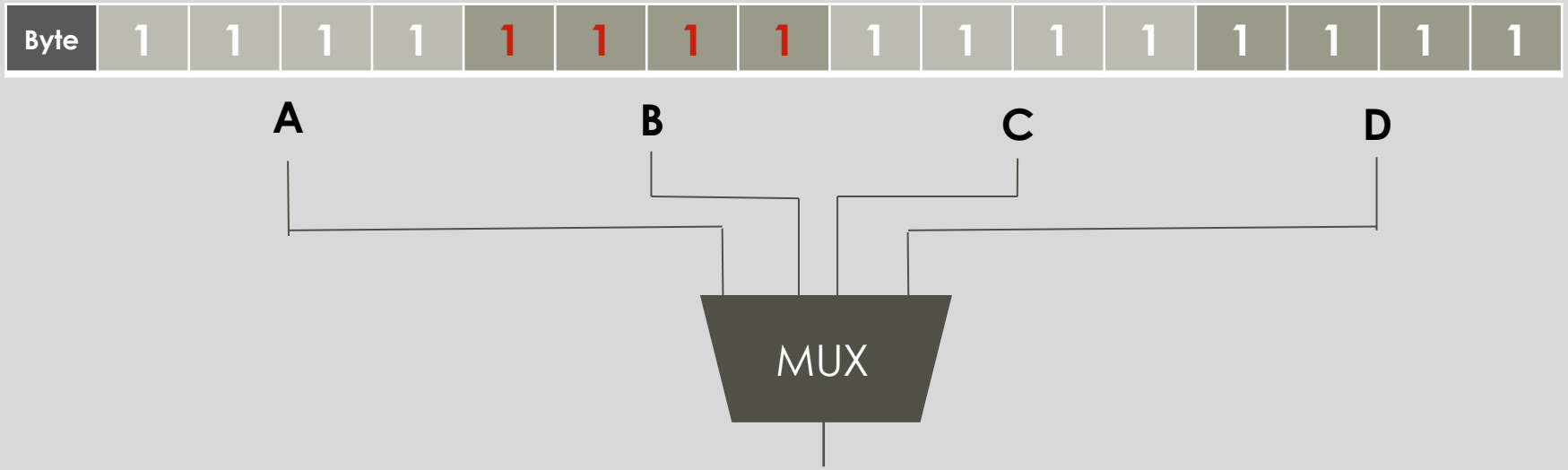
Invalid value 2: misaligned address

What does a misaligned address mean?

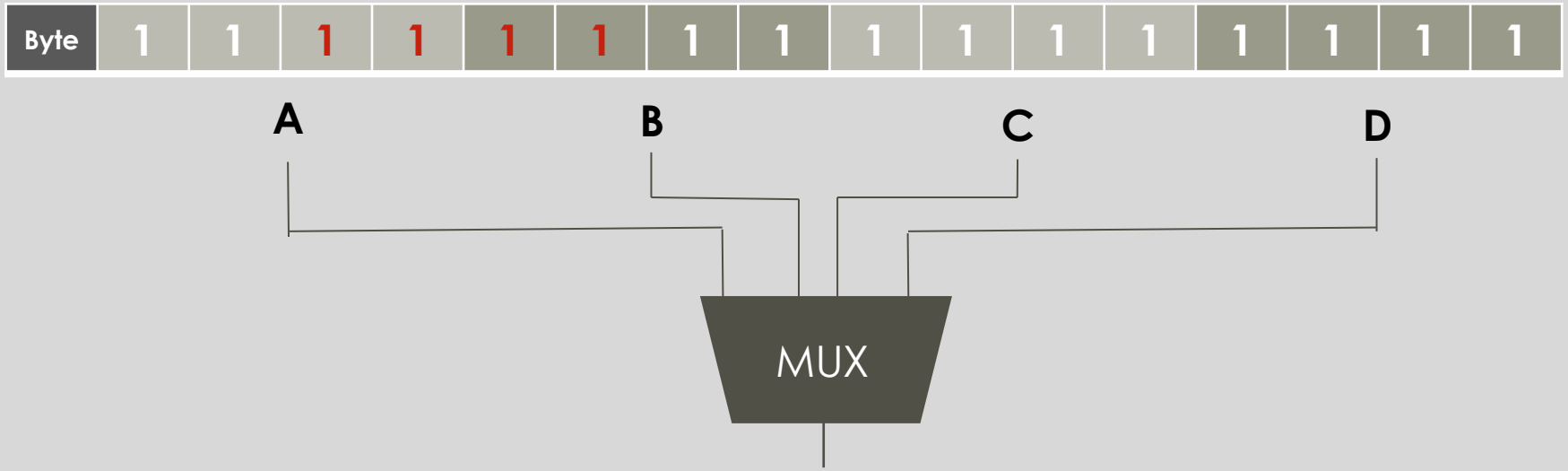
Valid addresses for:

char: divisble by **1** (which means any)
int (or unsigned int): divisible by **4**
size_t or pointers: divisible by **8**

