Software Engineering Course Notes

Giosuè Castellano, Fabio Pernisi

January 2024

Contents

1 Boolean Logic and Integers

1.1 Boolean Logic

1.1.1 Boolean values

- False = 0
- True = 1

1.1.2 Boolean variables:

$$x \in \{0, 1\}$$

1.1.3 Boolean expressions

Boolean operators:

operator	math	pseudocode	C code	logic gate
negation	$\neg$	not	!	[TBD]
conjunction	$\wedge, \times$	and	&&, &	[TBD]
disjunction	$\vee, +$	or	——, —	[TBD]

Example expression:

$$(a \text{ and } b) \text{ or } (\text{not } c)$$

Example function:

$$f(a, b, c) := (a \text{ and } b) \text{ or } c$$

1.1.4 Truth Tables

- The truth table for the NOT operator is as follows:

x	$\neg x$
0	1
1	0

- The truth table for the AND operator is as follows:

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

- The truth table for the OR operator is as follows:

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

1.1.4.1 More operators!

- Exclusive OR (XOR), NAND, and NOR are additional binary Boolean operators with their respective truth tables.
- Unary Boolean operators include constants (always true, always false), identity, and NOT.

Truth table for XOR:

x	y	x xor y
0	0	0
0	1	1
1	0	1
1	1	0

Truth table for NAND:

x	y	x nand y
0	0	1
0	1	1
1	0	1
1	1	0

Truth table for NOR:

x	y	x nor y
0	0	1
0	1	0
1	0	0
1	1	0

Example of expressions

- $(a \wedge b) \vee (\neg c)$
- $f(a, b, c) := (a \wedge b) \vee c$
- $w := \neg a$
- $z := a \wedge (\neg b)$
- $z := (\neg a) \vee (b \wedge c)$

1.1.4.2 Questions

Q: How many distinct unary Boolean operators?
A: one? (NOT)

Actually, we have 4 deterministic unary operators in total (counting 3 trivial unary operators):

always false

x	$\emptyset$
0	0
1	0

always true

x	1
0	1
1	1

identity

x	x
0	0
1	1

NOT

x	not x
0	1
1	0

1.1.5 Binary Operators

The number of distinct binary operators corresponds to the number of different truth tables that can be constructed with two Boolean inputs. There are $2^{2^2} = 16$ possible truth tables for binary operators.

x	y	op(x, y)
0	0	?
0	1	?
1	0	?
1	1	?

1.1.5.1 Representing binary operators

Common binary Boolean operators include AND, OR, and NOT. These are sufficient to represent all possible binary operations.
NAND and NOR are known as *universal gates* because they can be used to represent any Boolean operation.

Examples
x nand y = not (x and y)
x xor y = (x or y) and (not (x and y))

1.1.5.2 Proof via truth table

To prove that two expressions are equivalent, we can show that their truth tables are identical.

x	y	x XOR y	$(x \lor y) \land \neg(x \land y)$
0	0	0	0
0	1	1	1
1	0	0	0
1	1	1	1

The expressions are equivalent because their truth tables match.

1.1.6 Boolean identities I

- x and $0 = 0$
- x or $1 = 1$
- x and $1 = x$
- x or $0 = x$
- x or $x = x$
- x and $x = x$

1.1.7 Boolean identities II

- AND is commutative:
$$x \text{ and } y = y \text{ and } x$$
- AND is associative:
$$x \text{ and } (y \text{ and } z) = (x \text{ and } y) \text{ and } z$$
- OR is commutative:
$$x \text{ or } y = y \text{ or } x$$
- OR is associative:
$$x \text{ or } (y \text{ or } z) = (x \text{ or } y) \text{ or } z$$

1.1.8 Boolean identities III

- Distributivity (AND over OR):

$$x \text{ and } (y \text{ or } z) = (x \text{ and } y) \text{ or } (x \text{ and } z)$$

- Distributivity (OR over AND):

$$x \text{ or } (y \text{ and } z) = (x \text{ or } y) \text{ and } (x \text{ or } z)$$

- De Morgan's law (1):

$$(\text{not } x) \text{ and } (\text{not } y) = \text{not } (x \text{ or } y)$$

- De Morgan's law (2):

$$(\text{not } x) \text{ or } (\text{not } y) = \text{not } (x \text{ and } y)$$

1.1.9 Satisfiability problem

The satisfiability problem, often referred to as SAT, is a problem of determining if there exists an interpretation that satisfies a given Boolean formula. In other words, it is about finding an assignment to variables that makes the entire expression true.

1.1.9.1 Example Problem

Given a Boolean expression, our goal is to find a value for each variable such that the expression evaluates to true.
For the expression $x1 \wedge ((\neg x2 \vee x3) \wedge (\neg x3))$, the truth table is:

$x1$	$x2$	$x3$	$x1 \wedge ((\neg x2 \vee x3) \wedge (\neg x3))$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

A solution to the SAT problem for this expression is $x1 = 1$, $x2 = 0$, $x3 = 0$.

Definitions

- **Variable:** x_j for some $j \in J \subseteq \mathbb{N}$.
- **Literal:** Either x_j or $\neg x_j$ for some $j \in J$.
- **Disjunctive clause:** A disjunction of literals, e.g., $\bigvee_{j\in J^0} \neg x_j \vee \bigvee_{j \in J^1} x_j$ for some $J^0, J^1 \subseteq J$.
- **Conjunctive clause:** A conjunction of literals, e.g., $\bigwedge_{j\in J^0} \neg x_j \wedge \bigwedge_{j \in J^1} x_j$ for some $J^0, J^1 \subseteq J$.

1.1.10 Conjunctive normal form

Conjunctive Normal Form is a way of structuring logical expressions as a conjunction (AND) of disjunctive clauses (ORs of literals).

$$\bigwedge_{i \in I} \left(\bigvee_{j \in J_i^0} \neg x_j \vee \bigvee_{j \in J_i^1} x_j \right)$$

where $J_i^0, J_i^1 \subseteq J$ and $I \subseteq \mathbb{N}$.

1.1.10.1 Examples of CNF

- $(x1 \vee x2) \wedge (x3 \vee x4) \wedge (x5 \vee x6)$
- $(\neg x2 \vee \neg x4 \vee \neg x6) \wedge (x1 \vee x5 \vee x6 \vee x7 \vee x9) \wedge (\neg x1 \vee \neg x2 \vee \neg x3)$

1.1.11 Disjunctive normal form

The disjunctive normal form (DNF) is a disjunction of conjunctive clauses:

$$\bigvee_{i \in I} \left(\bigwedge_{j \in J_i^0} \neg x_j \wedge \bigwedge_{j \in J_i^1} x_j \right),$$

where $J_i^0, J_i^1 \subseteq J \subseteq \mathbb{N}, \forall i \in I \subseteq \mathbb{N}$

1.1.11.1 Examples:

- $(x1 \wedge x2) \vee (x3 \wedge x4) \vee (x5 \wedge x6)$
- $(x1 \wedge (\neg x2)) \vee (x3 \wedge (\neg x4))$

1.1.12 Theorems

- Every Boolean expression can be put into CNF
  - For every Boolean expression with n variables and k literals using operators {NOT, AND, OR}, there exists an equivalent CNF with $n + k$ variables $3k$ clauses and $7k$ literals at most.
  - Satisfiability for a CNF ("SAT") is hard.
- Every Boolean expression can be put in DNF
  - For every Boolean expression with n variables and k literals using operators {NOT, AND, OR}, there exists an equivalent DNF with n variables and $n \times 2^n$ literals at most.
  - Satisfiability for a DNF is trivial.

1.1.12.1 Example:

$(x2 \wedge (\neg x4) \wedge (\neg x6)) \vee (\neg x1 \wedge x5 \wedge x6 \wedge x7 \wedge x9) \vee ((\neg x1) \wedge (\neg x2) \wedge (\neg x3)) \vee (x4 \wedge x5 \wedge x6)$

1. Take any clause, e.g. $(x2 \wedge (\neg x4) \wedge (\neg x6))$.
2. Set $x2 = 1, x4 = 0, x6 = 0$.
3. Done.

1.2 Integer Arithmetic

- Computers are made out of Boolean gates
- But we want to represent numbers other than 0 and 1
- How do we proceed?
- Consider Booleans as binary digits (*bits*)
- Group them together to form numbers in base 2

1.2.1 Base-10 numbers

In base 10 (decimal), we have 10 digits: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
Using one digit, we can count to 9:

0 1 2 3 4 5 6 7 8 9

Then we need more digits:

10 11 12 13 14 15 16 17 18 19 20 21 22 23 . . .

1.2.2 Base-2 numbers

In base 2 (binary), we have 2 digits: {0, 1}
Using one digit, we can count to 1:
0 1

Then we need more digits:
10 11 100 101 110 111 1000 1001 . . .
If we wanted to count from 0 to 15, we may decide to use 4 digits:
0000 0001 0010 0011 0100 0101 0110 0111
1000 1001 1010 1011 1100 1101 1110 1111

1.2.2.1 Example

$1001_2 = ?$
$= 1 \times 8 + 0 \times 4 + 0 \times 2 + 1$
$= 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
$= 9$

Note:

- rightmost / least-significant bit is called bit 0
- leftmost / most-significant bit is called bit $n - 1$

1.2.3 Fixed bit width

In computer systems, integers have a fixed number of bits, determining their size and the range of values they can represent.

1.2.3.1 Commonly Used Integer Types and Their Bit Widths

bits	a.k.a.	C type
8	byte	`uint8_t`
		`unsigned char`
32		`uint32_t`
		`unsigned int` (Windows, Linux, BSD, macOS)
64		`uint64_t`
		`unsigned long` (Linux, BSD, macOS)
		`unsigned long long` (Windows)

1.2.4 Integers in hardware and in programming languages

- Most computers support 8, 16, 32, and 64-bit arithmetic natively.
- Arithmetic on integers larger than native size can be done in software, which is slower.
- In C, integer types are fixed-size, and arbitrary-sized integers require specific libraries.
- Python supports integers of arbitrary size, though with a performance penalty for very large numbers.

bits	largest integer $= 2^{\text{bits}} - 1$
8	255
16	65,535
32	4,294,967,295
64	18,446,744,073,709,551,615
128	340,282,366,920,938,463,463,374,607,431,768,211,455

$$1 \text{ decimal digit} = \log_2 10 \text{ bits} \approx 3.3219 \text{ bits}$$

1.2.5 Operations with integers

Integer operations in binary are analogous to the decimal operations learned in school.

Binary Addition Example

```
  0 1 0 1 0 0 1 1
+ 0 1 1 0 0 0 0 1
-----------------
  1 0 1 1 0 1 0 0
```

1.2.5.1 Operation complexity

- Addition and subtraction are straightforward, like their decimal counterparts.
- Multiplication is more complex, involving shifting and adding.
- Division is the most complex, requiring iterative subtraction and shifting.

1.2.6 Signed integers

Representing negative numbers in binary requires special encoding:

1.2.7 Sign-Magnitude Representation

- One bit is reserved for the sign (most significant bit).
- Zero has two representations: $+0$ and -0.
- Arithmetic with sign-magnitude is not straightforward.

1.2.8 One's Complement Representation

- The sign bit is also the most significant bit.
- Negative numbers are represented by flipping all bits of the positive value.
- Zero still has two representations: $+0$ and -0.
- Arithmetic is simpler than sign-magnitude but still not straightforward.

1.2.9 Signed integers: two's complement

It is the approach adopted by most current computers.

- In two's complement representation, a negative number x is represented by $2^n - x$ for an n-bit number.
- The most significant bit (MSB) is the sign bit, where 1 indicates a negative number.
- Flipping the sign requires inverting all bits and adding 1.
- This representation gives zero a single unique representation.
- Arithmetic operations for addition and subtraction are the same as for unsigned integers.

1.2.10 4-bit signed integers (two's complement)

b3	b2	b1	b0	unsigned	signed
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	2	2
0	0	1	1	3	3
0	1	0	0	4	4
0	1	0	1	5	5
0	1	1	0	6	6
0	1	1	1	7	7
1	0	0	0	8	-8
1	0	0	1	9	-7
1	0	1	0	10	-6
1	0	1	1	11	-5
1	1	0	0	12	-4
1	1	0	1	13	-3
1	1	1	0	14	-2
1	1	1	1	15	-1

1.2.11 Example:

signedness	decimal	binary
unsigned	$2 + 11 = 13$	$0010_b + 1011_b = 1101_b$
signed	$2 + (-5) = -3$	$0010_b + 1011_b = 1101_b$

1.2.12 Range of values for different bit sizes:

bits	-2^{bits-1} (min)	$2^{bits-1} - 1$ (max)
8	-128	127
16	-32768	32767
32	-2,147,483,648	2,147,483,647
64	$\approx -9.10^{18}$	$\approx 9.10^{18}$
128	$\approx -2.10^{38}$	$\approx 2.10^{38}$

2 Instructions and Memory

2.1 Integers (Continued)

Two's complement:

- Given a single n-bit pattern,
  - let u be its unsigned value
  - let s be its signed value,
- If bit $(n-1) = 0$, then:
  - $s := u$
- If bit $(n-1) = 1$, then:
  - $s := u - 2^n$

2.1.0.1 4-bit example:

- bit $(n-1) = 0 \Rightarrow s = u$
- bit $(n-1) = 1 \Rightarrow s = u - 2^n$

2.1.0.2 In general:

bit pattern	**00 . . . 0**	**01 . . . 1**	**10 . . . 0**	**11 . . . 1**
unsigned u	0	$(2^{n-1}) - 1$	(2^{n-1})	$(2^n) - 1$
signed s	0	$(2^{n-1}) - 1$	$-(2^{n-1})$	-1

- Unsigned: $u \in \{0, \ldots, (2^n) - 1\}$
- Signed: $s \in \{-(2^{n-1}), \ldots, -1, 0, \ldots (2^{n-1}) - 1\}$

2.1.0.3 Conversely:

- if $s \geq 0$
  - represent with bit pattern of $u = s$.
- if $s < 0$
  - represent with bit pattern of $u = 2^n - |s|$.
- if $s \notin \{-(2^{n-1}), \ldots, (2^{n-1}) - 1\}$
  - cannot represent, need larger n

2.1.1 Sign extension

Let us represent $s = -5$ in n-bit signed binary (two's complement):
$u = 2^n - |s| = 2^n - 5$

n	s	u	bit pattern
4	-5	11	1011
5	-5	27	11011
6	-5	59	111011
7	-5	123	1111011
8	-5	251	11111011
9	-5	507	111111011
10	-5	1019	1111111011
11	-5	2043	11111111011
12	-5	4091	111111111011

2.1.2 Increasing the number of bits

To convert an n-bit number to an $(n + k)$-bit number $(k \geq 0)$:

- Unsigned:
  - Additional high-order (leftmost) bits are set to zero
- Signed ("sign extension"):
  - Additional high-order (leftmost) bits are set to the value of bit $(n - 1)$

2.1.3 Base 16

Hexadecimal digits: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f$

- Pros:
  - Directly maps to binary numbers:
    - hex 12f3 = binary 0001 0010 1111 0011
  - More compact than binary
  - Directly maps to bytes:
    - two hex digits = one byte
- Cons:
  - Not human-friendly (esp. for arithmetic)

2.2 Characters and Text

How do we map bit patterns to characters in order to form text?

- Many standards
- Some similarities
- Some incompatibilities

2.2.1 ASCII (1963-)

- American Standard Code for Information Interchange
- Each character stored stored in 1 byte (8 bits, 256 possible characters)
- 128 standardized characters
- Many derivatives specify the remaining 128

[TBD]

2.2.2 Unicode (1988-)

- Associates "code points" (roughly, characters) to integers
- Up to 1,112,064 code points (currently 149,186 assigned)
- First 128 code points coincide with ASCII
- Multiple possible encodings into bytes ("transmission formats"):
  - UTF-8
    - First 128 code points encoded into a single byte (backward compatible with ASCII)
    - Sets most significant bit (bit 7) to 1 to signify "more bytes needed"
    - Up to 4 bytes per code point
    - Default on BSD, iOS/MacOS, Android/Linux and on most internet communications
  - UTF-16
    - Code points are encoded by either two or four bytes
    - Default on Windows, for Java code, and for SMS

2.2.3 Unicode (1988-)

- Aims at encoding all languages:
  - including extinct ones
  - left-to-right, right-to-left or vertical
  - and more (emojis)
- Some "characters" require multiple code points (flag emojis, skin tone modifiers)
- What is even a "character"? (code point, glyph, grapheme, cluster)
- Unicode is extremely complicated
- Latest version (v15.0.0, 2022) specification is 1,060 pages

2.3 Hardware

Logic gates allow us to compute Boolean functions “instantly” (subject to physical limits). But we need many logic gates, even for simple things (like 64-bit integer division).
$\rightarrow$ We break down complex algorithms into simple steps.

2.3.1 Components in a Computer

- Logic gates
- A clock
- Memory
- Input and output devices

2.3.2 A first abstraction

- Memory is N bits $\in \{0,1\}^N$ (e.g., for 16 GB, $N \approx 128 \cdot 10^9$)
- At every clock cycle (e.g., 1.2 GHz), we update the memory:

$$x_i' \leftarrow f_i(\mathbf{x}) \quad \forall i = 0, \dots, N$$

- Some of the memory comes from input devices
- Some of the memory is sent to output devices

2.3.3 A more realistic model

- We cannot update the whole memory at every clock cycle
  - That would be $128 \times 10^9 \times 1.2 \times 10^9 = 153.6 \times 10^{18}$ B/s
  - $\approx 153,600,000,000$ GB/s
- As of 2023, memory maxes out at ≈ 800 GB/s
- Instead, at each cycle, we only read/write a tiny amount of memory
- We cannot have too many different Boolean function f_i
  - Instead, at each cycle, the computer executes one of a limited set of **instructions** in a **processor** (a.k.a. *”Central Processing Unit”*, *CPU*), e.g.
    * no memory read / write
    * 64-bit arithmetic (+, -, $\times$, ...)
    * comparison ($<$, $>$, $=$, ...)
    * branch (if, while, ...)

2.4 Instruction Set Architectures (ISA)

2.4.1 An ISA specifies:

- How the machine is organized (memory, etc.)
- What instructions are available
- How instructions are encoded into bits

2.4.2 Two major ISAs in practice:

- **x86_64** (a.k.a. x64, x86_64, AMD64): Intel® and AMD® 64-bit CPUs
- **AArch64** (a.k.a. ARM64): ARM®-based 64-bits CPUs (phones, Apple M1 & M2)

Many older and less prominent ISAs:

- x86, Itanium, ARMv7, RISC-V, PowerPC, ...