VALID FOR	char	int	size_t
0x1003	✅	❌	❌
0x1004	✅	✅	❌
0x1008	✅	✅	✅



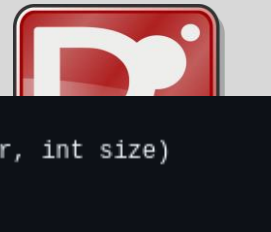
CTRL pin 1	CTRL pin 2	Result
0	0	**A**
0	1	**B**
1	0	**C**
1	1	**D**



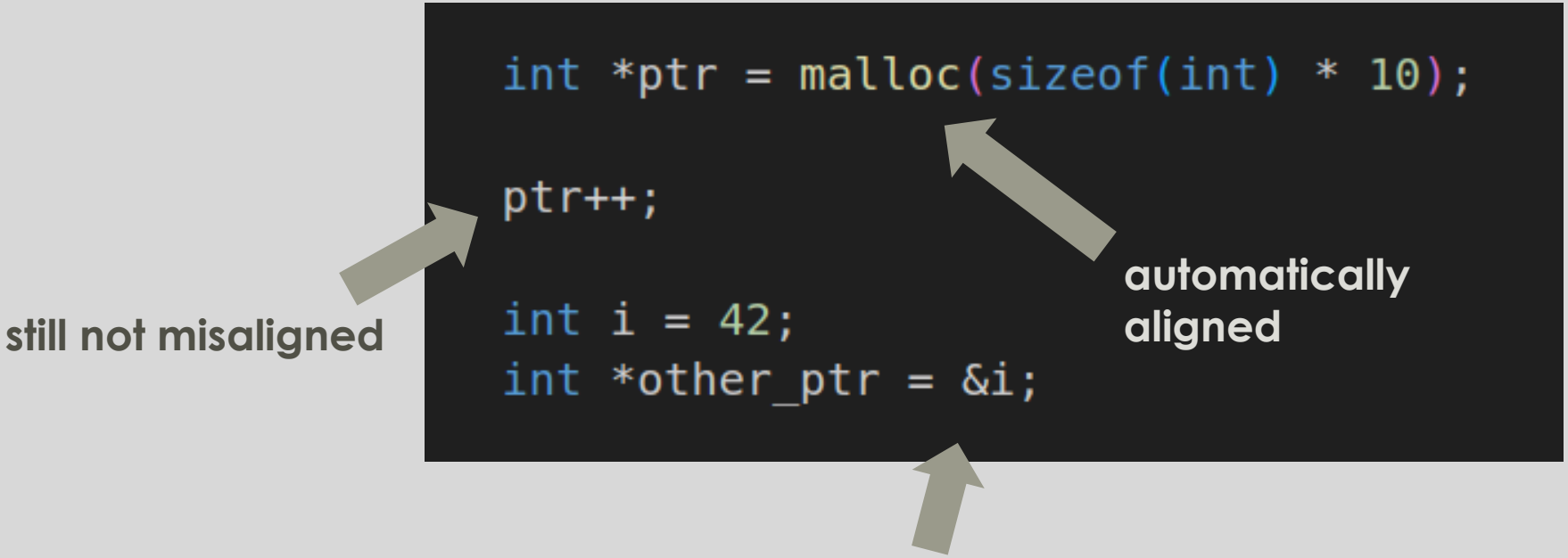
CTRL pin 1	CTRL pin 2	Result
0	0	A
0	1	B
1	0	C
1	1	D

Is there a simpler way to do this?

```c
void print_array(int *arr, int size)
{
    int i = 0;
    while (i < size)
    {
        printf("%d ", *(arr + i));
        i++;
    }
    printf("\n");
}
```




```c
void print_array(int *arr, int size)
{
    int i = 0;
+   if (arr == NULL)
+   {
+       dprintf(2, "Error: NULL pointer passed to print_array\n");
+       abort();
+   }
+   if ((unsigned long)arr % sizeof(int) != 0)
+   {
+       dprintf(2, "Error: Array is not aligned to int boundary.\n");
+       abort();
+   }
    while (i < size)
    {
        printf("%d ", *(arr + i));
        i++;
    }
    printf("\n");
}
```



How does something get misaligned?

```c
int *ptr = malloc(sizeof(int) * 10);

ptr++;

int i = 42;
int *other_ptr = &i;
```

still not misaligned

automatically aligned

this is also aligned correctly

very dangerous function

```c
char j = 'a';
int *lol = &j;
```

danger

```c
void dangerous_function(void *ptr)
{
    int *int_ptr = (int *)ptr;
    *int_ptr = 42;
    //hehe
}
```

Is it possible to prevent misalignment to happen?

YES, but then…

You need to enforce that T* can only point &T

AND

Disallow pointer casting

```
char j = 'a';
int *lol = &j;
```

danger

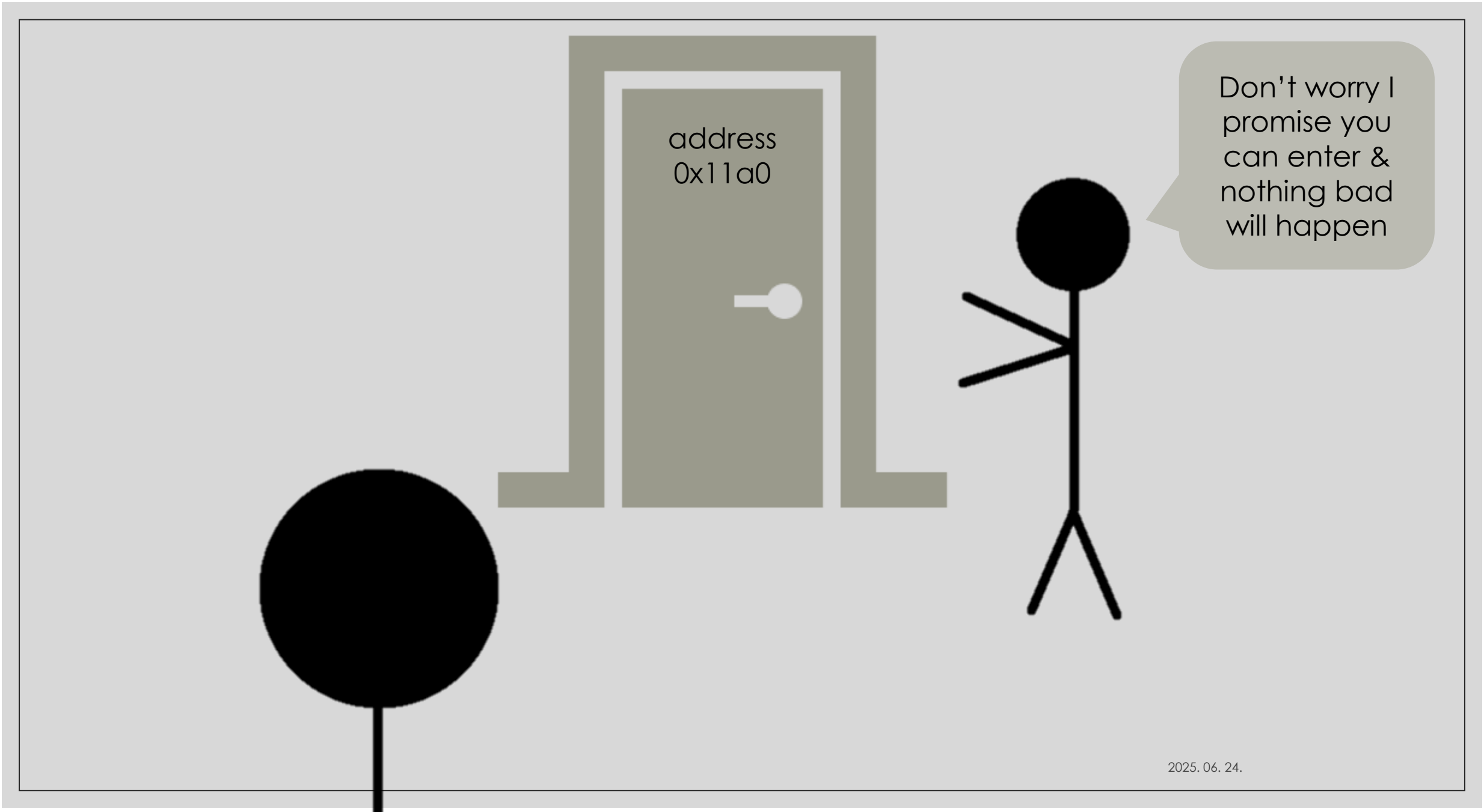