2.4.2.1 Code Example and Assembly Representation

```
 int f(int a, int b, int c)
{
   return (a * b) / c;
}
```

x86_64:

```
89 f8 89 d1 0f af c6 99 f7 f9 c3
```

f:

```
    mov eax, edi      # 89 f8
    mov ecx, edx      # 89 d1
    imul eax, esi     # 0f af c6
    cdq               # 99
    idiv ecx          # f7 f9
    ret               # c3
```

AArch64:

```
1b 01 7c 00 1a c2 0c 00 d6 5f 03 c0
```

f:

```
    mul w0, w0, w1  # 1b 01 7c 00
    sdiv w0, w0, w2 # 1a c2 0c 00
    ret             # d6 5f 03 c0
```

2.4.3 Assembly

- Assembly is the lowest-level programming language
- Assembly is in 1:1 correspondence with binary encoding of instructions
- Typically, one line per instruction

2.4.4 Instructions (x86_64)

```
f:
 mov eax, edi       # 89 f8
 mov ecx, edx       # 89 d1
 imul eax, esi      # 0f af c6
 cdq                # 99
 idiv ecx           # f7 f9
 ret                # c3
```

- **mov a, b** move $a \leftarrow b$
- **imul a, b** signed integer multiply $a \leftarrow a \times b$
- **idiv a** signed integer divide $eax \leftarrow eax/a$
- **cdq** convert double-word (32 bits) to quad-word (64 bits) sign-extend eax into $edx : eax$
- **ret** return return to calling function

2.4.5 Instructions (AArch64)

```
f:
 mul w0, w0, w1     # 1b 01 7c 00
 sdiv w0, w0, w2    # 1a c2 0c 00
 ret                # d6 5f 03 c0
```

- **mul a, b, c** multiply $a \leftarrow b \times c$
- **sdiv a, b, c** signed integer divide $a \leftarrow b/c$
- **ret** return return to calling function

2.4.6 Registers

2.4.6.1 x86_64:

```
f:
 mov eax, edi       # 89 f8
 mov ecx, edx       # 89 d1
 imul eax, esi      # 0f af c6
 cdq                # 99
 idiv ecx           # f7 f9
 ret                # c3
```

2.4.6.2 AArch64:

```
f:
 mul w0, w0, w1     # 1b 01 7c 00
 sdiv w0, w0, w2    # 1a c2 0c 00
 ret                # d6 5f 03 c0
```

- small, fixed set of variables that can be accessed instantly
- 16 (x86_64) or 31 (AArch64) general-purpose 64-bit registers
- special registers and flags (not accessible directly)
- larger registers for extended operations (e.g., non-integer numbers)

2.4.7 Registers (x86_64)

- sixteen 64-bit registers:

  ```
  rax, rbx, rcx, rdx, rsp, rbp, rsi, rdi,
  r8, r9, r10, r11, r12, r13, r14, r15
  ```

- we can access the lower 32 bits separately:

  ```
  eax, ebx, ecx, edx, esp, ebp, esi, edi,
  r8d, r9d, r10d, r11d, r12d, r13d, r14d, r15d
  ```

- we can access the lower 16 bits separately:

  ```
  ax, bx, cx, dx, bp, sp, si, di,
  r8w, r9w, r10w, r11w, r12w, r13w, r14w, r15w
  ```

- we can access the lower 8 bits separately:

  ```
  al, bl, cl, dl, bpl, spl, sil, dil,
  r8b, r9b, r10b, r11b, r12b, r13b, r14b, r15b
  ```

- we can access bits 8-15 separately for some registers:

  ```
  ah, bh, ch, dh
  ```

2.4.7.1 Example:

bits	**63..56**	**55..48**	**47..40**	**39..32**	**31..24**	**23..16**	**15..8**	**7..0**
64	rax							
32					eax			
16							ax	
8							ah	al

2.4.8 Registers (AArch64)

- thirty-one 64-bit registers:

  ```
  x0, ..., x30
  ```

- we can access the lower 32 bits separately:

  ```
  w0, ..., w30
  ```

- register 31 (`x31`, `w31`) is read-only (zero in most cases)

2.4.8.1 Example:

bits	63...56	55...48	47...40	39...32	31...24	23...16	15...8	7...0
64	**x0**							
32					**w0**			

2.5 Memory

```
int g(int *a, int *b)
{
    return *a + *b;
}
```

2.5.0.1 x86_64:

```
g:
  mov eax, DWORD PTR [rsi]
  add eax, DWORD PTR [rdi]
  ret
```

2.5.0.2 AArch64:

```
g:
  ldr w0, [x0]
  ldr w1, [x1]
  add w0, w2, w0
  ret
```

2.5.1 Memory

- From a process' perspective, memory is seen as a single long array of **bytes** (8 bits)
- Like registers, memory can be accessed in larger chunks (16, 32 or 64 bits)
- But the smallest addressable unit is the byte

2.5.2 Byte ordering

address	0	1	2	...	239	240	241	242	243	244	...	
value (hex)	ef	cd	ab	89	...	ff	a0	a1	a2	a3	42	...

- the byte at address 240 is (hex) **a0** = (decimal) 160
- the byte at address 241 is (hex) **a1** = (decimal) 161
- the byte at address 242 is (hex) **a2** = (decimal) 162
- the byte at address 243 is (hex) **a3** = (decimal) 163

Q: What is the value of the 32-bit integer at address 240?
A: It depends!

2.5.3 Byte ordering / "Endianness"

address	0	1	2	3	...	**239**	**240**	**241**	**242**	**243**	244	...
value (hex)	ef	cd	ab	89	...	ff	**a0**	**a1**	**a2**	**a3**	42	...

- "big-endian" (BE): 32-bit int at 240 is (hex) **a0 a1 a2 a3**
  - = (decimal) $160 \times 2^{24} + 161 \times 2^{16} + 162 \times 2^{8} + 163$
  - = (decimal) 2,694,947,491
- "little-endian" (LE): 32-bit int at 240 is (hex) **a3 a2 a1 a0**
  - = (decimal) $163 \times 2^{24} + 162 \times 2^{16} + 161 \times 2^{8} + 160$
  - = (decimal) 2,745,344,416
- `x86_64` is LE
- `AArch64` is LE by default (LE-only on Windows, MacOS, Linux)

2.5.4 Bit ordering

Because we cannot access individual bits on a CPU (smallest chunk is a byte), bit ordering does not matter here.
However the same problem crops up in other contexts (USB, Ethernet, Wifi, . . .)

2.5.5 Memory access notation

- In assembly, accessing memory is denoted using "`[`" and "`]`"
  - Moving the value 240 into a register:

    ```
    mov eax, 240   # eax = 240
    ldr w0, 240    # w0 = 240
    ```

  - Moving the 4 bytes of memory at address 240 into a register:

    ```
    mov eax, DWORD PTR [240]   # eax = (hex) a3a2a1a0
    ldr w0, [240]              # w0 = (hex) a3a2a1a0
    ```

3 Compilation, Operating Systems, Virtual Memory

3.1 Compilation

A compiler:

- reads source code,
- forms chunks of
  - data (constants, global variables)
  - executable machine code (functions)
- associates a *symbol* to each chunk (variable or function name)
- writes all into an "object" (".o") file (format: ELF, COFF, Mach-O)

The compiler leaves blank all *references to symbols* (incl. external symbols like global variables and global functions)

3.1.1 Linking

A linker reads "object" files and writes an executable file.

- it assigns a position in memory to every chunk of code and data
- it sets the value of the corresponding **symbol** to this position
- it resolves all references to **symbols**: replaces all references with the numeric value of the corresponding position in memory

3.1.2 Static and dynamic linking

- Static linking is performed in order to prepare an executable (.exe, ...) file.
- Dynamic linking is performed every time the executable is run
  - Object files built to be dynamically linked are called
    - shared objects (.so, Linux, MacOS), or
    - dynamically-linked libraries (.dll, Windows)
  - Typically used for
    - System libraries
    - Plugins

3.1.2.1 Why a separate linking phase?

- Separate linking simplifies compilations
  - (allows the compiler to write code using functions and variables it has not seen yet)
- It allows us to break down our code into multiple files...
  - that can be compiled separately
- It allows using code written and compiled by other people
  - saves time
  - lets us use closed-source software
- Dynamic linking allows us to use system libraries without shipping them
- It reduces the size of executables
- It helps in masking some system incompatibilities
  - (e.g., run the same .exe on Windows 10 and 11)
- It allows updating system libraries separately
- The compiler does not know the code inside external object files
  - it cannot check for mistakes based on that knowledge
  - it cannot optimize code based on that knowledge (at least for dynamic linking)
- Dynamically-linked libraries add complexity
  - (separate installation, incompatible versions, etc.)

3.1.3 Libraries

Libraries are collections of functions (and data) that can be used by different executables
Examples:

- `libjpeg`: read/write jpeg files
- `libssl`: cryptography
- `BLAS`: fast vector and matrix operations
- `Qt`: cross-platform GUI toolkit

Most languages have a *standard library*

- Distinct from the language itself, but usually necessary in any program
- The C language provides no functions.
  (All basic utilities (`strlen`, `printf`, `exit`) come from the standard library.)
- It is normally **dynamically linked**

3.1.4 Optimizing compilers

Optimizing compilers aim to improve the efficiency and performance of code during the compilation process. An example of such optimization is demonstrated through a simple C program:

```
int main() {
    int r = 0;
    for (int i = 0; i < 1000000; i++) {
        r = r + 2;
    }
    return r;
}
```

Instead of executing the loop one million times at runtime, the compiler optimizes the code by pre-calculating the final value of the variable `r`. This is achieved by a technique known as *constant folding*, where the compiler evaluates the result of constant expressions at compile time rather than at runtime.

3.1.4.1 Note

- "Optimal" = "best"
- "Optimizing" = "going towards the best possible result"
- Do not say: "I made my code *more optimal*"
- Do say: "*I optimized my code some more*"
  or
  "*I made my code faster*"

3.2 Operating Systems

The operating system (OS) manages the computer and provides services to applications.

3.2.1 Components

- The kernel handles:
  - most of the boot process (what happens upon power on)
  - memory allocation and sharing
  - input/output devices, through "drivers" (often dynamically loaded)
  - application coexistence and cooperation
- Optionally:
  - Standard libraries for some languages (C, C++, .NET, Swift, . . .)
  - Some additional common libraries
  - User interface (UI): command-line (CLI), graphical (GUI)
  - Some tools: CLI utilities, compilers, settings/configuration apps

3.2.1.1 Popular OSs:

- Windows
- MacOS, iOS (base OS: Darwin, kernel: XNU)
- Android, SteamOS (kernel: Linux)

3.2.1.2 Other current OSs:

- Debian, Ubuntu, Suse, Fedora, Arch, RHEL, AL2 (base OS: GNU, kernel: Linux)
- FreeBSD, OPNsense, TrueNAS, pfSense (base OS & kernel: FreeBSD)
- OpenBSD

All the above except Windows are descendants from "Unix"

3.2.2 Example: File Handling in C on Linux Systems

In C programming on Linux systems, file operations are commonly handled using functions provided by the standard library. For instance, to open a file for reading, the `fopen` function is used:

```
FILE *f = fopen("my\_file.txt", "r");
```

The `fopen` function is part of the standard library and performs the following actions:

- Calls Unix-specific `open()` function, also in the standard library.
- Acts as a high-level wrapper for the `open` system call in the Linux kernel.
- Allocates a structure which includes buffers for data and the file descriptor.
- Returns a file descriptor (of type `int`) upon successful opening of the file.

The `open` system call in the Linux kernel uses the filesystem and storage device drivers, such as SSD drivers, to access the file.

3.2.3 Levels of abstraction

- the processor only does elementary operations (move 64-bit to/from memory)
- the *kernel* implements basic functionality (managing devices, reading data from a file)
- the *standard library* provides more, OS-independent functionality (buffering, parsing data)
- other *libraries* may allow even more (e.g., decompressing a video file)

3.3 Memory (again)

3.3.1 Memory Virtualization

Memory virtualization is a fundamental concept in modern computing systems. It allows each process to see memory as if it were the only process running on the system. This isolation is achieved through the following mechanisms:

- Each time a process accesses memory, the hardware translates the virtual address provided by the process into a physical hardware address.
- This translation is performed using a *page table*, which is managed by the operating system's kernel.

3.3.1.1 An example

Consider the C code example provided:

```
#include <stdio.h>

int the_number = -1;

int main() {
    scanf("%d", &the_number);
    return 0;
}
```

The assembly instructions corresponding to the `scanf` function call might look like this:

```
# x86_64 Assembly
mov eax, DWORD PTR [4100]
```

3.3.1.2 Page table (managed by the kernel)

Here's how the memory virtualization works in this context:

- The processor looks up virtual address 4100 in the page table.
- It identifies the page (in this case, page 1), finds the base hardware address for the page (20480), and adds the offset within the page (4).
- Consequently, the actual memory access takes place at the hardware address 20484 (20480 + 4).

The page table example shows the mapping from virtual addresses to hardware addresses:

page	virtual address	hardware address
0	0 – 4095	65536 – 69631
1	4096 – 8191	20480 – 24575
2	8192 – 12287	4096 – 8191
...	...	...

For architectures like ARM (AArch64), the assembly instruction for accessing memory would look different:

```
# AArch64 Assembly
ldr w0, [4100]
```

3.3.2 Page table

- the page table itself is in memory!
- at a specific hardware address
- various techniques to make page lookup faster (it is a tree, with a cache)

3.3.3 Memory allocation

- The kernel is responsible for finding free hardware addresses that are not in use by any process.
- Virtual addresses are managed by the kernel, which can:
  - Allocate specific virtual addresses as requested by a process.
  - Search for and allocate free virtual addresses that are currently unassigned.
- Suitable entries are then added to the page table by the kernel.
- The kernel returns the virtual address to the requesting process.

3.3.4 Virtual memory

Virtual memory introduces several advantages and some drawbacks:

3.3.4.1 Cons:

- The process of translating virtual addresses to physical addresses can be slow.
- Initially, any memory sharing between processes requires mediation by the kernel.

3.3.4.2 Pros:

- Simplifies memory management for processes.
- Ensures process separation, preventing processes from interfering with each other.
- Allows quick relocation of large memory blocks by simply updating page table entries.
- Facilitates fast input/output operations as devices can be mapped to virtual addresses.
- Supports the extension of memory using various methods:
  - Swap space on storage devices.
  - Memory compression techniques.
  - Overcommitting memory beyond the physical limits.

3.3.5 Function Calls and Stack Management

In C, functions use the stack to manage local variables and return addresses. The stack is LIFO (last-in, first-out).
Function f1 initializes with two `uint64_t` variables and calls `f2` and `f3` in sequence:

```
void f1(void) {
    uint64_t a, b;
    f2();
    f3();
}
```

Function f2 allocates a `uint64_t` variable and calls `f3`:

```
void f2(void) {
    uint64_t c;
    f3();
}
```

Function f3 may use stack space, which is reclaimed when it returns to `f2`:

```
int f3(void) {
    // function body
}
```

`f3` executes and eventually returns to `f2`, which then returns to `f1`. Each return unwinds the stack, removing the local variables and return addresses.

Stack After f3 Returns to f2
After `f3`'s completion, the stack contains `f1`'s `a` and `b`, along with `f2`'s `c`.

Stack After f2 Returns to f1
Once `f2` has returned, the stack holds only `f1`'s variables `a` and `b`.

Function f1 Calls f3 Again
After the completion of `f2`, `f1` invokes `f3` once more, preparing the stack for another potential set of local variables from `f3`.

Function f1 Completes

```
void f1(void) {
    // final actions of f1
    return;
}
```

Finally, `f1` returns, and the stack is cleared to its state prior to `f1`'s call.
The lifecycle of the stack from the initiation of `f1` through the nested calls and back illustrates the LIFO nature of the stack in function call management.

3.3.6 Stack pointer

- x86_64: rsp (by convention – rsp is a general register)
- AArch64: sp (mandatorily – sp is a special register)
- In both cases, the stack actually grows downwards
- Default stack size on Linux: 8 MB
  - theoretical max recursion depth: 1,000,000

3.3.7 Heap

All the memory that is not on stack is sometimes called the ”heap”.

3.4 Tools

- Windows Subsystem for Linux
- Basics
  - `cd`, `ls`, `less`
  - TAB completion
  - command-line parameters, `-h`
  - `man`
- package management
  - Debian/Ubuntu: `apt-get`
  - Fedora/Suse: `dnf`
  - MacOS: `brew`
  - Windows: `winget`

4 Compiler invocation, IDEs, build systems

4.1 Compiling

4.1.1 Historical compilers

- Proprietary
  - Intel C++ Compiler (ICC, 1970's?)
  - Microsoft Visual C++ (MSVC, 1993)
  - ARM Compiler (ARMCC, 2005)
  - AMD Optimizing C/C++ Compiler (AOCC, 2017)
- Open source
  - GNU Compiler Collection (GCC, 1987)
  - LLVM (2003–)

4.1.2 Evolution of compilers

- 2014: ARM Compiler rebased on LLVM
- 2017: AMD Compiler was always based on LLVM
- 2021: Intel C++ Compiler rebased on LLVM

4.1.3 Current major compilers

- Microsoft Visual C++
  - default on MS Windows (in MS Visual Studio)
- GCC
  - default on most open source OSs
- LLVM (for C/C++: Clang)
  - base for hardware vendor (Intel, ARM, AMD, nVidia) compilers
  - default on MacOS, iOS (in Apple X Code)
  - default for native applications on Android

4.1.4 Components of a compiler

- Front-end (parses and analyses code – language-specific)
- Intermediate representation (IR) (most code optimization happens here)
- Back-end (writes assembly or machine code – ISA-specific)

4.1.4.1 LLVM frontends:

- C and C++ (Clang), Fortran (Flang), Rust, Zig, Swift

4.1.4.2 LLVM backends:

- Intel/AMD/ARM compilers, nVidia CUDA compiler, AMD ROCm

4.1.5 LLVM IR

gibberish verbatim picture

4.1.6 Compiler invocation (1)

- As usual, use `man gcc` / `man clang` for help.
- Compile and link:

  ```
  gcc -o executable source_code.c
  ```

- Compile only:

  ```
  gcc -c -o file.o file.c
  ```

- Link only:

  ```
  gcc -o executable file.o file1.o file2.o file3.o
  ```

- Write assembly (see also:)

  ```
  gcc -S assembly.s source_code.c
  ```

- Internally, `gcc` runs other tools (assembler: `as`, linker: `ld`)

4.1.7 Compiler invocation (2)

- Enable warnings:

  ```
  gcc -Wall -c -o file.o file.c
  ```

- Enable optimization:

  ```
  gcc -Wall -O3 -c -o file.o file.c
  ```

4.1.8 Note for MacOS

Install `binutils`:

- from MacPorts `https://www.macports.org`

  ```
  port install binutils
  ```

- or from Homebrew `https://brew.sh/`

  ```
  brew install binutils
  ```

Utilities may be prefixed by a `g`:

- `objdump` → `gobjdump`

4.1.9 Tools

- `hexdump` dump hexadecimal representation of any file
  - `hexdump -C` also print ASCII for valid ASCII bytes
  - `hexdump -C |less` "pipe" output to pager
  - `hexdump -C > file.hex` write output to a file
- `readelf` print symbols in ELF object file
  - `readelf -a` print all object information
- `objdump` dump contents of object file
  - `objdump -M intel -d` disassembles object file, prints assembly code
  - `objdump -p` similar to `readelf`
- or online: `http://godbolt.org`

4.2 Editing Code

4.2.1 Applications for writing code

- Text editors, i.e. Notepad
- Code editors, i.e. emacs, vim, Notepad++, VS Code
- Integrated development environment (IDE), i.e. Microsoft Visual Studio, Apple XCode, IDE: IntelliJ IDEA (paid)

4.2.2 More code editors

- gedit
- Kate
- Sublime Text (paid)
- many more...

4.2.3 More IDEs

- PyCharm (Python, paid)
- Android Studio (paid)
- KDevelop
- QtCreator
- Dev-C++
- Spyder (Python)
- . . .

4.2.4 Code editor vs. IDE

IDE pros:

- one-click compile
- IDE aware of whole project
  - can suggest code completions from different files
- integrated tools (e.g. debugger)

IDE cons:

- Project setup takes time and effort
- “Walled garden” problem
  - By default, anyone who wants to compile your project needs the same IDE.

4.3 Build Systems

4.3.1 How do we compile a complex project?

- Option 1:

```
gcc -Wall -O3 -c -o ggml.o ggml.c
gcc -Wall -O3 -c -o ggml-alloc.o ggml-alloc.c
g++ -Wall -O3 -c -o llama.o llama.cpp
g++ -Wall -O3 -c -o common/common.o common/common.c
g++ -Wall -O3 -c -o console.o console/common/console.c
g++ -Wall -O3 -c -o grammar-parser.o common/grammar-parser.c
g++ -Wall -O3 -shared -fPIC -o common.so common/ggml-alloc.o llama.o \
    common.o console.o grammar-parser.o
```

- Option 2:
  - Put above commands in a ”shell script” file, e.g., `compile.sh`
  - Run:

```
./compile.sh
```

  - Problems:
    - Difficult to modify (e.g., change compiler options)
    - We recompile everything everytime

4.3.2 Build automation

- IDE integrated:
  - Visual Studio
  - Xcode
- Stand-alone:
  - make
  - Bazel (based on Google’s internal tool Blaze) / Buck (Facebook)
  - Ninja (Google, for Chrome)
  - CMake (uses make, Ninja,...), qmake (uses make), Meson (uses Ninja, ...)

4.3.3 Make

Example:

- The `Makefile` contains rules for compiling source files and generating the shared library `libllama.so`.

- Compilation is performed using `gcc` for C files and `g++` for C++ files, with flags `-Wall` (enable all warnings), `-O3` (optimization level 3), and `-c` (compile without linking).
- The `.o` object files are created from their respective source files.
- The `libllama.so` shared library is created by linking the object files `gmgml.o`, `gmgml-alloc.o`, `llama.o`, `common.o`, `console.o`, and `grammar-parser.o` using `g++` with flags `-shared` and `-fPIC` (Position Independent Code).
- Command `make libllama.so` triggers the compilation and linking process defined in the `Makefile`, resulting in the creation of `libllama.so`.
- Run

  ```
  make libllama.so
  ```

4.3.4 Make rule syntax

```
target: source0 source1 source2 ...
	recipe
```

Whenever one of the sources was modified after the target, run the recipe (to rebuild the target).
Otherwise, consider target up-to-date and do nothing.

4.3.5 Make variables

- **Compiler Variables:** The Makefile specifies compilers and the associated flags for compiling C and C++ files. The variables `CC` and `CXX` are used for the C and C++ compilers respectively, with `gcc` for C and `g++` for C++. The flags `-Wall` (enables all compiler's warning messages) and `-O3` (optimization level 3) are set for both compilers using `CFLAGS` and `CXXFLAGS`.
- **Object File Rules:** Rules for generating object files from source files are defined. Each rule follows the format `target: dependencies` followed by the command to create the target. For example, `gmgml.o` is compiled from `gmgml.c` and depends on header files `gmgml.h` and `gmgml-cuda.h`.
- **Use of Variables:** The `$(...)` syntax is used to reference variables. For instance, `$(CC)` and `$(CFLAGS)` in a rule's command extract the values of the `CC` and `CFLAGS` variables respectively.
- **Shared Library:** The target `libllama.so` demonstrates creating a shared library from multiple object files. It uses the `-shared` and `-fPIC` flags, with `-fPIC` indicating Position Independent Code, necessary for shared libraries.

4.3.6 Special make variables

- `$(@)` the target of the current rule
- `$<)` the first source of the current rule
- `$(^)` all the sources of the current rule

[big verbatim]

4.3.7 Static pattern rules

- Static pattern syntax:

  ```
  target0 target1 target2 ... : target-pattern : source-pattern
      recipe
  ```

- Target pattern contains %, which will match anything
- Source pattern also contains %, which is replaced by the match in target
- Example:

  ```
  some_file.o other_file.o third_file.o : %.o : %.c
      recipe
  ```

  is equivalent to:

  ```
  some_file.o: some_file.c
      recipe
  other_file.o: other_file.c
      recipe
  third_file.o: third_file.c
      recipe
  ```

[two big verbatim]

4.3.8 Phony and default targets

A "phony" target does not necessarily correspond to a file name:

```
.PHONY: clean

clean:
    rm libllama.so
```

If no target is provided to the make command, the default target is the first one. A common pattern is:

```
.PHONY: default

default: libllama.so
```

- **.PHONY Target:**
  - The ‘.PHONY‘ target is a built-in Make feature that helps to define targets that are not associated with files. Targets under ‘.PHONY‘ are always considered out of date and Make executes their recipes every time they are invoked, irrespective of any file with the same name as the target.
  - In the example Makefile, ‘.PHONY‘ is used to declare ‘default‘ and ‘clean‘ as phony targets. This ensures that ‘make default‘ will build the ‘libllama.so‘ library and ‘make clean‘ will execute the cleaning commands even if files named ‘default‘ or ‘clean‘ exist in the directory.
- **Usage of .PHONY in the Example:**
  - `default`: This phony target is usually the first in a Makefile and therefore the default when ‘make‘ is executed without arguments. It depends on `$(LIBTARGET)`, which causes the build of the ‘libllama.so‘ shared library.
  - `clean`: This target is responsible for removing all the build artifacts, such as object files and the shared library, thus cleaning the build directory. The command ‘rm -f $(COBJS) $(CXOBJS) $(LIBTARGET)‘ is invoked, which forces removal without prompting, even if there are no files to remove (the ‘-f‘ option).

4.3.9 Using shell commands

The syntax is:

```
$(shell any-shell-command)
```

For example:

```
TODAY := $(shell date)
CFILES := $(shell ls *.c)
```

4.3.10 String replacement in variables

- The syntax is:

  ```
  $(variable:pattern=replacement)
  ```

- The pattern contains %, which will match any substring
- The replacement may contain %, which will be replaced by the matched substring
- For example:

  ```
  C_FILES := $(shell ls *.c)
  O_FILES := $(C_FILES:%.c=%.o)
  ```

4.3.11 For more about make

```
# Using make
man make

# Writing Makefiles
info make
```

5 Programming Languages

5.1 Compilers vs Interpreters

5.1.1 Parsing

Parsing is the process of taking the source code and creating the corresponding abstract syntax tree (AST).
Example:

```
t = 3 * ((y * w) + x)
```

becomes:

5.1.2 Compiler vs. interpreter

- A compiler:
  - parses the source code into an AST
  - takes the AST and [...] writes the corresponding assembly / machine code
- An interpreter:
  - parses the source code into an AST
  - takes the AST and performs the corresponding operations

5.1.3 Pros and cons

Programming languages can be implemented either through compilation or interpretation, though this is not an inherent property of the language itself. Below are the advantages and disadvantages of interpreters, along with examples and explanations of how languages compile.

5.1.3.1 Advantages of interpreters:

- Eliminates the compilation step, simplifying the execution process.
- Provides portability across different platforms without the need for recompilation.

5.1.3.2 Disadvantages of interpreters:

- Requires the presence of the interpreter on the user's machine.
- Typically executes code slower than compiled native machine code.

Compiled or interpreted is *not an inherent property* of a language.

5.1.3.3 Example: Python

Python, often used as a prime example of an interpreted language due to its reference implementation (CPython), showcases the flexibility of language implementations. CPython compiles Python code to an intermediate form known as bytecode, which it then interprets. However, other implementations like PyPy use Just-In-Time compilation techniques to improve performance, reflecting the language's versatility.
Python exemplifies a hybrid approach. While CPython produces bytecode, translating Python code to an intermediate language, it also interprets the bytecode. This intermediate step adds a layer of abstraction that allows Python to be platform-independent while also incurring the typical overhead of interpretation. Moreover, alternative implementations like Cython can compile Python code into optimized C, potentially transforming Python scripts into high-performance executables.
Preferred Execution Model: Despite the flexibility, languages tend to be associated with a standard execution model based on their common implementations—compiled languages like C and interpreted ones like Python.

5.1.3.4 Compiled languages:

- C, C++
- Rust, Go, Zig
- Pascal, Fortran, COBOL

5.1.3.5 Interpreted languages:

- Python, Javascript, Lua
- Lisp, Perl, PHP, R, Ruby, VBScript

[TBD]

5.1.4 Compilation Targets and Strategies

Understanding the compilation process involves knowing not only how source code is transformed but also what it is compiled into.

5.1.4.1 Compilation Targets

Different languages choose different compilation targets:

- **Nim:** Compiles to C code, which is then compiled again to machine code, leveraging the efficiency of C compilers.
- **Dart:** Targets JavaScript, allowing the compiled code to be interpreted by JavaScript engines in web browsers.
- **Java:** Compiles to bytecode for the Java Virtual Machine (JVM), which serves as an abstract processor architecture. The bytecode is portable but requires users to have the JVM for execution.

- **Python:** The CPython interpreter compiles Python into bytecode, which is immediately interpreted. This approach grants Python scripts a level of portability across platforms.

5.1.4.2 Pros and Cons of JVM

The JVM introduces both advantages and drawbacks:

- **Advantage:** JVM bytecode is platform-independent.
- **Drawback:** It necessitates that the JVM interpreter be installed on the user's machine.

5.1.5 Just-in-Time Compilation

An alternative approach to traditional compilation is Just-in-Time (JIT) compilation:

- JIT avoids long compilation times by compiling code on-the-fly, as it is needed.
- This strategy can result in execution that is both portable and fast.
- By compiling section-by-section—be it a file, function, or block of code—JIT compilers can optimize at runtime, providing speed without sacrificing flexibility.

JIT compilation offers a balance, aiming to combine the best aspects of interpretation (portability) and compilation (speed).

5.1.5.1 Conclusion

The choice of a compilation target impacts the distribution, execution, and performance of a programming language. Understanding these concepts is vital for developers to make informed decisions about code deployment and runtime environments.

5.1.6 Languages with JIT compilation

- Julia
- C#
- Java (source code compiled to JVM code; JVM code JIT compiled to native code)
- PyPy (Python)
- LuaJIT (Lua)

5.1.7 Pros and cons (summary)

	Compiled	Interpreted	Compiled to VM	Just-in-time
Needs compilation step	yes	no	yes	no
Needs interpreter / VM	no	yes	yes	yes
Portable	no	yes	yes	yes
Speed	fast	slow	in-between	slow at first, then fast

5.1.8 Language summary

- Ahead-of-time (AOT) compiled-to-machine-code languages:
  - C, C++, Rust, Go, Zig, Pascal, Fortran, COBOL
  - Nim (through C)
- Purely interpreted languages:
  - Lisp, Perl, R, VBScript
- Other:
  - Python, Lua: internally compiled to bytecode, then interpreted
  - PyPy (Python), LuaJIT (Lua): internally compiled to bytecode, then JIT compiled
  - Java, C#: explicitly compiled to bytecode (bytecode shipped to user), then JIT compiled
  - Julia: JIT compiled
  - JavaScript: interpreted and JIT compiled

5.2 Types

5.2.1 Static vs. Dynamic Type Checking

Type systems in programming languages can be broadly classified into static and dynamic type checking mechanisms, each with its approach to ensuring type correctness.

5.2.1.1 Static Type Checking

- In static type checking, the type correctness of variables is verified at compile-time.
- The compiler analyzes the source code and ensures that all type rules are followed before the code is run.
- Any type mismatch or type errors are detected and reported as compile-time errors.
- Example in C:

```
int f() {
    return "this is a string" / 5; // Compile-time error
}
```

- The advantage of static type checking is that many errors can be caught early in the development process.
- It enforces type discipline, as the code must explicitly declare and adhere to type contracts.

5.2.1.2 Dynamic Type Checking

- Dynamic type checking, on the other hand, resolves types at runtime.
- It allows more flexibility in the code, as types are enforced when operations are actually performed.
- Errors due to type mismatches are thrown during the execution of the program, which may lead to runtime exceptions.
- Example in Python:

```
def f():
    return "this is a string" / 5 # Runtime TypeError
```

- This flexibility can be advantageous for rapid prototyping and for code that requires dynamic type behavior.
- However, it also means that certain types of errors will only surface when the problematic code path is executed.

Comparative Analysis
While static type checking adds a layer of safety by catching errors early, it requires more upfront declarations and can be more rigid. Dynamic type checking offers flexibility at the cost of potential runtime errors. The choice between static and dynamic typing depends on the specific needs and context of the software project.

5.2.2 Strong and Weak Typing in Programming Languages

The terms ”strong” and ”weak” typing are used to describe how strictly a programming language adheres to type rules, especially concerning type conversions and type safety.

5.2.2.1 Weak Typing

Weak typing indicates a language where types may be implicitly converted and operations may succeed even if they do not make sense from a type-safety perspective. Example from C:

- Implicit type conversion allows assigning a floating-point number to an integer variable without an error, merely truncating the value.
- Pointer type conversions can lead to compiled code that may cause runtime errors or undefined behavior.

```
int a = -1.8; // Implicitly converted to int, truncated to -1
int *p = (int *)((long int)"abc" + 5); // Compiles, but unsafe
*p = 3; // Dereferencing p could lead to a crash
```

5.2.2.2 Strong Typing

Strong typing means the language enforces strict adherence to typing, preventing implicit, potentially unsafe conversions. Example from Python:

- Operations on incompatible types are not allowed and result in a clear error message.
- Certain type-related operations are defined within the language, such as multiplying a string by an integer to repeat the string.

```
>>> "a" + 4
TypeError: Can only concatenate str (not "int") to str

>>> "a" * 4
'aaaa'
```

The distinction between strong and weak typing can affect the reliability, safety, and clarity of code. Strongly typed languages may prevent subtle bugs at the cost of less flexibility, while weakly typed languages might allow for a wider range of behaviors at the risk of runtime errors.

5.3 Memory management

5.3.1 Manual Memory Management in C

C programming requires developers to manage memory manually, which includes allocating and freeing memory as needed.

5.3.1.1 Common Mistakes in Memory Management

When using `malloc` for dynamic memory allocation, common mistakes include:

- Not checking if `malloc` actually succeeded in allocating memory.
- Forgetting to release memory with `free`, leading to memory leaks.

5.3.1.2 Example of Poor Memory Management

The following C function `getint` does not check for allocation failure and forgets to free the allocated memory:

```
int getint() {
    char *buffer = malloc(1024);
    size_t n = fread(buffer, 1, 1023, stdin);
    buffer[n] = '\0';
    return strtol(buffer, NULL, 0);
    // Missing free(buffer) leads to a memory leak
}
```

5.3.1.3 Corrected Memory Management

A more robust version of the function includes error checking and properly frees the allocated memory.

5.3.1.4 Key Takeaways

Proper memory management is crucial in C to avoid errors and resource leaks. Developers must ensure that every `malloc` call is paired with a corresponding `free` and that all potential errors are checked and handled appropriately.

5.3.2 Automatic memory management

Automatic memory management simplifies the developer's job by abstracting the details of allocating and freeing memory. This is commonly seen in higher-level languages like Python.
Here, memory management is handled automatically through mechanisms like **reference counting** and **garbage collection**, which track and manage memory usage without direct intervention from the programmer.
Example

```
def getint():
    buffer = input()
    return int(buffer)
```

5.3.3 How does automatic memory management work?

- **Reference Counting:** It is the process of tracking how many references exist to a particular object in memory.
  - When an object is created, its reference count is set to one.
  - The reference count is incremented each time the object is referenced and decremented when the reference goes out of scope.
  - When the reference count drops to zero, the object's memory is deallocated.
- **Garbage Collection:** Complements reference counting by handling cyclic references where two or more objects refer to each other, preventing their reference counts from reaching zero.

5.3.3.1 Refcount example

Consider the following pyhton function:

```
def f():
    s = ("abc" + "def") + "ghi"
    t = s
    return t
```

1. "abc" created, refcount 1
2. "def" created, refcount 1
3. "abcdef" created, refcount 1
4. ("abc" + "def") is done, "abc" refcount 0, "def" refcount 0, both freed
5. "ghi" created, refcount 1

6. "abcdefghi" created, refcount 1
7. ("abc" + "def") + "ghi" is done, "abcdef" and "ghi" freed
8. s = "abcdefghi" done, but it is an assignment, refcount of "abcdefghi" stays 1
9. s referenced, refcount of "abcdefghi" becomes 2
10. t = s done, but it is an assignment, refcount stays 2
11. t referenced, refcount of "abcdefghi" becomes 3
12. return t is done, but it is a return, refcount of "abcdefghi" stays 3
13. s and t go out of scope, refcount of "abcdefghi" becomes 1

5.3.4 Problem with refcounting

5.3.4.1 Cycles:

```
class C:
    pass

def do_nothing():
    a = C()
    t = a
    for i in range(10000000):
        n = C()
        t.prev = t
        t = n
    a.prev = t
    return 1
```

5.3.5 Garbage collection

- keep track of all variables in scope
- keep track of all allocated blocks of memory
- every few seconds, "garbage collection"
  - look through all the variables, if they reference some memory, mark it as in-use
  - look at every block, if not referenced, free it
- **Pro:** does not suffer from cycle issue
- **Con:** memory usage can grow a lot between garbage collections
- **Con:** garbage collections pauses can block the process for a long time (making it feel unresponsive)

5.4 Other Language Features

5.4.1 Macros

Macros in C are preprocessor directives that allow for automatic generation of source code fragments.

5.4.1.1 Example and Usage

- An example macro in C that repeats arguments:

  ```
  #define THIS_5X(a) a, a, a, a, a
  int array[10] = { THIS_5X(1), THIS_5X(2) };
  ```

  This expands into an array initialization with repeated values.

- Macros can calculate the number of elements in an array:

  ```
  #define ARRAY_ELEMENTS(a) (sizeof(a) / sizeof((a)[0]))
  ```

5.4.1.2 Cautions with Macros

Macros are simple text replacements and can lead to unexpected results without careful use.