very dangerous function

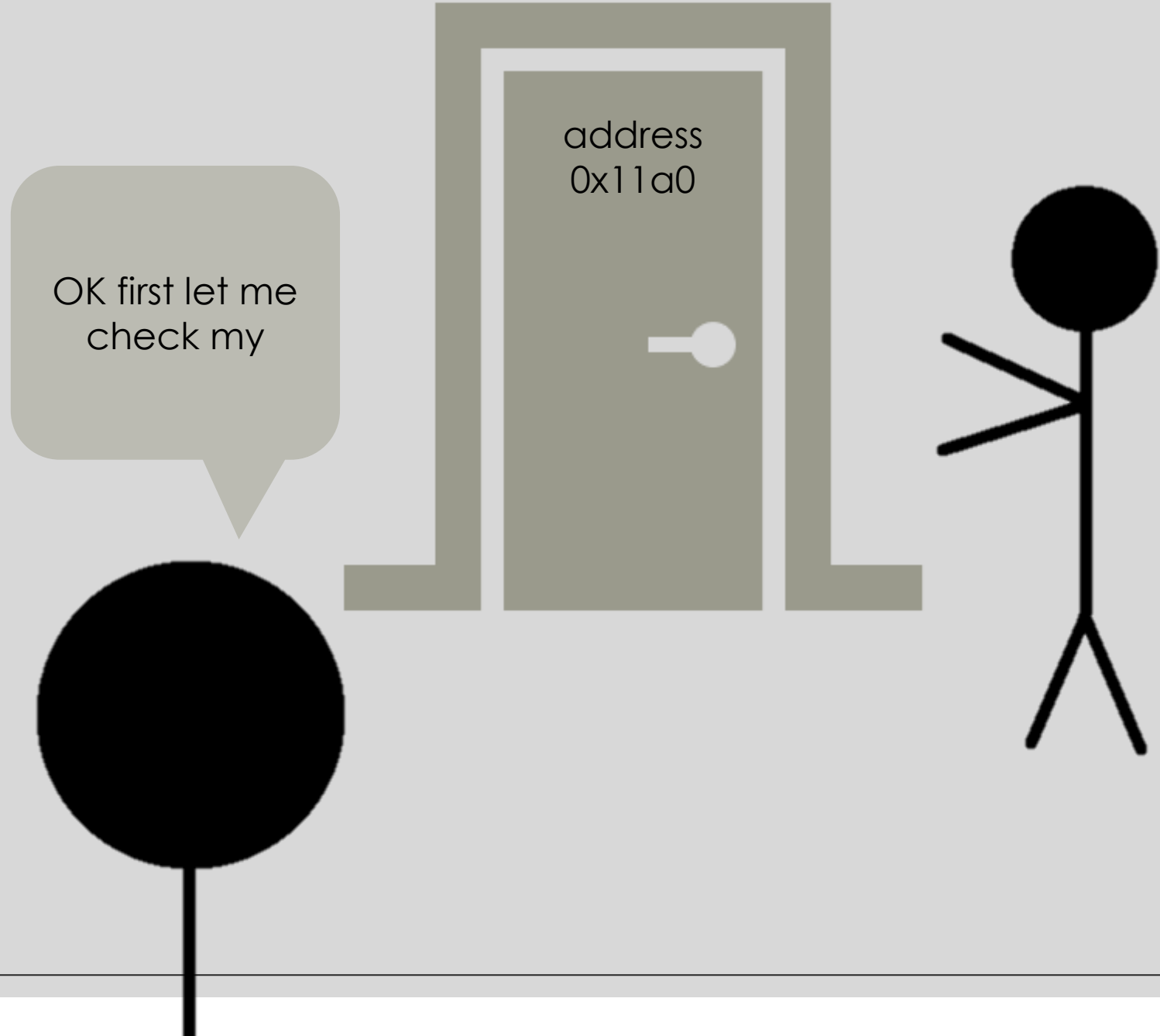
```
void dangerous_function(void *ptr)
{
    int *int_ptr = (int *)ptr;
    *int_ptr = 42;
    //hehe
}
```

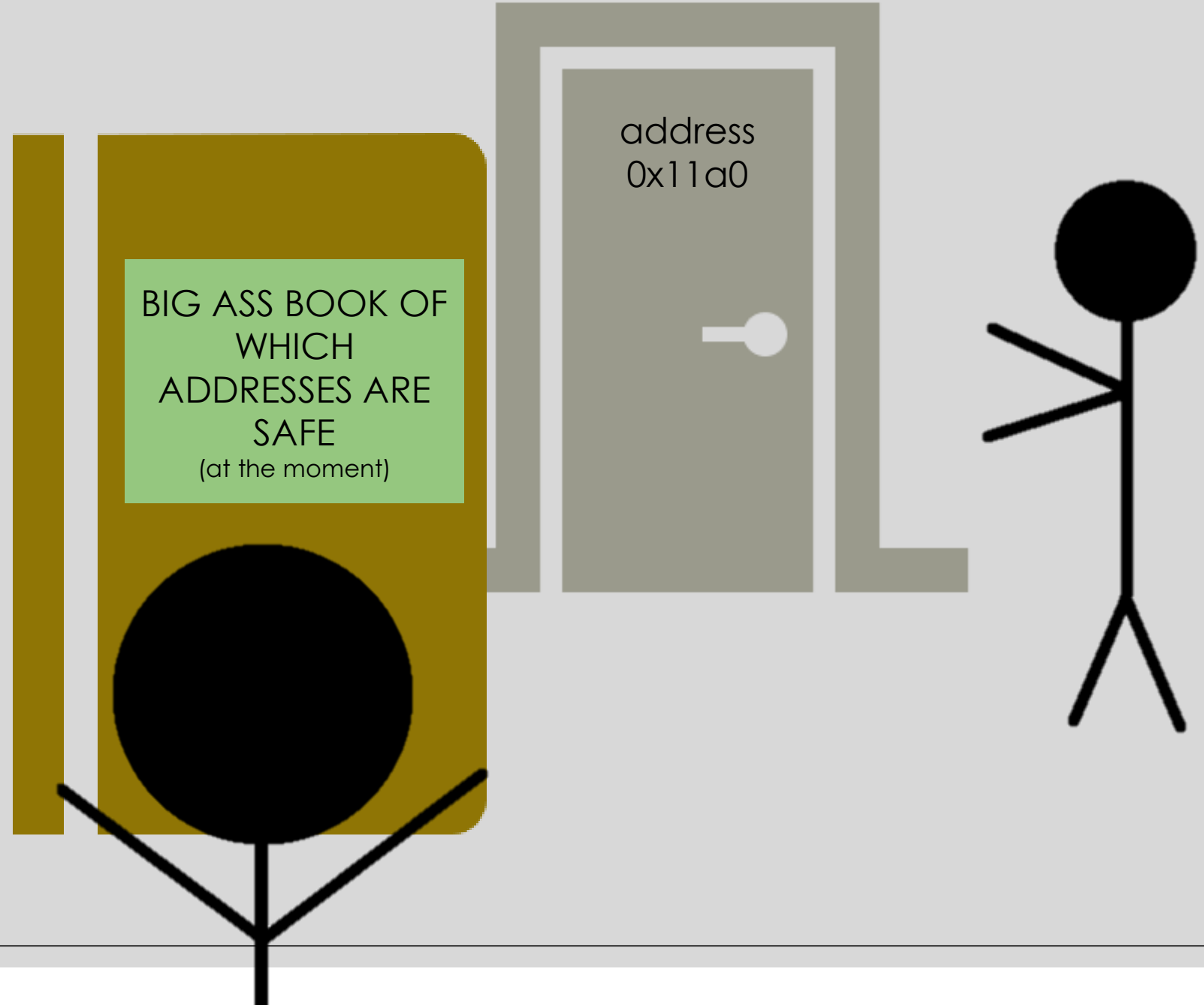
Invalid value 3: an address of an object after the end of its lifetime

AKA how to prevent „use after free”?

?

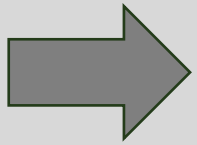

address
0x11a0
Don’t worry I promise you can enter & nothing bad will happen
2025. 06. 24.


address
0x11a0
OK first let me
check my


address
0x11a0
BIG ASS BOOK OF WHICH ADDRESSES ARE SAFE
(at the moment)

Compiler optimizations

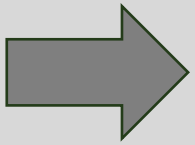
```
int sum = 0;
int i = 0;
while (i < 4)
{
    sum += array[i];
    i++;
}
```



```
int sum = 0;
sum += array[0];
sum += array[1];
sum += array[2];
sum += array[3];
```

Loop unrolling

```c
int sum = 0;
int i = 0;
while (i < n)
{
    sum += array[i];
    i++;
}
```



```c
int sum = 0;
int i = 0;
while (i + 3 < n)
{
    sum += array[i];
    sum += array[i + 1];
    sum += array[i + 2];
    sum += array[i + 3];
    i += 4;
}

while (i < n)
{
    sum += array[i];
    i++;
}
```

```c
void    rush(int x, int y)
{
    int max_y;
    int current_y;

    max_y = y;
    current_y = 1;
    while (current_y <= max_y)
    {
        if (current_y == 1)
        {
            first_line(x);
            ft_putchar('\n');
        }
        else if (current_y == max_y)
        {
```

- This loop **might be infinite** (on y == INT_MAX)
- Infinite loops cannot be unrolled
- If compiler assumes no overflow possible, this is not a problem anymore

So compiler works with the assumption that

N + 1 > N

It thinks after INT_MAX comes INT_MAX + 1, then INT_MAX + 2 till infinity

Because the Standard allows literally everything to happen at UB, for te sake of convenience, when writing code compiler most of the time assumes that whatever would cause UB is **impossible**. Because then:

- If it doesn't happen, we are fine
- If it anyway happens, it will have some weird consequence we are absolutely not prepared of - but that is also fine!