- Incorrect macro that doesn't account for operator precedence:

  ```
  #define PRODUCT_WRONG(a, b) a * b
  int a = PRODUCT_WRONG(1 + 2, 3 + 4); // Yields 11, not 21
  ```

- Corrected macro with parentheses ensuring proper precedence:

  ```
  #define PRODUCT_CORRECT(a, b) ((a) * (b))
  int a = PRODUCT_CORRECT(1 + 2, 3 + 4); // Correctly yields 21
  ```

5.4.2 Generics

Generics provide a way to write functions and data structures that can operate on any data type.

5.4.2.1 Generics in Different Languages

- In C, without built-in generics, functions must be written for each data type:

  ```
  void int_array_sort(int *array, int size);
  void float_array_sort(float *array, int size);
  ```

- Python's dynamic type checking negates the need for explicit generics:

  ```
  def array_sort(array):
      # Works for any type that supports comparison and sorting
  ```

- C++ introduces templates, which are its way of implementing generics:

  ```
  template <typename T>
  void array_sort(T *array);
  ```

  Templates allow functions and classes to operate with any type.

5.4.2.2 Implications

Understanding macros and generics is fundamental for writing reusable and efficient code. While macros offer powerful text manipulation at compile-time, generics enable type-agnostic programming, leading to more flexible and maintainable codebases.

5.4.3 Languages with generics

- C++
- C#
- Java
- Go
- Rust
- Swift
- TypeScript
- ...

5.4.4 Object-oriented programming

A compound type is any type that is defined in terms of one or more other types.

- In C:

  ```
  struct point {
      float x;
      float y;
  };
  ```

- In Python:

```
    class Point:
        def __init__(self):
            self.x = 0.0
            self.y = 0.0
```

In object-oriented programming (OOP), compound types ("classes") can have functions attached to them ("methods").
As a consequence, in OOP, *data* and the *methods* that operate on them are usually defined close together.
We can construct complex type hierarchies:

- define a class for *vehicle*, has a price method
- define a class for *bike*, inherits from *vehicle*
  - inherits the price method from *vehicle* (no need to rewrite it)
  - among other properties, has two wheels
- define a class for *car*, inherits from *vehicle*
  - inherits the price method from *vehicle* (no need to rewrite it)
  - among others has four wheels
- etc.

5.4.5 Functional programming

Functional programming treats functions as "first-class" citizens of the programming language.

- Functions can be used within expressions.
- Functions can be assigned to variables, passed as arguments, or returned from other functions.

A python example

```
def map(array, fn):
    r = array.copy()
    for i in range(len(array)):
        r[i] = fn(r[i])
    return r

def double_it(x):
    return x * 2

map([0, 1, 2, 3, 4], double_it)
# -> [0, 2, 4, 6, 8]
```

5.4.6 Declarative and logic programming

Declarative programming focuses on describing *what* the program should accomplish rather than specifying *how* to achieve it.

5.4.6.1 Logic Programming

- In logic programming, the programmer defines the desired results by specifying constraints and relationships, not the steps to solve the problem.
- An example is using SAT solvers to determine the satisfiability of logical formulas.

5.4.6.2 Example SAT formulas:

```
x1 and (not x2 or x3) and (not x3)
```

This SAT formula describes a logical constraint. The solver's role is to find a solution that satisfies this constraint, without the programmer detailing the steps to find such a solution.

6 Portability, Dependencies and Packaging

6.1 Appliaction Binary Interface (ABI)

- most Windows laptops, Linux laptops and pre-M1 Macs share the same ISA: `x86_64`
- iPhones, Android phones, M1 and M2 Macs share the same ISA: `AArch64`

Q: Why, then, do applications need to be recompiled separately for each platform?
e.g., iPhone vs. Android phone
A: Because platforms have different OSs and ABIs.

6.1.1 What is an ABI?

An application binary interfaces (ABI) defines:

- file format for
  - object files
  - dynamically-linked files (shared objects / dll)
  - and executable files
- convention for function calls
- convention for system calls

It is called binary because it is independent of the language in which applications are written (i.e., it is related to the machine code, not to the source code)

6.1.2 ABI: function calls (x86_64)

6.1.2.1 ABI: function calls (x86_64)

The Application Binary Interface (ABI) for function calls within the x86_64 architecture is exemplified by a simple C program that makes a call to the `puts` function. The assembly code generated by the `clang` compiler on Linux and the Microsoft Visual C++ (MSVC) compiler on Windows shows the differences in calling conventions and assembly syntax.
For **clang/Linux/x86_64**, the assembly instructions prepare the function argument by loading the address of the string into `rdi`, which is the register for the first argument as per the calling convention. It then calls `puts` and exits the program with a return value of 0.
For **MSVC/Windows/x86_64**, the assembly code shows the Windows calling convention where the string address is placed into the `rcx` register, and then the `puts` function is called. After the function call, the stack is cleaned up and the program returns.

6.1.2.2 ABI: function calls (AArch64)

In the case of the AArch64 architecture, the C program remains the same, but the assembly output changes significantly to reflect the different register and calling convention.
For **clang/MacOS/AArch64**, the assembly output uses the `stp` instruction to store pair of registers and the `adrp` and `add` instructions to form the address of the string. The first argument is moved into the `x0` register before the `puts` call is made.

For **MSVC/Windows/AArch64**, similar to the clang/MacOS assembly, we see the use of `stp` for storing register pairs and `adrp` plus `add` to handle the string address. The `puts` function is then called with the argument in `x0`, followed by the function's exit sequence.
`Try it for yourself: godbolt.org`

6.2 Portable Code

How do we ship code that work across all platforms?

6.2.1 Option 1: interpreters

- use interpreted languages, ship source
  - Python, Javascript, ...
- languages that compile to virtual machine code
  - ship VM code
  - optionally, ship VM interpreter
  - Java, C#

6.2.2 Option 2: multiple compilations

- compile one executable on each platform
- in some cases, cross-compilation is possible
  - MacOS → iOS
  - Linux → Android

6.2.2.1 What if we cannot (or do not want to) recompile?

6.2.3 Option 3: Translation

Use case: same OS, different ISA

- Translation is a form of compilation
- From machine code
- To machine code (of a different ISA)

Example: Apple Rosetta 2 translates `x86_64` into `AArch64`

6.2.4 Option 4: Compatibility layers

Use case: different OSs, same ISA

- add OS support for a foreign ABI
  - foreign file formats (for objects, DLLs and executables)
  - foreign convention for system calls

- add libraries for foreign ABI
  - foreign convention for function calls
- Examples:
  - Wine allows running Windows apps on Linux.
  - WSLv1 allows running Linux apps on Windows.

6.2.5 Option 5: emulation

- an emulator is an **interpreter** for machine code (e.g. QEmu)
- much slower than running the code
- JIT can mitigate slowness, to some extent
- typically, a full-blown **operating system** runs inside the interpreter!

6.2.6 Option 6: virtualization

- virtualization is essentially hardware-assisted emulation
  (e.g., Xen, KVM, VirtualBox, VMware, Apple Parallels, WSLv2)
- virtualized software must target the same ISA as hardware
- like emulation, runs a full-blown **operating system**

6.2.7 Definitions

- The **hypervisor** is the software that manages the **guest OS**.
- It can be the **host OS** itself ("Type 1": Xen, KVM)
- It can be a process within the **host OS** ("Type 2": Apple Parallels)

Virtualization is mainly deals with security:
Let **guest OSs** believe they have direct access to hardware...
... but every hardware access is tightly controlled by the **hypervisor**

6.2.7.1 Virtualization is the main technology enabling "cloud computing".

- Amazon Web Services runs **Xen**
- Google Cloud Platform runs **KVM**
- Customers rent a virtual machine in a datacenter
  - They can connect (remotely) to this machine
  - It runs their (**guest**) OS of choice
  - It acts as if it was physical hardware

6.2.8 Option 7: containers

Use case: Same ISA, same kernel, different OS.

- Containers are a lightweight form of virtualization.
- The host's kernel also acts as a kernel for the guest.
- Mainly: filesystems, libraries and applications are separated.

Examples:

- A Debian Linux guest on a Fedora Linux host
- A Debian 11 Linux guest on a Debian 12 host
- A Debian 12 guest with specific libraries installed, on a Debian 12 host

6.3 Application Programming Interfaces (API)

6.3.1 Definition

An API defines how a library (or any other service) is to be used.

6.3.2 Library API

```
FILE *fopen(const char *path, const char *mode);
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closedfd
```

6.3.3 Web API

```
GET https://www.google.com/search?q=<query>
```

6.3.3.1 Example:

```
google-chrome https://www.google.com/search?q=Software%20Engineering
GET https://cloudflare.com/cdn-cgi/trace
```

6.3.3.2 Example:

```
curl -s "https://cloudflare.com/cdn-cgi/trace" PUT https://api.cloudflare.com/client/v

curl --request PUT \
     --url https://api.cloudflare.com/client/v4/zones/zone_identifier/dns_records/ide
     --header 'Content-Type: application/json' \
     --header 'X-Auth-Email: ' \
     --data '{
       "content": "198.51.100.4",
       "name": "example.com",
       "proxied": false,
       "type": "A",
       "comment": "Domain verification record",
       "tags": [
         "owner: dns-team"
       ],
       "ttl": 3600
     }'
```

6.3.4 APIs and portability

- many APIs are cross-platform
  - C standard library
  - Almost all Python modules
  - Qt, Electron, Flutter, ... (frameworks for GUI applications)
  - WEB APIs only depend on an internet connection
- some are specific to a platform
  - Windows UI Library, MacOS Cocoa

6.4 Dependencies

- your code requires libA version ≥ 1.1, lib B version ≥ 4.5
  - ⋆ lib B version 4.5 requires libX version 2.0 and libA version 0.8
  - ⋆ lib B version 4.7 requires libX version 2.0 and libA version 1.1
  - ⋆ lib B version 4.6 requires libX version 2.0 and libA version 2.0
  - ⋆ lib X version 2.0 requires libA version ≤ 1.9

  How do we install all this?
  Which version do we install?

6.4.1 Package managers

Package managers solve this problem for you.
They can solve it...

- at the OS level:
  - ⋆ MacOS: `brew install <package>`
  - ⋆ Debian/Ubuntu Linux: `apt-get install <package>`
  - ⋆ Fedora/Suse Linux: `dnf install <package>`
- at the language level:
  - ⋆ Python: `pip install <module>`
  - ⋆ JavaScript/Node: `npm install <package>`

6.4.2 Limitations

- package selection may be limited (packaging is labor-intensive)
- security and trust

6.4.3 Tutorial: Treating Integers as Strings of Bits in python

- **OR Operation:** The bitwise OR (|) combines bits such that the bit in the result is 1 if either bit is 1.

  ```
  >>> x = 0b110000 | 0b000011
  >>> f"{x:06b}"  # '110011'
  ```

- **AND Operation:** The bitwise AND (&) combines bits with a result of 1 only if both bits are 1.

  ```
  >>> x = 0b111100 & 0b001111
  >>> f"{x:06b}"  # '001100'
  ```

- **XOR Operation:** The bitwise XOR (^) combines bits with a result of 1 only if the bits are different.

  ```
  >>> x = 0b101010 ^ 0b110011
  >>> f"{x:06b}"  # '011001'
  ```

- **Left Shift Operation:** Shifts bits to the left, filling in with zeros (<<).

  ```
  >>> x = 0b000110 << 2
  >>> f"{x:06b}"  # '011000'
  ```

- **Right Shift Operation:** Shifts bits to the right, discarding excess bits (>>).

  ```
  >>> x = 0b011000 >> 2
  >>> f"{x:06b}"  # '000110'
  ```

7 Python formatted strings, list comprehensions

7.1 Python formatted string literals

Formatted string literals look like string literals, but are prepended with an `f`:

```
f'Hello'
```

They allow to:

- build a string from python expressions
- specify how to format those expressions

Syntax: `f'raw_string {python_expression : format} ...'`

- `raw_string`: anything not between "{}" is a raw string literal
- between "{}" we specify a formatted expression
  - ⋆ `python_expression`: most python expressions are allowed here must not be ambiguous, e.g. no `".  "`
  - ⋆ `format`: specifies how to convert the corresponding expression to a string
- the f-string is *immediately* evaluated into a string:

```
>>> x = 3
>>> f'Hello {x}'
'Hello 3'
>>> type(f'Hello {x}')
<class 'str'>
```

Format:

- `<` for left align, `>` for right-align
- `+` include sign even for positive numbers
- `(space)` empty space for positive numbers
- `b` binary, `d` decimal (default), `x` hexadecimal
- `f` floating point number (fixed-point notation)
- `20` (or other number) specify the minimum width of the conversion result
- `.10` (or other number) specify the number of digits after the decimal dot

Examples:

```
>>> f'pi = {math.pi:+6.2f}'
'pi = +3.14'
>>> f'pi = {math.pi:<+6.2f}'
'pi = +3.14 '
```

7.1.0.1 Documentation:

- f-strings
- format specification

7.2 String methods

- `str.find(substr)`: Return the lowest index in the `str` where substring `substr` is found. Return -1 if `substr` is not found.
- `str.replace(old, new)`: Return a copy of `str` with all occurrences of substring `old` replaced by `new`.
- `str.strip(chars)`: Return a copy of the string with the leading and trailing characters removed. The `chars` argument is a string specifying the set of characters to be removed. If omitted or None, the `chars` argument defaults to removing whitespace.
- `str.split(sep=None, maxsplit=-1)`: Return a list of the words in the string, using `sep` as the delimiter string. If `sep` is None, runs of consecutive whitespace are regarded as a single separator (the result will contain no empty strings). If `maxsplit` is given, at most `maxsplit` splits are done.
- `str.join(iterable)`: Return the concatenation of the strings in `iterable`. The separator between elements is `str`.

7.3 Conditional Expressions

Syntax:

```
x if C else y
```

Example:

```
>>> x = 5
>>> 'big' if x > 10 else 'small'
'small'
```

7.3.0.1 Only the appropriate value is evaluated

```
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

7.3.0.2 Only the appropriate value is evaluated (example)

```
>>> import math
>>> x = 4
>>> 1 / x if x != 0 else math.inf
0.25

>>> x = 0
>>> 1 / x if x != 0 else math.inf
inf
```

> documentation

7.4 List comprehensions

7.4.1 List comprehension syntax:

The general syntax for a list comprehension in Python is:

`[ expression for variable in iterable ]`

- iterates over the values in `iterable`
- at each iteration, assign value to `variable`
- evaluate `expression` (may refer to `variable`)
- result becomes one list entry

7.4.1.1 Examples

```
>>> [i * 2 for i in range(8)]
[0, 2, 4, 6, 8, 10, 12, 14]

>>> [str(i) for i in range(8)]
['0', '1', '2', '3', '4', '5', '6', '7']

>>> [f'{i:02d}' for i in range(8)]
['00', '01', '02', '03', '04', '05', '06', '07']
>>> [f'{i:03b}' for i in range(8)]
['000', '001', '010', '011', '100', '101', '110', '111']
```

7.5 Dictionary Comprehensions

Dictionary comprehension syntax:
`{key_expr : val_expr for variable in iterable}`
Same as list comprehension, but for dict.
Example:
`>>> { i : f'{i:02b}' for i in range(4) }{0: '00', 1: '01', 2: '10', 3: '11'}`

7.6 Generator Expressions

Generator expression syntax:
(same as list comprehension, but with parentheses)

$$(\text{expression} \quad \textbf{for} \quad \text{variable} \quad \textbf{in} \quad \text{iterable})$$

Same as list comprehension, but creates a generator (iterable)
Example:

```
>>> a = (i * 2 for i in range(2 ** 28))
>>> a
<generator object <genexpr> at 0x7fa9195c5d80>
>>> sum(a)
72057593769492480
```

Note: there is no tuple comprehension

7.6.0.1 Documentation

- list comprehensions
- dict comprehensions
- generator expressions

8 Regular Expressions

8.1 Definition

Regular expressions are a mini-language for text pattern matching.

8.1.0.1 Example

Q: Find all occurrences of the word "memory" in the files in this directory.