So:

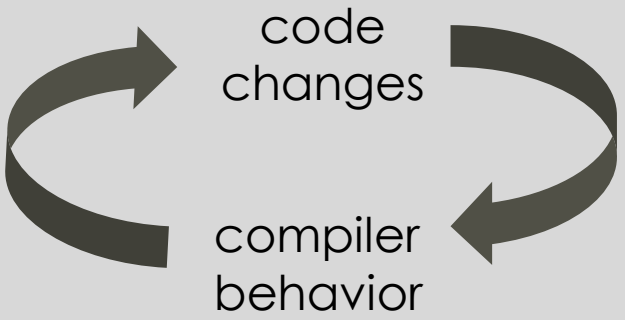
- It does not have to ever think about overflow
- Does not have to care about NULL checks
- Can assume an environment where data races are impossible
- …

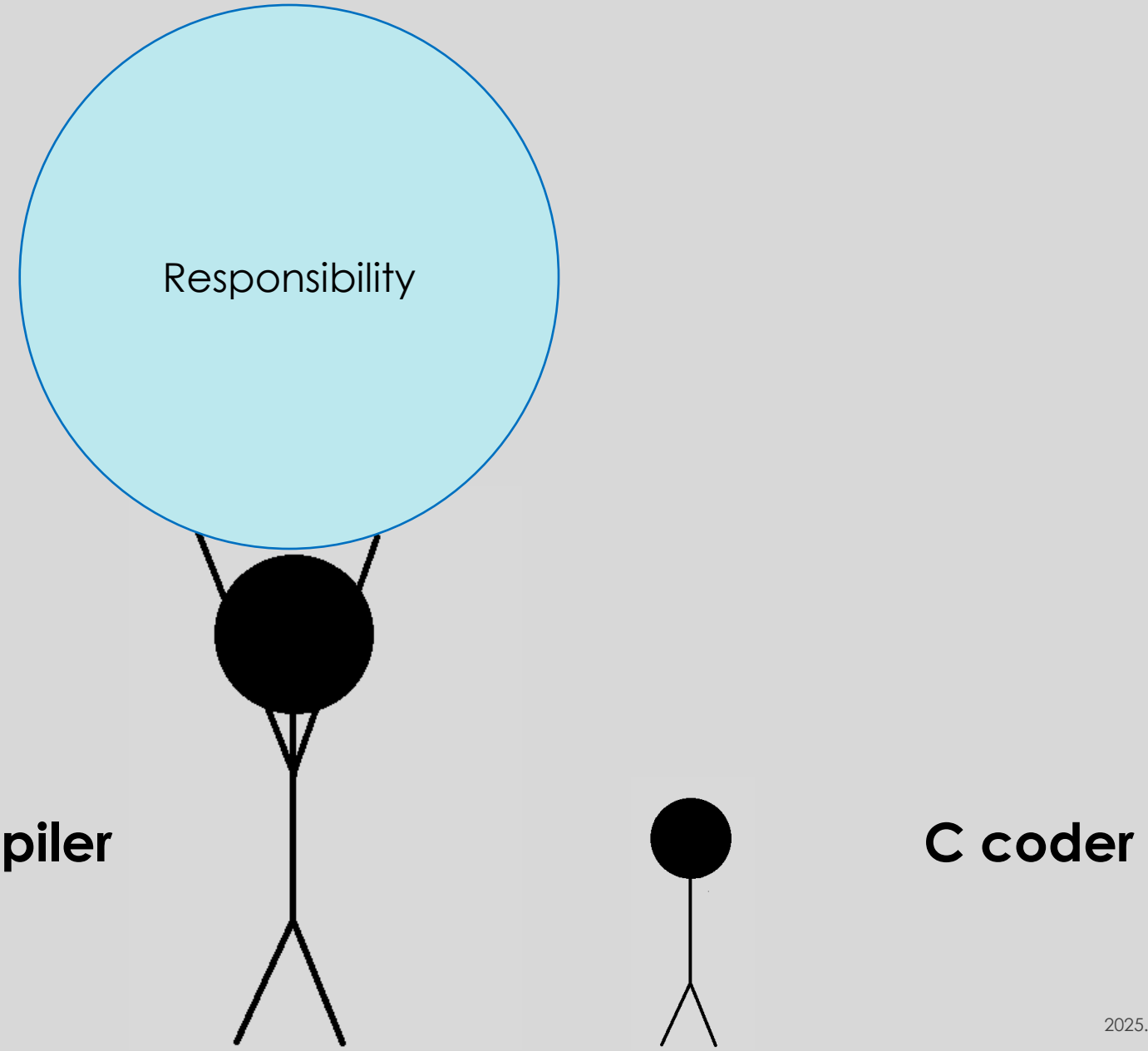
Reasons for keeping UB

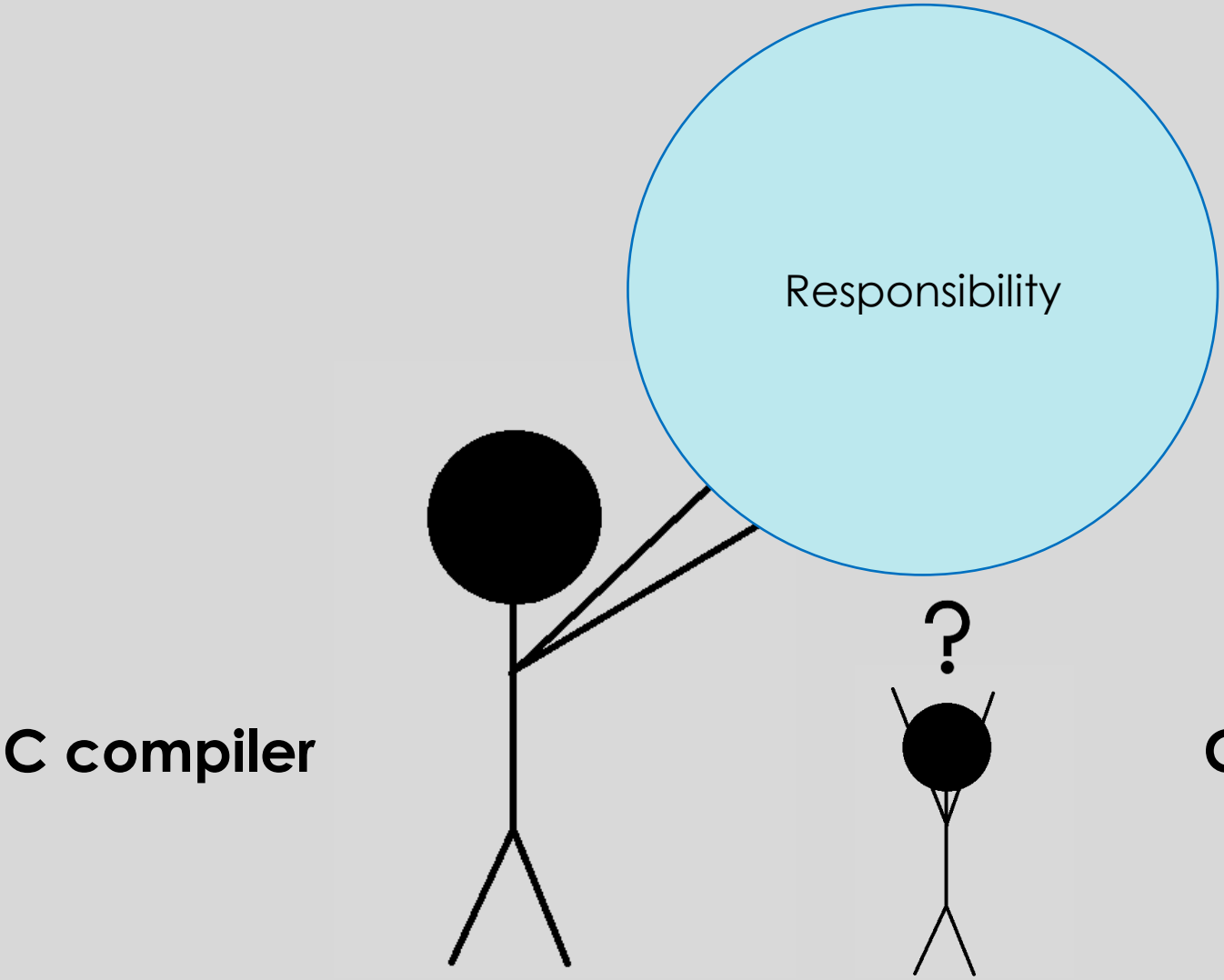
1. Convenience
2. Speed
3. History
4. Adaptability

Historical reasons

- By the time of writing the first C standard, different compilers were **already in standard use**
- Certain important projects were written **assuming specific compiler behavior**
- Forcing them to change compiler would have required serious refactoring
- Instead, the standard **incorporated** already existing behaviors
- If different compilers handled a case differently → rather left UB

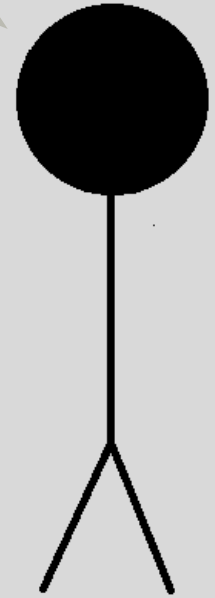


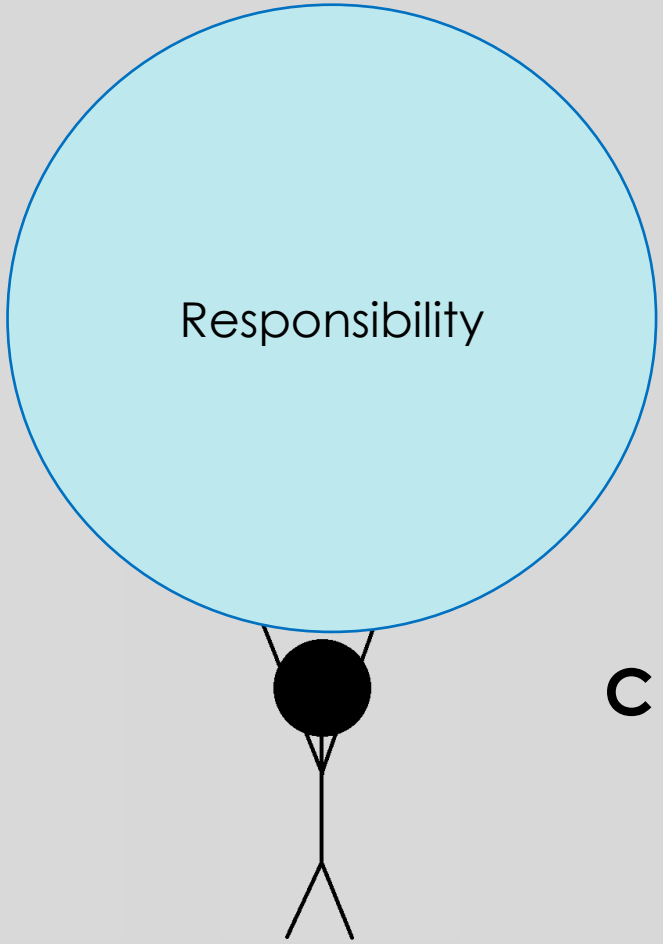

Responsibility
C compiler
C coder



C compiler

C coder





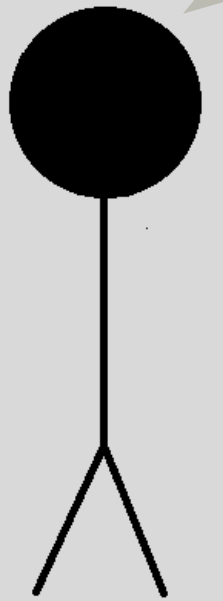
C compiler

C coder

What do you mean what comes after 2147483647?
It's 2147483648 of course, then 2147483649…
It's cause an int has infinite size
Also NULL pointers stop existing the moment they are dereferenced

Also data races
happen and ever
always ends w
newline, a signal
for a SIGSEGV wi
return becau
developers knov
they are doing a
are all responsible

C compiler



2025. 06. 24.

Chapter VI.
Let’s introduce some controversy

How to treat UB at 42?

- Your functions should not quit unexpectedly (segmentation fault, bus error, double free, etc) apart from undefined behaviors. If this happens, your project will be considered non functional and will receive a `0` during the evaluation.

Main question:

- Do we want to be true to spirit of C (write code that doesn't consider edge cases but becomes very efficient when everybody respects the rules)
- Or do we want to establish rules for safety

Resources for further reading

- **LLVM Project Blog – What every C programmer should know about Undefined Behavior**: https://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html
- **A Guide to Undefined Behavior in C and C++**: https://blog.regehr.org/archives/213
- **C++ programmer's guide to Undefined Behavior**: https://pvs-studio.com/en/blog/posts/cpp/1129/
- **Using the language D in Quantum Break**: https://dconf.org/2016/talks/watson.html