```
grep 'memory' *
```

8.2 Matching

8.2.1 The grep command

```
grep [OPTION...] PATTERNS [FILE...]
```

8.2.1.1 Options:

- `-E`: "extended" regular expressions (we will use this syntax)
- `-R`: recursive (if a directory is given, look all files in it, incl. subdirectories)
- `-i`: case insensitive (a same as A)

8.2.1.2 Patterns:

Use single-quotes (`' '`) to avoid shell interference

8.2.1.3 Files:

if no file provided, grep reads its (piped) input

8.2.2 Piping to grep

Q: Find all files in the current directory whose name contains the letter L

```
ls | grep -E -i 'L'
```

8.2.3 Introduction to regular expressions

- by default, patterns are looked for line-by-line
- strings of "normal" characters are matched

```
grep -E 'memory' *
```

8.2.4 Anchors

- the ^ character at the beginning of a regex matches the beginning of a line
- the $ character at the end of a regex matches the end of a line

Examples:

```
grep -E '^int' *
grep -E 's$' *
```

8.2.5 Repetitions

- `?` indicates that the previous character may or may not occur (once)
- `*` indicates that the previous character may occur zero or multiple times
- `+` indicates that the previous character may occur one or more times
- {`4`} indicates that the previous character must occur 4 times
- {`4,`} indicates that the previous character must occur 4 or more times
- {`4,8`} indicates that the previous character must occur between 4 and 8 times

Examples:

```
grep -E 's?printf' *
grep -E '^ *printf' *
grep -E '0b+0' *
grep -E 'e{2,}' *
```

8.2.6 Grouping

Any part of a regex can be grouped using parentheses. Repetitions then apply to the group instead of a single character.
Examples:

```
grep -E '(Abc)+'    # matches 'Abc', 'AbcAbc', 'AbcAbcAbc', ...
```

8.2.7 Match any character

The dot (`.`) matches any character:
Examples:

```
grep -E 'X.Y'   # matches 'XaY', 'XbY', 'X+Y', ...
grep -E 'X.*y'  # matches 'XabcY', 'X-*-y', ...
```

8.2.8 Bracket expressions

- One character can be matched to multiple options using square brackets:

```
grep -E '[abc]XY'  # matches aXY or bXY or cXY
grep -E '[01]+'    # matches binary numbers
```

- We can express ranges of characters using a dash:

```
grep -E '[0123456789]+'     # matches decimal numbers
grep -E '[0-9]+'            # equivalent
grep -E '[0-9a-fA-F]+'      # matches hexadecimal numbers
grep -E '[A-Z][a-z]*'       # matches words that start with a capital letter
```

- Bracket expressions are negated if the first character is ˆ:

```
grep -E '[^A]sprintf'      # matches "printf", "sprintf" ... but not "sprint
```

8.2.9 Disjunctions

Multiple options can be given using the "—" character:

```
grep -E 'system_(startup|shutdown)'  # matches "system_startup" or "system_shutdown"
```

8.2.10 Special characters

Special characters can be "escaped" using a backslash ("\"):

```
grep -E 'printf\(.*\)'  # matches "printf("Hello %s", name)"
```

8.2.11 Using regular expressions in less

Searching for patterns in the `less` pager is performed by typing "/".
Patterns are specified using regular expressions

8.3 Search and Replace: `sed`

```
sed [OPTION...]  SCRIPT [FILE...]
```

- Options:
  - ⋆ `-E`: "extended" regular expressions (we will use this syntax)
  - ⋆ `-i`: edit file in-place (instead of printing)
- Script: Use single-quotes (` ' `) to avoid shell interference
- Files: if no file provided, `sed` reads its (piped) input

8.3.1 Basic search and replace

```
sed -E 's/REGEX/REPLACEMENT/'
```

- Examples:

```
sed -E 's/python/Python/'    # replace "python" with "Python"
sed -E 's/printf\(/\fprintf\(stderr, /' # replace "printf(" with "fprintf(stde
```

- Allow multiple replacements per line:

```
sed -E 's/REGEX/REPLACEMENT/g'   # g stands for global
```

- Use delimiter different from "/":

```
sed -E 's|REGEX|REPLACEMENT|'
sed -E 's_REGEX_REPLACEMENT_'
```

8.3.2 Advanced search and replace

- In the replacement string, \1 indicate the first parenthesized group, \2 the second, etc.:

```
# replace "Hello, World!" with "Bye, World!"
sed -E 's/Hello, ([A-Za-z]*\!)/Bye, \1!/'
```

- Groups are numbered in the order of the opening parentheses from the left:

```
sed -E 's/((a(b|z)+)(c+))/\1\3/g'
#         ^  ^    ^  ^
#         1  2    3  4
```

8.4 Regular Expressions in Programming Languages

8.4.1 Using regular expressions in C

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

int main()
{
    regex_t re;

    // REG_EXTENDED: POSIX extended regular expression
    // REG_NOSUB: do not report position of matches
    if (regcomp(&re, "0[xX][0-9a-fA-F]+", REG_EXTENDED | REG_NOSUB)) {
        fprintf(stderr, "Failed to compile regex\n");
        return 1;
    }

    int r = regexec(&re, "Does this contain a hex number, like 0xff ?", 0, NULL, 0);

    if (r == 0) {
        printf("Found\n");
    } else if (r == REG_NOMATCH) {
        printf("Not found\n");
    }

    regfree(&re);

    return 0;
}
```

See: `man regex`

8.4.2 Using regular expressions in Python

```
>>> import re
>>> m = re.search(r'0[xX][0-9a-fA-F]+', 'Does this contain a hex number, like 0xff ?'
>>> m.group(0)
'0xff'
```

> documentation

9 Version Control Systems

9.1 Basic Version Control Commands

```
Assume you have a project:
myproject.py

To try a modification, but unsure of its success:
mkdir versions/
mkdir versions/v1/
cp myproject.py versions/v1/

After modifications:
- If the modification is good:
  mkdir versions/v2/
  cp myproject.py versions/v2/

- To revert to the old version:
  cp versions/v1/myproject.py myproject.py
```

9.1.1 Use Cases for Version Control

- Experiment with changes (try things).
- Determine when and how bugs were introduced.
- Coordinate work among multiple collaborators on a project.

9.1.2 Version Control Systems (VCS) / Source Code Management (SCM)

- **Revision Control System (RCS), 1982**: Early system, operated on single files.
- **Concurrent Versions System (CVS), 1986**: Introduced the concept of repositories.
- **Apache Subversion ("SVN"), 2000**: Centralized VCS, allowed versioning of directories.
- **Mercurial ("Hg"), 2005** and **Git, 2005**:
  - ⋆ Distributed VCS, enabling workflows with local repositories.
  - ⋆ Git has spawned a large hosting industry (e.g., GitHub, GitLab).
  - ⋆ Both are used internally at major tech companies.
- **Piper**: Google's internal monorepo VCS, not publicly available.

9.1.3 Git fundamentals

A **repository** stores the complete history of a project.

- It is typically contained in the `.git/` directory at the root of the project.

9.1.3.1 Commit

A **commit** is a unit of change capturing:

- A snapshot of all the project files.
- Metadata such as author, date, and the **parent commit**.

Commits are uniquely identified by a **hash**.

9.1.4 Hashes

- A **hash** function maps any sequence of bits to a fixed-length bit string.
- The map is *surjective* but Git assumes it to be *bijective* for practical purposes.
- Git uses **SHA-1** for hashing: 160 bits / 20 bytes / 40 hexadecimal digits.
    - ⋆ Example hash: `1e6cac37c5c8c5ee99ec104954d09b07e96116ba`
- Git is migrating to **SHA-256**: 256 bits / 32 bytes / 64 hexadecimal digits.

9.1.5 Hashes in Git

- Commits are designated by SHA-1 hashes.
- A short hash prefix can be used to refer to commits if it is unambiguous.

9.2 Building a Commit

9.2.1 Working Tree

- The **working tree** consists of files that are currently being worked on.
- We never interact with the `.git/` directory contents directly.
- The working tree reflects the current state of files, excluding the `.git/` directory.
- We can use the `git checkout` command to update the working tree to match a specific commit.

9.2.2 Staging Area

- Before committing, we **stage** changes, specifying which modifications should be included in the next commit.
- The staging area is a snapshot of what our next commit will look like.

9.2.3 Commit

- After staging, we **commit** the changes to the repository's history.
- A commit includes a snapshot of the staged changes and a commit message describing the changes.

9.2.3.1 Staging and Committing Example

```
# Creating or modifying files
new_file_A.py
new_file_B.py
new_file_C.py

# Staging files
git add new_file_A.py new_file_B.py

# Committing to the repository
git commit -m "My first commit."
```

Listing Past Commits

```
git log
```

```
commit 6ea8433f89c78588190435c877b1d3e7c708 (HEAD -> main)
Author: Laurent Perriraz <lperriraz@dev>
Date:   Fri Sep 29 02:44:18 2023 +0200

    My first commit.
```

9.2.4 Automatic Adding

Add multiple files at once using a pattern:

```
git add *.py
```

Add all files in the working tree:

```
git add -A
```

Exclude files from automatic adding using `.gitignore`.

9.2.5 Observing the State

```
git status
```

9.2.6 Showing Differences

```
git diff
```

9.2.7 Staging Changes

```
git add -A
git status
git diff
git diff --staged
```

9.2.8 Committing Changes

```
git commit -m "My second commit."
git log
```

9.2.9 Checking Out Commits

```
git checkout <commit-hash>
git log
git log --all
```

9.2.10 Working with Commits

- Modify files in the working tree and stage the changes.
- Commit the staged changes with a descriptive message.
- Use `git log` to see the history of commits.
- Check out a specific commit to view or revert to that snapshot.

9.3 Branches

9.3.1 Commit Structure

Commits in Git form a linked list structure, with each commit pointing back to its parent.

9.3.2 Checking Out Commits

You can switch between commits using:

```
git checkout <commit-hash>
```

9.3.3 Problem with Detached HEAD

If you check out a commit directly, you enter a 'detached HEAD' state where changes are not on any branch.

9.3.4 Creating Branches

To avoid a detached HEAD, create a new branch:

```
git branch <branch-name>
```

Branches allow to work on features or bug fixes while keeping the main branch stable.

9.3.5 Switching Branches

To switch to a different branch:

```
git checkout <branch-name>
```

9.3.6 Visualizing Branches

You can visualize the commit history and branches using:

```
git log --all --graph
```

9.3.6.1 Branching Example

- Initial state after two commits:

  ```
      commit 31a051 (HEAD -> main)
      commit 6ea843
  ```

- Create a new branch and switch to it:

  ```
      git branch my_branch
      git checkout my_branch
  ```

- The branch points to the current commit.

9.3.7 Merging

Merging incorporates changes from one branch into another.

```
git merge <branch-name>
```

9.3.8 Merge Conflicts

Conflicts occur when the same lines are changed in both branches. They need to be resolved manually.

9.3.9 Rebasing

Rebasing is an alternative to merging, replaying commits from one branch onto another.

```
git rebase <base-branch-name>
```

9.3.10 Handling Merge Conflicts

Resolve conflicts by editing files, then stage and commit the resolution.

```
# Edit files to resolve conflicts
git add <file>
git commit
```

9.3.10.1 Important Points

- Always commit changes before switching branches.
- Use merging or rebasing as per the team's workflow conventions.
- Resolve conflicts carefully to maintain code integrity.

9.4 Remotes

9.4.1 Sharing Commits

Git is distributed and allows multiple remotes. There's no central server.

9.4.1.1 Fetching Commits

To download commits from a remote repository:

```
git fetch <URL>
```

The URL must be accessible, either public or with appropriate credentials.

9.4.1.2 Cloning Repositories

If the repository was initially cloned:

```
git clone <URL>
```

Use 'git fetch' without a URL to fetch from the original source.

9.4.1.3 Format Patch

To save commits as files for email, use:

```
git format-patch
```

9.4.2 Best Practices

- Regularly push to and pull from remotes to keep local and remote repositories synchronized.
- Use 'git fetch' to update local references with remote repository.
- Use 'git pull' to fetch and merge changes from the remote server to your working directory.
- Use 'git push' to update the remote repository with your local changes.

10 Software Licenses

10.1 Closed-source software

Most end-user software is closed-source (proprietary).

- The executable is distributed to customers.
- The source code is either
  - ⋆ never revealed (most commonly), or
  - ⋆ only made available to select customers (rarely).

10.1.0.1 Example of closed-source software

- Operating systems: Microsoft Windows; Android, iOS, MacOS (except kernel)
- Office suites: Microsoft 365, iWork
- Creative software: Adobe suite, Autodesk suite, Final Cut Pro, Pro Tools, Logic Pro
- Development software: Visual Studio, XCode (except compiler)
- Collaboration software: Zoom, Teams, Skype, Slack, Discord
- Server-side and enterprise software: Microsoft IIS, SAP, OpenAI GPT-4
- Almost all videogames
- Almost all mobile apps

10.2 Free software

- "free" as in freedom (not "free lunch")
- defined by the Free Software Foundation (FSF, est. 1985)
- software attached with a *license* (uses copyright law)
- gives *freedoms* (rights) to the *user*, to:
  - ⋆ run the software as they wish
  - ⋆ study and modify the software as they wish
  - ⋆ redistribute (original and modified versions)
- based on the philosophy that all *software should be free* to protect users
- the "GNU" project is FSF's software collection

10.3 Open-source software

- defined by the Open Source Initiative (OSI, est. 1998)
- software attached with a *license* (uses copyright law)
- specifies how software can be *distributed*:
  - ⋆ no restrictions on redistribution
    - ∗ no discrimination against specific users, fields, products, other software, other technologies
  - ⋆ source code must be available
  - ⋆ derived works must be allowed
  - ⋆ but modifications can be required to be clearly delineated

10.3.1 FSF vs. OSI

- **FSF:** for the user's sake, all software *should be free* on ethical grounds – free software licenses are a means to that end
- **OSI:** help businesses and developers publish and disseminate their open-source software – pragmatically, we do not want to add hurdles if they impair practical use

10.3.1.1 In practice?

The FSF and OSI each maintain a list of "approved" license.
Most FSF-approved *free software* licenses are also OSI-approved *open-source* licenses. And vice-versa.
The difference lies in the licenses each organization *promotes.*

10.3.1.2 The "baseline" FSF license

The GNU General Public License (GPL):

- any user who receives the executable must be provided the source code as well upon request
- any derivative work is automatically covered by the GPL (the GPL is "viral")
- dynamic linking with GPL software counts as derivative work

10.3.1.3 Amended FSF licenses

- The "more permissive" **GNU Lesser General Public License (LGPL)**:
  - ⋆ adds exception to allow dynamic linking with non-GPL software
- The "more restrictive" **GNU Affero General Public License (AGPL)**
  - ⋆ definition of "user" includes over-the-network interactions

10.3.2 Typical open-source licenses

- Most popular: Apache License, BSD License, MIT License
- "permissive licenses": fewer constraints on derivative work
  - ⋆ unmodified parts still covered by the original license
  - ⋆ but modified parts are not, can even be closed source
- some require acknowledgement of the original work (authors and/or project)
- differences among permissive licenses are minor (but important to lawyers)

10.3.2.1 Example projects

- "free software" (GPL-type licenses)
  - ⋆ Linux kernel, **GPL**
  - ⋆ GNU project, **GPL**
  - ⋆ gcc, **GPL**
  - ⋆ glibc (gcc's standard C library), **LGPL**
  - ⋆ git, **GPL**
  - ⋆ gmp, **LGPL**
- "open source" (permissive licenses)
  - ⋆ Apache web server, **Apache**
  - ⋆ NGINX web server, **BSD**
  - ⋆ LLVM, **Apache**
  - ⋆ Chrome (more precisely: chromium), **BSD**
  - ⋆ Node.js, Angular, React, **MIT**

10.4 End-user cost

Whether or not customers pay for software is orthogonal to source availability.

Cost	Closed-source	Free / Open-source
0	TikTok, Whatsapp, Discord	Chrome, Gimp, VLC, Blender
> 0	Photoshop, Maya, Ableton	Red Hat Enterprise Linux

10.5 Commercial, non-commercial

Whether or not developers are commercial entities is orthogonal to source availability.

developers	closed-source
non-commercial	(most amateur code until 2010s, some government software, legacy scientific soft
commercial	Microsoft Windows, Microsoft 365, iWork, Adobe suite, Autodesk, …

*the distinction between commercial and non-commercial is often blurry

10.5.0.1 How can commercial software be free / open-source?

- software has zero price, sell support and services (Ubuntu, Red Hat, NGINX)
- software costs money, convince customers not to redistribute it (Red Hat)
- open-core: basic functionality is open-source, sell advanced features (NGINX)
- open-sourced software accesses proprietary services (Chrome)
- open-sourced software is not core business (LLVM)

10.6 Use cases

- Closed-source:
  - ⋆ source code is your "secret sauce"
  - ⋆ customers willing to pay
- Open-source permissive licenses:
  - ⋆ encourage wider adoption
  - ⋆ encourage commercial entities to participate
- Free software GPL-type licenses:
  - ⋆ protect users (ethical grounds)
  - ⋆ force downstream developers to reciprocate
- Share source code, but do not give any right to modify (limited usefulness)

10.7 Patents

In most countries:

- Contrary to copyright law (protects creative processes) patents are not a fundamental right
- Patents are a pragmatic compromise for promoting innovation.

  The bargain is:
  - ⋆ Share your innovation with the patent office (as opposed to keeping it secret)
  - ⋆ Get N-year exclusivity on commercialization

10.7.1 Patents and software innovation

- Software innovation is quicker: N years is like centuries
- Ideas are cheap, execution is everything
- Software is close to mathematics (discovered, not invented)
- Patent disclosures do not include code! They don't actually help anyone.

10.7.2 Stuff that has been patented

- Buy with a single click (Amazon)
- Automatically make email addresses clickable (Apple)
- Fourier (1768–1830) series for compression (Fraunhofer Institute)

11 Specifications

11.1 A note about C

11.1.1 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 and 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

11.1.2 Why the C ABI?

The C ABI is simple:

- just functions and simple types: integer, pointer, `struct`
- no objects or methods
- no exceptions

11.1.3 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

11.2 Specifications

11.2.1 What is even the C language?

11.2.2 Questions

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

11.2.3 What is valid C?

- Pragmatically, C code is valid if your compiler produces a valid executable.
- However, there are many compilers.
- It would be convenient if they agreed on a definition for the C language.

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 committee 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"
- 2023: ISO working on update "C23", provisionally "C2x"

Hence `-std=c2x`

11.2.4 Who defines the C language nowadays?

- A "working group" within ISO: "WG14"
  - ⋆ Compiler writers
  - ⋆ Hardware vendor representatives
  - ⋆ OS maintainers
  - ⋆ Academics
- > C23 draft (742 pages)

11.3 Behaviours

11.3.1 Locale-Specific Behavior

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

11.3.1.1 Example

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

```
int a = islower('æ');
```

11.3.2 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

11.3.2.1 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
};
```

11.3.3 Implementation-defined Behavior

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

11.3.3.1 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

11.4 Undefined Behavior

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

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

Example

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

11.4.1 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."

- A simple C program that attempts to divide by zero can cause a runtime error:

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

  When `argc` is 1, the program attempts to divide by zero, resulting in a floating-point exception.

- Compiling with warnings enabled can alert the developer to the potential issue:

  ```
  clang -O3 -std=c2x -o main main.c
  main.c:3:11: warning: division by zero is undefined
  ```

- The actual result of running the compiled program may vary, producing different outputs or crashing:

  ```
  ./main
  -882586408
  1687000168
  -1071941800
  -60110776
  ```

11.4.2 Easy UB: division overflow

Integer division in C follows specific rules, but certain edge cases, like division overflow, can result in undefined behavior.

Truncation Towards Zero

- The result of the / operator is the algebraic quotient with any fractional part discarded, a process known as truncation toward zero.
- For example, both `5 / -2` and `-5 / 2` would result in `-2`.

Representability and Equality

- If the quotient `a/b` is representable in the data type of the result, then the equation `(a/b)*b + a%b` should equal `a`.
- This property is used to confirm that division and modulus operations are consistent with mathematical definitions.

Undefined Behavior in Division

- When `a/b` produces a quotient that is not representable, such as an integer division by zero or division overflow, the behavior is undefined.
- Undefined behavior means the output and side effects of the code are unpredictable and not reliable.

11.4.2.1 Example

Consider the following C code that demonstrates undefined behavior due to division overflow:

```
#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;
}
```

11.4.3 Explanation

- The largest positive value an `int` can represent is `INT_MAX`, and the smallest negative value is `INT_MIN`.
- Dividing `INT_MIN` by `-1` is undefined because the positive result cannot be represented as an `int` (it would be one more than `INT_MAX`).
- Compilers may produce a warning, and the program may exhibit unexpected behavior when executed.

11.4.4 Integer overflow

Overflow occurs when an arithmetic operation exceeds the maximum size that can be represented within a given number of bits.

11.4.5 Unsigned Integer Overflow

- Unlike signed integer overflow, unsigned integer overflow is well-defined in C.
- It exhibits wrap-around behavior, meaning that if the result of an operation is too large to be represented in the given unsigned type, it wraps around from the largest representable number back to zero.

11.4.5.1 Example of Unsigned Overflow

The following C code demonstrates unsigned integer overflow:

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

int main() {
    uint8_t a;
    for (int i = 0; i < 1000; i++) {
        printf("%012" PRIu8 "\n", a);
        a = a + 1;
    }
    return 0;
}
```

Explanation

- The `uint8_t` type is an 8-bit unsigned integer, capable of representing values from 0 to $2^8 - 1$ (255).
- When the loop increments `a` beyond 255, it wraps around to 0 due to unsigned overflow.
- This behavior is predictable and consistent with the C standard.

Wrap-around Behavior

- If `i` and `j` are n-bit unsigned integers, the result of `i + j` is $(i + j) \mod 2^n$.
- This property ensures that any operation on unsigned n-bit integers produces the bottom n bits of the true arithmetic value.

Hardware Implementation

- Modern architectures like x86_64 and ARM64 adhere to this behavior in their instruction sets, handling unsigned integer overflow by wrapping around.

11.4.6 Signed integer overflow

While hardware architectures like x86_64 and AArch64 have instructions that exhibit wrap-around behavior on overflow, the C language standard specifies that signed integer overflow is undefined behavior.

11.4.6.1 Example

Consider this C code snippet:

```
#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;
}
```

Explanation

- The code attempts to increment an integer `a` that is already at `INT_MAX`, the largest value a signed integer can hold.
- Theoretically, adding 1 to `INT_MAX` would wrap around to `INT_MIN` due to overflow, but in C, this behavior is not guaranteed and should not be relied upon.
- The actual output of the program can vary depending on how the compiler chooses to handle the overflow.

11.4.7 Invalid Pointer Dereferencing

When a pointer in C is assigned an invalid value, dereferencing it results in undefined behavior.

Standards Specification

- The C standard states: "If an invalid value has been assigned to the pointer, the behavior of the unary `*` operator is undefined."
- This can lead to various issues, including segmentation faults, data corruption, or other erratic behavior.

Example and Consequences
Here's a C code snippet that can cause undefined behavior due to invalid pointer dereferencing:

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

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

Analysis

- The function `int_at` attempts to dereference a pointer that has been cast from an integer with value 1, which is not a valid memory address for an `int`.
- Running this code typically results in a segmentation fault, as the address does not point to a legitimate memory location.

Handling Invalid Pointers

- To avoid undefined behavior, pointers should always be validated before dereferencing.
- For instance, checking if a pointer is `NULL` before using it can prevent some forms of invalid pointer dereferencing.

11.4.7.1 Example of Pointer Validation

A safer version of the function would include a check for `NULL`:

```
int int_at(int *pointer) {
    if (pointer == NULL) {
        return 0; // Or another error-handling strategy
    }
    int r = *pointer;
    return r;
}
```

12 Lecture 12

12.1 Undefined Behavior

12.1.1 Recap

The C standard use a few key words that have precise definitions.
Examples:

- `isspace()` : "The `isspace` function tests for any character that is a standard white-space character or is one of a locale-specific set of characters [. . .]" (p206)
- `qsort()` : "[. . .] If two elements compare as equal, their order in the resulting sorted array is unspecified." (p369)
- **Byte**: "A byte is composed of a contiguous sequence of bits, the number of which is implementation-defined." (p4)
- "If an object is referred to outside of its lifetime, the behavior is *undefined*." (p36)
- Locale-specific behavior: Behavior that depends on local conventions [...] that each implementation documents. (e.g., `isspace()`)
- Unspecified behavior: Behavior for which there are multiple possibilities. (e.g., `qsort()`)
  - ⋆ Implementation-defined behavior: Unspecified behavior where each implementation (compiler / platform / OS) documents which choice is made. (e.g., `byte`)
- Undefined behavior

12.1.1.1 Undefined behavior

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

*C23 standard

Possible consequences:

- compilation or execution crashes
- situation completely ignored with unpredictable results,
- implementation-defined behavior
- by chance, nothing happens and everything goes as intended by the programmer (bad!)
- anything else

12.1.2 We have already seen

All of the following trigger undefined behavior:

- division by zero
- division overflow
- signed integer overflow
- dereferencing invalid pointers

```
int main()
{
    int i = INT_MAX + 1;
    int b = (i == 100);
    printf("b = %d\n", b);
    return 0;
}
```

The compiler is allowed to produce code with output:

```
b = 0
```

```
b = 1
```

```
b = 42
```

Deleting all your files now...
If an expression is UB, it does not just get a "wrong" value: it invalidates the whole program.

12.1.3 Not an idle threat

The provided C code sample demonstrates a situation where undefined behavior can occur due to compiler optimizations. A static function pointer is initialized to NULL and is intended to point to a function that would erase all files.

- The function `this_function_is_never_called` is defined which sets the function pointer to the `erase_all_files` function, but it is never called.
- The `main` function returns the result of the function pointer, which is dereferenced and executed.

When compiled with `gcc` using the `-O3` optimization flag, the program results in a segmentation fault, as the function pointer is NULL. However, when compiled with `clang` using the same optimization level, the program executes the function that the pointer is supposed to point to, resulting in the message "Deleting all your files NOW..." and potentially dangerous behavior.

12.1.3.1 Integer Overflow

On x86_64 and AArch64, "add" has wrap-around semantics:
add w0, INT_MAX, 1 ⇒ w0 = INT_MIN

- will yield i = INT_MIN sometimes
- still undefined behavior
- will create bugs in the future!

In another example, an integer overflow case is presented. The function `f` adds 1 to the maximum value an integer can hold, which due to wrap-around semantics in x86_64 and AArch64 architectures, results in the minimum integer value:

- In a situation where the variable `i` is set to `INT_MAX` and then passed to `f(i)`, it could sometimes yield `i = INT_MIN`.
- This behavior is considered undefined, meaning that the compiler is not required to handle this situation in any particular way.
- Such scenarios can lead to bugs that are difficult to trace and rectify in the future.

Following the C standard, the compiled code is (only) bound to behave as if it was running on the "C abstract machine".
No additional constraints are placed on the compiler when targeting a particular ISA even if that ISA's specification has no undefined behavior

12.1.4 (Almost) everything wrong is undefined behavior (1)

"The behavior is undefined in the following circumstances: [...] An unmatched ' or " character is encountered on a logical source line during tokenization" (p584)

```
#include <stdio.h>

int main()
{
    printf("Hello
}

% Error message generated by the compiler
test.c:5:16: error: missing terminating " character
    5 |     printf("Hello
      |            ^
```

All modern compilers turn this (and all other parsing errors) into implementation-defined behavior specifically: interrupted compilation with error message

12.1.5 (Almost) everything wrong is undefined behavior (2)

```
#include <stdio.h>

char *f()
{
    char buffer[16];
    snprintf(buffer, sizeof(buffer), "Hello");
    return buffer;
}

int main()
{
    char *s = f();
    printf("Here is the return value of f():\n");
    printf("%s\n", s);
    return 0;
}

% Compiler warnings and errors
gcc -O3 -o bug bug.c
bug.c: In function ‘f’:
bug.c:9:16: warning: function returns address of local variable [-Wreturn-local-addr]
    9 |     return buffer;
      |            ^

% Runtime output causing a segmentation fault
./bug
Here is the return value of f():
Segmentation fault (core dumped)
```

12.1.6 Undefined behavior can time-travel

"[...] However, if any such execution contains an undefined operation, this document places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation)." (C++20, p7)

```
int f(int a, int b)
{
    printf("a = %d, b = %d\n", a, b);
    printf("We could get a crash now:\n");
    return a / b;
}
```

The compiler is allowed to produce an executable that does this:

```
a = 10, b = 0
DELETING ALL FILES NOW, HA HA HA HA !!!!!
We could get a crash now:
Floating point exception (core dumped)
```

12.1.7 Undefined behavior can time-travel (really)

The example C code sample demonstrates two critical concepts: handling division by zero and an illustration of what is colloquially known as "time travel" in code execution due to compiler optimizations.

- The function `f(int a, int b)` performs an integer division of `a` by `b`. If `b` is zero, the division would result in a runtime error: a floating point exception.
- In the `main` function, the result of the division is conditionally assigned based on the `argc` parameter. If `argc` is less than 2, the result is set to 5; otherwise, it's set to the result of `strtol` conversion of the second argument. This result is then passed as the divisor to the function `f`.
- When the code is compiled with `gcc` using optimization flag `-O3` and executed with `argc` less than 2, the "time travel" behavior occurs. The compiler optimizes the code in a way that the division by zero exception happens even before the conditional check in `main` can prevent it.
- The assembly output below the code shows the division instruction `idiv` which is the source of the exception. The program crashes with a floating point exception, demonstrating the consequences of undefined behavior in C.

12.1.8 But why?!??

- Performance!
- It is all about letting the compiler make **assumptions**
  - ⋆ Specifically, the compiler assumes that undefined behavior never happens

12.2 Pointer aliasing rules

"Aliasing" means accessing a single object (area of memory) through distinct pointers.
The C standard specifies *"strict aliasing"*:
An object can only be accessed (both read or written) through pointers to that type of object.
⇒ If two pointers have different types, they **must** point to distinct objects.
"An object shall have its stored value accessed only by an lvalue expression that has one of the following types:

- a type compatible with the effective type of the object,
- a qualified version of a type compatible with the effective type of the object,
- a type that is the signed or unsigned type corresponding to the effective type of the object,
- a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,

- an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or
- a character type." (p71)

12.2.1 a type compatible with the effective type of the object

Valid:

```
typedef int my_int;

my_int f(int *pointer)
{
    my_int *my_pointer = pointer;
    return *my_pointer;
}
```

Undefined behavior:

```
int f(long *pointer)
{
    int *my_pointer = (int *)pointer;
    return *my_pointer;
}
```

12.2.1.1 Explanation:

In the context of pointer aliasing, the term *"effective type"* refers to the actual data type that a memory location is being treated as at any given point in the program. Strict aliasing rules in C dictate that you can only safely access a memory location through a pointer that matches the effective type of the data stored at that location.
In the valid example, ‘my_int‘ is a type alias for ‘int‘, and thus, ‘my_int *‘ and ‘int *‘ are considered compatible types. This means that the pointer ‘my_pointer‘ is of a type compatible with the effective type of the object ‘pointer‘ points to (which is ‘int‘). As a result, accessing the value pointed by ‘my_pointer‘ is well-defined and adheres to the strict aliasing rules.
On the other hand, the undefined behavior example demonstrates a violation of these rules. Here, a ‘long *‘ is being forcefully cast to an ‘int *‘. ‘int‘ and ‘long‘ are distinct types and may have different sizes or representations in memory. According to strict aliasing rules, an object of type ‘long‘ must not be accessed through a pointer of type ‘int *‘. This breach can lead the compiler to make optimizations based on the assumption of strict aliasing, potentially resulting in unpredictable behavior at runtime.

12.2.2 a qualified version of a type compatible with the effective type of the object

Valid:

```
int f(int *pointer)
{
    const int *my_pointer = (const int *)pointer;
    return *my_pointer;
}
```

12.2.2.1 Explanation:

In this valid example, the pointer ‘pointer‘ is of type ‘int *‘, and it is cast to ‘const int *‘ for ‘my_pointer‘. This is permissible under the strict aliasing rules as it involves adding a qualifier to the type.
The type ‘const int‘ is considered a qualified version of ‘int‘. The act of adding the ‘const‘ qualifier doesn't change the effective type of the object; it simply adds a constraint that prevents modification of the object through the ‘const‘-qualified pointer. In C, it's allowed to access an object through a pointer to a qualified version of the object's effective type. This ensures that the access remains well-defined, and the object's memory representation is correctly interpreted.
However, it's important to note that while you can safely read the value pointed to by ‘my_pointer‘, attempting to modify it (if ‘my_pointer‘ was not originally ‘const‘) would violate the immutability imposed by the ‘const‘ qualifier and lead to undefined behavior.

12.2.3 a type that is the signed or unsigned type corresponding to the effective type of the object

Valid:

```
unsigned int f(int *pointer)
{
    unsigned int *my_pointer = (unsigned int *)pointer;
    return *my_pointer;
}
```

12.2.3.1 Explanation:

In this example, a pointer of type ‘int *‘ is cast to ‘unsigned int *‘ and then dereferenced. While ‘int‘ and ‘unsigned int‘ are both integer types, they represent numbers in different ways: ‘int‘ for signed integers and ‘unsigned int‘ for unsigned integers.
According to the strict aliasing rules, it is generally not safe to access an object through a pointer of a different type. However, ‘int‘ and ‘unsigned int‘ are an exception to this rule. The C standard allows objects to be accessed by pointers of their corresponding signed or unsigned types because the memory representation is the same for both signed and unsigned variants of the same size. This means that the bit pattern in memory is interpreted differently but not modified when accessing an ‘int‘ as an ‘unsigned int‘ or vice versa.
Nonetheless, this practice should be approached with caution. The reinterpretation of the bit pattern means that the numerical value accessed via ‘my_pointer‘ may differ significantly from the value pointed to by ‘pointer‘. For instance, a negative ‘int‘ value, when accessed as an ‘unsigned int‘, will be interpreted as a large positive number. Such behavior is well-defined but can lead to unexpected results if not carefully managed.

12.2.4 a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object

Valid:

```
unsigned int f(int *pointer)
{
    const unsigned int *my_pointer = (const unsigned int *)pointer;
    return *my_pointer;
}
```

12.2.4.1 Explanation:

In this example, a pointer of type ‘int *‘ is cast to ‘const unsigned int *‘ and then dereferenced. As previously discussed, the C standard allows the object to be accessed through a pointer to the corresponding signed or unsigned type, acknowledging that ‘int‘ and ‘unsigned int‘ have compatible memory representations.
Adding the ‘const‘ qualifier to ‘unsigned int *‘ to form ‘const unsigned int *‘ is also permissible and aligns with the strict aliasing rules. As noted in the earlier example, adding a qualifier like ‘const‘ introduces a promise not to modify the object through this pointer, ensuring read-only access. This action does not change the underlying effective type of the object in memory but merely adds a constraint on how the pointer can be used.
Combining these two aspects, this example is valid because it involves accessing an object through a pointer to a qualified version of a type (with ‘const‘) that corresponds to the signed or unsigned type of the effective type of the object (‘int‘ being accessed as ‘unsigned int‘). However, the same caution mentioned previously applies here: the numerical value accessed via ‘my_pointer‘ might be interpreted differently than the value pointed to by ‘pointer‘, due to the difference between signed and unsigned representation.

12.2.5 an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a sub-aggregate or contained union)

Valid:

```
struct vec3d {
    int x, y, z;
};

void vec3d_copy(struct vec3d *dst, struct vec3d *src)
{
    *dst = *src;
}
```

12.2.5.1 Explanation:

This example demonstrates the valid use of an aggregate type, specifically a ‘struct‘, in the context of strict aliasing rules. In C, an aggregate type is a data type that groups multiple individual variables, possibly of different types, into a single unit. In this case, ‘struct vec3d‘ is an aggregate type composed of three ‘int‘ members: ‘x‘, ‘y‘, and ‘z‘.

The strict aliasing rules permit an object to be accessed through a pointer to an aggregate or union type that includes among its members the type compatible with the effective type of the object. Here, both ‘src‘ and ‘dst‘ are pointers to ‘struct vec3d‘, and the operation ‘*dst = *src‘ involves copying the contents of one ‘struct vec3d‘ object to another. This operation is well-defined because it respects the memory layout and type of the aggregate ‘struct vec3d‘.
It’s worth noting that aggregate types are treated as a single unit. This means that accessing or modifying a ‘struct vec3d‘ object as a whole (as done in this example) is different from accessing or modifying its individual members. This example adheres to the rules by treating ‘src‘ and ‘dst‘ as pointers to the whole aggregate type, ensuring that the strict aliasing rules are followed.

12.2.6 a character type

Valid:

```
struct vec3d {
    int x, y, z;
};

void copy(char *dst, char *src, size_t n)
{
    for (size_t i = 0; i < n; i++) {
        dst[i] = src[i];
    }
}

int main()
{
    struct vec3d a = {1, 2, 3};
    struct vec3d b;

    copy((char *)&b, (char *)&a, sizeof(a));

    return 0;
}
```

12.2.6.1 Explanation:

This example illustrates an exception in the strict aliasing rules that allows any object to be accessed through a pointer to a character type. In this case, ‘char *‘ is used to access and copy the bytes of a ‘struct vec3d‘ object. This is permissible because character types (‘char‘, ‘signed char‘, ‘unsigned char‘) are specifically allowed to alias any other type.
The function ‘copy‘ takes pointers to ‘char‘ and copies ‘n‘ bytes from the location pointed to by ‘src‘ to the location pointed to by ‘dst‘. In the ‘main‘ function, the addresses of ‘struct vec3d‘ objects ‘a‘ and ‘b‘ are cast to ‘char *‘, and the ‘copy‘ function is used to replicate the memory contents of ‘a‘ into ‘b‘. This is a common technique used for generic memory copying in C, often seen in functions like ‘memcpy‘.

This is valid under the C standard because it is an established exception to the strict aliasing rules, acknowledging that character pointers can be used for low-level memory manipulation, allowing them to access the byte representation of any object.

12.2.7 Strict aliasing violations

Whenever we cast a pointer type to another pointer type, it is very likely that we invoke undefined behavior.
Danger! Probable undefined behavior ahead:

```
int *a;
short *b = a;
```

12.2.8 Strict aliasing violations (1)

```
uint32_t build_u32(uint16_t a, uint16_t b)
{
    uint32_t r;

    uint16_t *p = &r;

    p[0] = a;
    p[1] = b;

    return r;
}
```

12.2.8.1 Explanation:

This code violates strict aliasing rules by accessing a ‘uint32_t‘ object through a pointer of type ‘uint16_t *‘. The C standard requires that an object's memory be accessed only through its own type or a compatible type. Here, ‘uint32_t‘ and ‘uint16_t‘ are distinct types with different sizes, and accessing the ‘uint32_t‘ object ‘r‘ as if it were an array of ‘uint16_t‘ leads to undefined behavior according to the strict aliasing rules.

12.2.9 Strict aliasing violations (2)

This code demonstrates a strict aliasing violation involving different structure types. Although ‘struct my$_{d}ata_{0}$‘, $‘structmy_{d}ata_{1}‘, and‘structmy_{d}ata_{2}‘allhaveacommoninitialmember‘subtype‘,$
$data‘pointerasifitcouldbe‘structmy_{d}ata_{1}*‘or‘structmy_{d}ata_{2}*‘basedonthe‘subtype‘.Thisviolatesstrictal$

12.2.10 Strict aliasing violations (3)

The code snippet involves a ‘union‘ named ‘mux‘ that allows its memory to be accessed as either an array of ‘int32$_{t}$‘(‘i‘)$or‘int16_{t}‘(‘s‘).Whileunionsaredesignedtoprovideawaytotreatthesamemen$
$specificbehavior, whichisgenerallydiscouraged$
Note: Some compilers promise to yield the intended operations here.

12.2.11 How do I do type-punning then?

"Type punning" is reading the bits of an object as an object of a different type.
Valid:

```
int main()
{
    int i[2];
    short s[4];

    i[0] = 0x03020100;
    i[1] = 0x07060504;

    memcpy(s, i, 2 * sizeof(int));

    printf("%d %d %d %d\n", m.s[0], m.s[1], m.s[2], m.s[3]);

    return 0;
}
```

12.2.11.1 Explanation:

The code demonstrates valid type-punning using 'memcpy' to safely copy data from an 'int' array to a 'short' array. This method avoids strict aliasing violations by copying the memory representation without directly accessing the data through pointers of differing types. This approach ensures that the underlying bytes are transferred accurately without invoking undefined behavior, making the operation safe and portable across different compilers and platforms. 'memcpy' respects the memory boundaries and types, providing a reliable method for type-punning in compliant C code.

12.2.12 Why is strict aliasing good for code optimization?

Strict aliasing rules significantly impact code optimization by providing the compiler with guarantees about memory access patterns. These guarantees enable compilers to perform more aggressive optimizations, resulting in faster and more efficient code. Here's how:

- **Assumption of Independence:** When strict aliasing rules are followed, the compiler assumes that pointers of different types point to non-overlapping memory regions. This assumption allows the compiler to optimize code without having to consider the possibility of one pointer aliasing (or pointing to the same memory location as) another pointer of a different type. As a result, the compiler can generate more streamlined and efficient machine code, as it is free from the burden of inserting additional checks or conservative code paths that account for potential aliasing.

- **Loop Optimizations:** The examples labeled "Fast" demonstrate how adherence to strict aliasing rules enables the compiler to optimize loops more effectively. The compiler can safely assume that the elements being accessed in the loop do not overlap in memory, allowing it to minimize memory loads and stores, and sometimes even vectorize the loop (use SIMD instructions). This results in fewer instructions and a more efficient utilization of the CPU's execution units.

- **Effective Instruction Scheduling:** By understanding the non-overlapping nature of memory accesses, the compiler can schedule instructions in a way that reduces pipeline stalls and improves the overall throughput of the program. This is particularly beneficial in modern processors with deep pipelines and multiple execution units.

12.2.13 And when strict aliasing is not enough?

When strict aliasing is not enough to guarantee non-overlapping memory regions, particularly in pointer-intensive operations, the `restrict` keyword can be used as an additional hint to the compiler. This keyword is a promise to the compiler that for the lifetime of the pointer, only the pointer itself or a derivative of it will be used to access the object to which it points. This assurance allows the compiler to make further optimizations by assuming that pointers declared with `restrict` are not aliased, meaning they do not point to overlapping memory regions.

- **Enhanced Loop Optimizations:** The use of `restrict` enables the compiler to assume that the arrays `dst`, `src`, and `constant` do not overlap. This allows for more aggressive optimizations in loops, as seen in the "Fast" example. The compiler can optimize memory access patterns and instruction scheduling, resulting in more efficient code execution.
- **Improved Parallelism:** Knowing that the data does not overlap allows the compiler to avoid conservative assumptions about data dependencies. This can lead to improved parallel execution, both at the instruction level (such as pipelining and instruction-level parallelism) and at higher levels (such as loop unrolling and vectorization).
- **Better Resource Utilization:** With clearer assumptions about memory usage, the compiler can make more informed decisions about register allocation, instruction selection, and other optimizations that contribute to better utilization of the CPU and memory hierarchy.

12.3 More types of undefined behavior

12.3.1 Unaligned pointers

Every type has a required alignment (which we can query with `alignof(type)`). (see p44)
Every pointer to that type must be a multiple of that alignment.
Undefined behavior:

```
int *alloc_5_bytes()
{
    char *c = malloc(1 + sizeof(int));
    return c + 1;
}
```

12.3.2 Out-of-bounds pointer arithmetic

"When two pointers are (added or) subtracted, both shall point to elements of the same array object, or one past the last element of the array object;" (p84)
Undefined behavior:

```
size_t eight()
{
    char c[4];
    return &(c[8]) - &(c[0]);
}
```

12.3.3 Infinite loops

An infinite loop with no side effects is undefined behavior.
Undefined behavior:

```
while (1)
{
}
```

Valid:

```
while (1)
{
    printf("Hello\n");
}
```

12.3.4 Shift beyond integer size

Undefined behavior:

- left/right shift integer by a negative number

  ```
  uint32_t a = 1 >> -5;
  ```

- left/right shift an n-bit integer by n or more positions

  ```
  uint32_t a = 1;
  uint32_t b = a << 32;
  ```

- left shift signed integer i by k positions and $i \times 2^k$ is not representable

  ```
  uint32_t a = -1024;
  uint32_t b = a << 30;
  ```

C23 pp584–594: 218 types of undefined behavior

13 Floating-point arithmetic

13.1 Real Numbers

13.1.1 How do we represent non-integers?

Keeping in mind:

- If we consider n bits of memory,
  - ⋆ their values can take 2^n combinations
  - ⋆ so we can represent 2^n numbers at best with those n bits
- We have a finite amount of memory,
  - ⋆ so we cannot represent all real numbers
- We (typically) want fast operations,
  - ⋆ so (ideally) we need hardware to perform them.
  - ⋆ Hardware has tight limits on the number of logic gates available
  - ⋆ meaning we use very few bits (say 16, 32 or 64, like for integers)
  - ⋆ . . . further restricting how many real numbers we can represent

13.1.2 Practical limitations

- Integer are restricted in one way:
  - ⋆ their range (e.g. [`INT_MIN`, `INT_MAX`])
- Reals are restricted in two ways:
  - ⋆ their range (e.g. $[-10^{308}, 10^{308}]$)
  - ⋆ the number of reals we can represent in that range (e.g. $\{..., 0, 10^{-200}, 2 \times 10^{-200}, ...\}$)
    i.e. their precision

13.2 Fixed-point arithmetic

13.2.1 Decimal example

Instead of computing money values in €, we could use c:
e.g. 29.99€ = 2999c
then use integer operations.

- This is fixed-point arithmetic
- specifically, with 2 decimal places reserved for the fractional part.

If $+, -, \times, /$ are the elementary integer operations:

- $euro\_to\_cent(e) ::= e \times 100$
  - ⋆ $euro\_to\_cent(5) = 500$
- $cent\_to\_euro(a) ::= a/100$
  - ⋆ $cent\_to\_euro(700) = 7$
- $cent\_add(a, b) ::= a + b$
  - ⋆ $cent\_add(700, 500) = 1200$
- $cent\_sub(a, b) ::= a - b$
  - ⋆ $cent\_sub(700, 500) = 200$
- $cent\_mul(a, b) ::= (a \times b)/100$
  - ⋆ $5 \times 7 : cent\_mul(500, 700) = 500 \times 700/100 = 3500$
- $cent\_div(a, b) ::= (a \times 100)/b$
  - ⋆ $8/4 : cent\_div(800, 400) = (800 \times 100)/400 = 200$

13.2.2 Binary fixed-point arithmetic

- There is no universally accepted standard for fixed-point arithmetic
- But there is no real need for one:
  - ⋆ Only two parameters:
    n**:** total number of bits
    p**:** number of bits after the decimal point
  - ⋆ All the operations are just integer operations
    - ∗ For *mul* and *div*, two integer operations each

13.2.2.1 Binary example

- A 64-bit integer can be used for fixed-point arithmetic by dividing it into two parts: a 32-bit integer part and a 32-bit fractional part.
- The conversion and arithmetic operations are defined as follows:

```
i64_to_fix(e) := e × 2^32
fix_to_i64(a) := a / 2^32
fix_add(a, b) := a + b
fix_sub(a, b) := a - b
fix_mul(a, b) := (a × b) / 2^32
fix_div(a, b) := (a × 2^32) / b
```

The function $i64_to_fix()$ is taking an integer e of type $i64$ (which is a 64-bit integer) and converting it into a fixed-point representation of type *fix*. The multiplication by 2^32 effectively shifts the integer value e into the higher 32 bits of the 64-bit representation, leaving the lower 32 bits for the fractional part. This way, it is possible to perform fixed-point arithmetic with fix types while maintaining precision for the fractional component of the values.

These operations are implemented using integer arithmetic, providing precise control over the scaling and range of the values.

13.2.2.2 Implementation in C

Fixed-point arithmetic operations can be implemented in C using a typedef for a 64-bit integer to represent fixed-point numbers. The integer part occupies the upper 32 bits and the fractional part occupies the lower 32 bits.

- **Conversion to Fixed-Point:** An integer is converted to fixed-point format by shifting it left by 32 bits, moving the integer part to the upper 32 bits.
- **Conversion from Fixed-Point:** A fixed-point number is converted back to an integer by shifting it right by 32 bits, discarding the fractional part.
- **Addition and Subtraction:** These operations are performed directly on the fixed-point representations, with the result being in fixed-point format.
- **Multiplication:** Multiplication requires casting to a wider integer (128-bit) to accommodate the intermediate result before shifting right by 32 bits to maintain the correct scale.
- **Division:** Division is handled by first shifting the dividend left by 32 bits (to align the integer part correctly) before performing the division.

Each operation maintains the scale of fixed-point representation and ensures that the integer and fractional parts are processed correctly. Care is taken during multiplication and division to avoid overflow and to maintain precision.

13.2.2.3 Fixed-point arithmetic, Pros

- fast, no need for extra hardware
- easy to understand and study (predictible):
    - ⋆ uniform absolute precision (e.g. 2^{-32} over whole range)

13.2.2.4 Fixed-point arithmetic, Cons

- limited range (e.g. $[-2147483648.999, 2147483647.999]$)
- limited precision (e.g. $2^{-32} \approx 0.000000002328$)

Possible improvements:

- larger range
- better absolute precision around zero
- lower absolute precision for big numbers

13.3 Floating-point Arithmetic

13.3.1 Scientific notation

Take the number -2147483648.999:

$$\begin{aligned}-2147483648.999 &= -2.147483648999 \times 10^9 \\ &= -2.147483648999e + 9\end{aligned}$$

Similarly, take the number 0.0000000002328:

$$\begin{aligned}0.0000000002328 &= 2.328 \times 10^{-10} \\ &= 2.328e - 10\end{aligned}$$

13.3.2 Scientific Notation

Scientific notation is a way to express numbers that are too large or too small to be conveniently written in decimal form. It is commonly used in calculations and by scientists, mathematicians, and engineers.

- In scientific notation, numbers are written as a product of two numbers: a coefficient and 10 raised to a power, for example, $-2.147483648999 \times 10^9$.
- The coefficient must be a number between 1 and 9, followed by a decimal point and the rest of the significant digits.
- The exponent is written as a power of 10, indicating how many places the decimal point should be moved to convert the number into a standard decimal number.

13.3.3 Binary Floating-Point Numbers

In computing, binary floating-point numbers are a way to represent real numbers in a binary system, typically using three components: the sign, exponent, and mantissa.

- The sign determines if the number is positive or negative.
- The mantissa, or significand, is composed of the significant digits of the number.
- The exponent indicates where the decimal (or binary) point is placed relative to the beginning of the mantissa.

Binary floating-point numbers allow for a wide range of values to be represented in a standardized way, which is why they are commonly used in computer systems.

13.3.4 Pros and Cons of Binary Floating-Point Numbers

Pros:

- Efficient to process with modern hardware.
- Predictable precision and representation across different systems.

Cons:

- Limited precision, which can lead to rounding errors in calculations.
- Limited range, meaning extremely large or small numbers may not be representable.

13.3.5 Floating-point standard

- In 1985, the Institute of Electrical and Electronics Engineers publishes standard #754 about floating-point arithmetic (IEEE-754)
- The standard #754 defines the number of bits used for the sign, exponent, and mantissa. This standard ensures that floating-point operations behave consistently across different computing systems.
- Most hardware makers adopt the standard very quickly thereafter (Intel 80387, launched in 1987, is fully compliant)
- x86_64 natively supports binary32 and binary64 formats
- AArch64 natively supports binary16, binary32 and binary64 formats

component	binary16	binary32	binary64
$\pm$ sign bit	1	1	1
mmmmm... mantissa bits	10	23	52
xxxx... exponent bits	5	8	11
exponent range	-14..15	-126...127	-1022...1023

13.3.6 Precision

Absolute and Relative Precision

- **Absolute precision:** For a given x, the smallest ε such that

$$fl(x+\varepsilon) \neq fl(x)$$

- **Relative precision:** For a given x,

$$\varepsilon := \frac{\varepsilon}{x}$$

13.3.7 Binary64 vs. Fixed-Point 32+32 Comparison

Precision at Different Scales

- The fixed-point 32+32 format provides a consistent absolute precision across different magnitudes, beneficial for values around 1 or below.
- Floating-point binary64 format, based on IEEE-754, offers varying precision, with relative precision being consistent but absolute precision varying with the value's magnitude.

Range

- Fixed-point 32+32 has a limited range up to $|x| \leq 2.15 \times 10^9$, making it unsuitable for very large values.
- Binary64 supports a significantly larger range up to $|x| \leq 1.80 \times 10^{308}$, accommodating both very large and very small values.

	fixed-point 32+32		floating-point binary64	
	absolute	relative	absolute	relative
precision at 10^{-9}	2.33×10^{-10}	0.0233	2.07×10^{-25}	2.22×10^{-16}
precision at 10^{-6}	2.33×10^{-10}	2.33×10^{-5}	2.12×10^{-22}	2.22×10^{-16}
precision at 10^{-3}	2.33×10^{-10}	2.33×10^{-8}	2.17×10^{-19}	2.22×10^{-16}
precision at 1	2.33×10^{-10}	2.33×10^{-11}	2.22×10^{-16}	2.22×10^{-16}
precision at 10^{3}	2.33×10^{-10}	2.33×10^{-14}	1.14×10^{-13}	2.22×10^{-16}
precision at 10^{6}	2.33×10^{-10}	2.33×10^{-17}	1.16×10^{-10}	2.22×10^{-16}
precision at 10^{9}	2.33×10^{-10}	2.33×10^{-20}	1.19×10^{-7}	2.22×10^{-16}
precision at 10^{16}	X		2.00	2.22×10^{-16}
range	$\|x\| \leq 2.15 \times 10^9$		$\|x\| \leq 1.80 \times 10^{308}$	

13.3.8 The Floating-Point Number Line

- The floating-point number line is non-linear, with denser representation of numbers near zero and sparser as the magnitude increases.
- This reflects the constant relative precision of floating-point numbers.

13.3.9 Programming Languages and IEEE-754 Standard

- Several programming languages mandate compliance with the IEEE-754 standard for floating-point arithmetic, ensuring consistency across computing platforms.
- Languages like C (since C99) and C++ (since C++03) offer types like `float` for binary32 and `double` for binary64.
- Modern languages such as Python and JavaScript also adhere to this standard, with Python using binary64 precision by default.

language	**since**	**binary32**	**binary64**
C	C99	float	double
C++	C++03	float	double
Fortran	Fortran 2003	real	double
Rust		f32	f64
Python			✓
JavaScript			✓

13.3.10 Inaccuracy in Base 10 and Base 2

In floating-point arithmetic, inaccuracy can arise due to the way numbers are represented in base 10 and base 2. For example, fractions like $\frac{1}{3}$ and $\frac{2}{3}$ do not have exact representations in base 10 and result in repeating decimals. This inaccuracy is also present in binary (base 2), where numbers like 0.1 cannot be precisely represented.

13.3.11 Numerical Instability

Numerical instability refers to errors that can grow when applying mathematical operations, especially in iterative calculations or when dealing with very large or small numbers.

Derivative Approximation
A common operation in numerical analysis is the approximation of derivatives using finite differences:

$$\frac{df(x)}{dx} \approx \frac{f(x+\delta) - f(x)}{\delta}$$

However, choosing a very small δ can lead to significant inaccuracies due to the limited precision of floating-point numbers.

13.3.12 Example of Numerical Instability

Consider computing the derivative of $f(x) = x$ at large values of x using a small δ. When x is much larger than δ, the subtraction $(f(x+\delta) - f(x))$ can result in loss of precision, and the computed derivative will deviate from the expected value of 1.

13.3.13 What is happening?

- At $x = 10^{+5}$, we first compute $(10^{+5} + 10^{-6})$
  - ⋆ which is a big number, close to 10^{+5}
  - ⋆ floating-point numbers have a good *relative* accuracy everywhere, $\sim 2.22 \times 10^{-16}$
  - ⋆ but at 10^{+5}, the *absolute* accuracy is not great, $\sim 1.91 \times 10^{-6}$
  - ⋆ so the result of $(10^{+5} + 10^{-6})$ may be off by roughly 1.91×10^{-6}
- We then subtract 10^{+5}.
  - ⋆ If we were using exact arithmetic, we would get 10^{-6} exactly,
  - ⋆ but we are using floating-point arithmetic,
  - ⋆ so we get something close to 10^{-6}...

    - ⋆ ... but potentially off by roughly 1.91×10^{-6}
- We divide by 10^{-6},
    - ⋆ and get a number in $[1 - 1.91, 1 + 1.91]$

The deviations we just discussed arise because floating-point numbers have more relative precision near zero, and less absolute precision as their magnitude increases.
Therefore,

- floating-point accuracy is often great
- but some algorithms are **unstable**
- we need to be extremely careful before trusting floating-point results

13.3.14 Never do exact comparisons

```
>>> 0.1 + 0.2 == 0.3
False

>>> 1.0 - 1e-16 == 1.0
True
```

13.3.15 So how do we do comparisons?

- If exact comparisons are important, **do not use floating-point arithmetic**.
- If we care about speed and can tolerate some errors...

```
>>> 0.1 + 0.2 == 0.3
False
```

becomes

```
>>> tolerance = 1e-10
>>> abs( (0.1 + 0.2) - 0.3 ) < tolerance
True

>>> x = 0.0
```

becomes

```
>>> x > -tolerance
```

13.4 Rounding Floating-Point Numbers

When a real number cannot be represented exactly by a floating-point number, we approximate it by ”rounding” to the nearest floating-point number. The IEEE-754 standard outlines several rounding modes to handle such cases:

- **Round to Nearest, Ties to Even:** The default rounding mode which rounds to the nearest value; if equidistant, it rounds to the nearest even number in the floating-point representation.
- **Round to Nearest, Ties Away from Zero:** Rounds to the nearest value; if equidistant, it rounds away from zero.
- **Round Toward Zero:** Always rounds towards zero, truncating the extra digits.
- **Round Toward $+\infty$:** Always rounds up.
- **Round Toward $-\infty$:** Always rounds down.

13.4.1 Determinism in Floating-Point Arithmetic

While floating-point arithmetic may be inaccurate due to rounding, it is deterministic:

- The result of most operations is precisely defined by the standard.
- We can predict the result of operations bit-for-bit, ensuring reproducible calculations.

13.4.2 IEEE-754 Rounding Rules

Let us denote by $fl(x)$ the floating-point representation of the real number $x \in \mathbb{R}$.
The IEEE-754 standard mandates correct rounding as per the selected rounding mode for various operations:

- **Addition, Subtraction:** $x + y$ results in $fl(x + y)$.
- **Multiplication, Division:** x/y results in $fl(x/y)$.
- **Square Root:** $\sqrt{x}$ results in $fl(\sqrt{x})$.
- **Fused Multiply-Add:** $fma(x, y, z)$ results in $fl(x \times y + z)$.

13.4.3 Division example

When executing

$$z = x/y$$

- we first take the floating-point numbers x and y, and consider them as if the were (exact, infinite-precision) real numbers
- we compute the (exact, infinite-precision) real quotient x/y.
- we round the result according to the current **rounding mode**: $\mathrm{fl}(x/y)$
- we store the **rounded** floating-point value into z

13.4.4 Expression example

Consider the expression:

$$\frac{y \times (x + 4.0)}{z - 3.0}$$

In a floating-point computation, due to rounding errors, this expression gives:

$$fl\,(fl(y \times fl(x + 4)) \div fl(z - 3))$$

where fl denotes the floating-point representation of a number, highlighting the sequential rounding that occurs at each step.

13.4.5 About fused multiply-add

The fused multiply-add operation is a common computation in many numerical algorithms. It combines multiplication and addition but must be treated with caution:
Beware: The fused multiply-add function $fma(x, y, z)$ does not simply equate to $x\times y+z$.
Indeed:

- $fma(x, y, z)$ computes $fl(x \times y + z)$, providing a more accurate result by combining the operations into a single step to minimize rounding errors.
- However, separately computing $x \times y + z$ results in $fl(fl(x \times y) + z)$, where each operation introduces potential rounding errors.

13.4.6 More floating-point non-identities

Certain algebraic identities do not hold for floating-point numbers due to rounding errors and finite precision:

- Associativity does not hold: $x + (y + z) \neq (x + y) + z$.
- Distributivity does not hold: $x \times (y + z) \neq x \times y + x \times z$.

The IEEE-754 standard defines the representation and behavior of floating-point numbers:

- It mandates correct rounding for basic operations such as $+$, $-$, $\times$, $\div$, $\sqrt{}$, and $fma()$.
- However, it does not mandate correct rounding for more complex functions like trigonometric, hyperbolic, exponential, and logarithmic functions. This can lead to discrepancies in different implementations and requires careful handling.

13.4.7 Floating-point and compilers

- C99 and C++03 mandate IEEE-754
- which in turn mandates correct rounding for $+, -, \times, \div, \sqrt{()}$, fma().
- However, if we do not specify a C or C++ standard (e.g., `-std=c17` or `-std=c++20`), gcc and clang do not follow IEEE-754
  - ⋆ they will happily exploit associativity and distributivity
  - ⋆ they will replace $x \times y + z$ by fma(x, y, z)

13.4.8 Why does correct rounding matter?

- (generally) not because of accuracy
- but because for any real number x, there is exactly one correct rounding
- as a result, there is no ambiguity:
  - ⋆ given a set of floating-point numbers
  - ⋆ given any expression involving those numbers and $+, -, \times, \div, \sqrt{()}, \text{fma}()$
  - ⋆ there is exactly one correct answer
  - ⋆ which is precisely specified by IEEE-754, down to its bit representation

13.4.9 What happens without correct rounding?

Results can change when:

- we change architecture
- we change compiler
- we change the standard C library
- we change the version of the compiler
- we change the version of the standard C library
- we change our code (even a completely unrelated part)

Note: If we use sin, cos, log, exp, ..., which are not correctly rounded, then we are exposed to result *changes* whenever we change the version of the standard C library (which could be dynamically linked!)

13.5 Beyond floating-point Arithmetic

13.5.1 Interval arithmetic

Interval arithmetic is an approach to encompass the uncertainty and rounding errors in numerical computations.

- Every real number $x \in \mathbb{R}$ is represented by an interval $[l, u]$ using a pair of floating-point numbers, ensuring that x lies within this interval.
- This method employs rounding modes, specifically *Round toward* $+\infty$ and *Round toward* $-\infty$, to compute the accurate interval for each operation.

Advantages:

- It is a fast method.
- It provides knowledge of how accurate a result is.

Disadvantages:

- The interval $[l, u]$ can become excessively large quickly, often yielding overly pessimistic bounds.

13.5.2 Unum

Unum arithmetic is a relatively new system, introduced to improve the precision and reliability of computations:

- It was introduced in 2015, with the latest revision in 2017.
- Unum claims to allocate precision better within a given fixed bit width and offers optional interval arithmetic.
- However, it has seen very limited adoption, mainly because there is no hardware support on mainstream platforms as of the last update.

13.5.3 The GNU Multi-Precision Library (GMP)

The GNU Multi-Precision Library is an important tool for computations requiring high precision:

- GMP is a C library designed to provide support for variable-width integers and arbitrary-size rational numbers.
- A rational number is expressed as a fraction $\frac{\text{numerator}}{\text{denominator}}$, where the greatest common divisor (gcd) of the numerator and denominator is 1, ensuring the fraction is in its simplest form.
- More information about GMP can be found at `http://gmplib.org`.

13.5.4 The GNU MPFR library

The GNU MPFR library enhances the GNU Multi-Precision Library (GMP) by providing a facility for arbitrary-precision floating-point computation.

```
double x = 22.0 / 7.0;
printf("%.20f\n", x);

mpfr_t x;
mpfr_init2(x, 512);             // initialize x with 512-bit mantissa
mpfr_set_ui(x, 22, MPFR_RNDN);  // set x to value 22, round-to-nearest
mpfr_div_ui(x, x, 7, MPFR_RNDN); // divide x to 7, round-to-nearest
mpfr_printf("%.20Rf\n", x);     // print x
mpfr_clear(x);                  // free memory
```

For more details, visit `http://mpfr.org`.

13.5.5 Python fractions

Python supports arbitrary-precision integers and provides a module for exact rational number arithmetic.

```
>>> -2 ** 65
-36893488147419103232   # correct result, no overflow

import fractions
a = fractions.Fraction(numerator, denominator)
```

13.5.6 Why don't we always use exact rational numbers?

While exact rational numbers provide precision, their use is not always practical in computing due to several factors:
Convenience:

- Using GMP in C or importing fractions in Python may not always be straightforward.

Memory:

- The size of the numerator and denominator can become very large, especially in iterative algorithms, even after gcd reductions.

Speed:

- Operations on arbitrary-sized integers or exact rationals are slower (often by an order of magnitude) compared to fixed-size native types, due to lack of native hardware support.

13.5.7 The Case for Exact Rational Numbers

Should we use exact rational numbers more often in computations? Yes, particularly when:

- Exactness is crucial to the application.
- Computational speed is not a limiting factor.

Exact rational numbers provide unparalleled precision and are essential in applications where accuracy cannot be compromised, such as in cryptography or symbolic mathematics.

13.5.8 Symbolic computations

Symbolic computation systems are powerful tools that allow for manipulation and solving of algebraic expressions symbolically rather than numerically. This means that the expressions are kept in their algebraic form, allowing for exact solutions and manipulations.

In a symbolic algebra system:

- The square root of 2, denoted as $\sqrt{2}$, is never approximated to a decimal such as 1.4142. It is kept in the root form for exact calculations. For instance, in SageMath:

  ```
  sage: sqrt(8)
  2*sqrt(2)
  ```

- Variables can be defined without assigning them specific values, which is useful for carrying out algebraic manipulations. For example:

```
sage: x, y, z = var('x y z')
sage: sqrt(8) * x
2*sqrt(2)*x
```

- Symbolic computations allow solving algebraic problems, such as equations, symbolically. For instance:

```
sage: x, b, c = var('x b c')
sage: solve([x^2 + b*x + c == 0], x)
[x == -1/2*b - 1/2*sqrt(b^2 - 4*c), x == -1/2*b + 1/2*sqrt(b^2 - 4*c)]
```

13.5.9 Symbolic Algebra Systems

Several systems are available for performing symbolic computations:

- SageMath is an open-source mathematical software system with a syntax similar to Python.
- Maple is a commercial computer algebra system.
- Wolfram Mathematica is a widely used system that offers powerful symbolic computation capabilities.

WolframAlpha, a web application based on Mathematica, provides step-by-step solutions and is accessible at WolframAlpha.

14 Software engineering practices

14.1 Tools for Program Correctness

Today:

1. Documentation
2. Testing
3. Static analysis
4. Dynamic analysis

- Each uncovers bugs
- For each, there are useful tools (compilers can help!)

14.2 Documentation

Documentation is **Good**.

- Allows others to understand your code
- Allows yourself (in a few weeks) to understand your own code
- Helps make your thought process and assumptions explicit

14.2.1 Types of documentation

- Reference manuals
- Tutorials
- Questions and answers (Q&A)

14.2.2 Reference manuals

- Authoritative source of information
  If the code does not do what the manual says, then the code is wrong.
- Must be complete
- Must use precise language
  Even at the cost of legibility
- Examples: “man” pages, C standard, IEEE-754 specifications

14.2.3 Tutorials

- Beginner-friendly
- Usually emphasize getting things to work quickly
  even at the cost of completeness
- Good tutorials do not sacrifice accuracy (but many bad ones do)
- Examples: various books (K&R C, Think Python) and intro material

14.2.4 Questions and answers (Q&A)

- Prioritize quick answers to frequently asked questions
- Not exhaustive
- Examples: Stack Overflow, various FAQs

14.2.4.1 When reading documentation:

- as a beginner, aim for **tutorials** and Q&As
- as you become an expert, you need a **reference manual**.

14.2.4.2 When writing documentation:

- ideally, you **write all three**!

14.2.5 Automated documentation

Automated documentation systems

- read and parse source code
- find functions (methods, classes, ...)
- create a (PDF or webpage) document containing function signatures
- specially-formatted comments in the source code are copied into the documentation along with the corresponding function signatures

14.2.6 Automated documentation systems

- General:
  - ⋆ doxygen
  - ⋆ sphinx
- Python-specific:
  - ⋆ pdoc
  - ⋆ PyDoc
  - ⋆ pydoctor

Note: Some projects choose to not use automated documentation.

14.3 Testing

```
/*
 * This functions returns:
 * 5 if one or both of its arguments are 5
 * 0 otherwise
 */
int five_if_some_five(int a, int b)
{
    if (a == 5)
        a = 0;
    if (b == 5)
        b = 0;
    return a | b;
}

int tests()
{
    int errors = 0;
    errors += (five_if_some_five(100, 100) != 0);
    errors += (five_if_some_five(100,   5) != 5);
    return errors;
}
```

14.3.1 Test coverage

- line coverage:
  is every line of code covered by some test case?
- branch coverage:
  for every conditional branch, is there a test covering each of the two possibilities (taking the branch or not taking it?)

After running the provided tests using coverage analysis tools, the results are as follows:

- The `clang` command with flags `-Wall -O3 --coverage` was used to compile the source file `five.c` into the object file `five.o`, and to enable coverage analysis.
- Executing the compiled test binary `./test` resulted in zero errors, indicating that all the tests passed successfully.
- Coverage analysis was performed using `gcov`. The report generated by `gcov five.c` showed that 100% of the lines were executed, suggesting full line coverage.
- When running `gcov -b five.c`, which includes branch coverage information, the report indicated that all branches were executed 100% of the time, demonstrating full branch coverage.
- However, further detailed analysis using `gcov` revealed that the branches within the `five_if_some_five` function were not equally covered. Specifically, the condition `if (a == 5)` was never true (0% taken), indicating that the test cases did not cover the scenario where `a` is equal to 5.

- On the other hand, the branch `if (b == 5)` was taken 50% of the time, corresponding to the test case where `b` is equal to 5.

14.3.2 Line coverage vs. branch coverage

```
int test()
{
    int errors = 0;

    errors += (five_if_some_five(100, 100) != 0);

    return errors;
}
```

Line coverage:	100%
Branch coverage:	50%

14.3.3 How does it work?

Line coverage measures the percentage of code lines that have been executed during a test run. Line coverage does not ensure that all edge cases or logical branches are tested; it only indicates which lines of code were run.

14.3.4 Limitations of test coverage measures (1)

```
[language=C, basicstyle=\footnotesize\ttfamily, breaklines=true, showstringspaces=fal
/*
 This functions returns:
 5 if one or both of its arguments are 5
 0 otherwise
 */
int WRONG_five_if_some_five(int a, int b)
{
    return a | b;
}

int test()
{
    return (WRONG_five_if_some_five(0, 5) == 5);
}
```

Line coverage: 100% Branch coverage: 100%

14.3.4.1 Explanation:

Despite achieving 100% line and branch coverage, the test case does not effectively validate the function's correctness. The function `WRONG_five_if_some_five` uses the bitwise OR operation, which does not implement the intended logic correctly. While the test case covers the scenario where one argument is 5 (yielding the expected result due to the nature of bitwise OR with 0 and 5), it fails to catch incorrect results for other inputs, like both arguments being 5 or neither being 5.

14.3.5 Limitations of test coverage measures (2)

The example illustrates that even with 100% line and branch coverage, test coverage metrics can fail to ensure the correctness of the function. The function is intended to return 5 if any of the arguments is 5, and 0 otherwise. The test cases do not cover all possible paths, specifically the case where neither argument is 5. Consequently, it does not detect the flaw in the function logic, where the function incorrectly modifies the arguments and potentially returns the wrong result.

14.3.6 Assertions

- Assertions are used to document (and check) assumptions made in the code.
- An assertion failure
  - ⋆ should correspond to a bug in your code,
  - ⋆ triggers an immediate crash (`abort()`) of your program.

```
#include <assert.h>

int gcd(int a, int b)
{
    if (a < b) {
        int t = a;
        a = b;
        b = t;
    }

    while (b != 0) {
        assert(a >= b); // <--- this should always be true
        int t = a % b;
        a = b;
        b = t;
    }

    return a;
}
```

14.3.7 Disabling assertions

```
clang -DNDEBUG -Wall -O3 -o main main.c
```

(equivalent to `#define NDEBUG` at the beginning of every file)

14.3.8 Error vs assertion failure

- an error happens when, for external reasons, your program cannot run
  - ⋆ examples: out of memory, file cannot be read, network unreachable
- an assertion fails if a fundamental assumption in your code is violated
  - ⋆ indicates a bug in your code

14.4 Static Analysis

- **Static analysis** operates on the source code
    - ⋆ (before any assembly or executable code is produced)
- Compilers do advanced case analysis on the code
    - ⋆ (in order to produce faster code)
- The same analysis can be used to find (potential) bugs
- Not an exact science
    - ⋆ Relies on heuristics to detect hazardous code
    - ⋆ Suffers from false negatives and false positives

14.4.1 Clang's static analyzer

If you use a Makefile, run

```
scan-build make
```

>¿ result

14.4.2 Python linters

- A "linter" is a static analyzer
- Typically, linters enforce a specific coding style

Examples:

- Pylint
- flake8
- mypy (adds static type checking)

```
def fib(n):
    a, b = 0, 1
    while a < n:
        yield a
        a, b = b, a+b

def fib(n: int) -> Iterator[int]:
    a, b = 0, 1
    while a < n:
        yield a
        a, b = b, a+b
```

14.5 Dynamic Analysis

- Dynamic analysis operates on the running executable (during testing)
  - ⋆ by adding runtime checks
  - ⋆ can find more bugs than static analysis...
  - ⋆ ... but only for those bugs are triggered by some test!

14.5.1 Sanitizers

With *sanitizers*, runtime checks are added by the *compiler*.

14.5.1.1 UBSan

- The "undefined behavior sanitizer" detects many types of undefined behavior (at runtime)
- triggers an immediate crash (with an explanation message)
- Pass `"-fsanitize=undefined"` to `gcc` or `clang`

14.5.1.2 Demonstration of UBSan in Action

The code sample illustrates the usage of the Undefined Behavior Sanitizer (UBSan) by comparing the output of a program compiled without and with UBSan.

- The function `f(int a, int b)` attempts to divide `a` by `b`. When compiled without UBSan and executed with `b` as zero, it results in a floating point exception. This is a runtime error that occurs due to division by zero, an undefined behavior in C.
- When compiled with UBSan using `clang -O3 -fsanitize=undefined -o timetravel timetravel.c`, the program terminates with a runtime error message indicating the cause of the error: division by zero. UBSan provides a stack trace pointing to the exact location in the code where the undefined behavior occurred.
- The output with UBSan includes details about the file name, line number, and even the assembly instruction pointer location, which are invaluable for debugging.
- UBSan causes the program to abort execution instead of continuing past the undefined behavior, which helps prevent further errors or unpredictable behavior in the program's execution.

14.5.1.3 Pros

- Fixes the anything-can-happen problem with undefined behavior (we get a crash with an explanation instead)
- No false positives

14.5.1.4 Cons

- Not all types of undefined behavior detected (most are)
- Does not always stop the compiler from exploiting undefined behavior
- Overhead (~3x slowdown)
- Needs good tests

14.5.2 ASan

- The “address sanitizer” detects many types memory access errors (at runtime)
- Separate from UBSan because it uses different mechanisms
- triggers an immediate crash (with an explanation message)
- Pass “-fsanitize=address” to gcc or clang

14.5.2.1 Example

In the function `f`, a local buffer is declared, and a string is written into it. The function then returns a pointer to this local buffer, which is a mistake because the buffer’s scope ends when the function returns, rendering the pointer invalid.
When the program is compiled without sanitization, this bug may go unnoticed, potentially leading to undefined behavior at runtime. However, when compiled with ASan using `clang -O3 -fsanitize=address -o bug bug.c`, the error is caught:

- ASan outputs an error message indicating that there was a read of size 1 at a memory address that is part of the stack (i.e., the local `buffer`) after the scope of the `buffer` has ended.
- The error message includes a stack trace that shows where the invalid read occurred, in this case, the `puts` function called from `main`, which attempts to print the contents of the now non-existent buffer.
- The output also pinpoints the exact location in the source code where the invalid access occurs, aiding in debugging the issue.

14.5.2.2 ASan detects (1)

- Out-of-bounds accesses to heap, stack and globals

```
int a[10];
printf("%d\n", a[20]);
```

- Use-after-free

```
free(pointer);
printf("%d\n", *pointer);
```

14.5.2.3 ASan detects (2)

- Use-after-return

```
int *f()
{
  int a[10];
  return a;
}

void g()
{
  int *pointer = f();
  printf("%d\n", pointer[0]);
}
```

- Use-after-scope

```
void g()
{
  int *pointer;

  if (1)
  {
    int a[10];
    pointer = a;
  }

  printf("%d\n", pointer[0]);
}
```

14.5.2.4 ASan detects (3)

- Double-free, invalid free

```
void *other_pointer = pointer;
free(pointer);
free(other_pointer);

int a[10];
free(a);
```

- Memory leaks

```
void f()
{
    void *ptr = malloc(10);
}
```

Pros

- Detects most memory issues
- No false positives

Cons

- Not every memory issue detected (many are)
- Overhead (∼2x slowdown)
- Needs good tests

14.5.3 Valgrind

- Valgrind adds runtime checks on already-compiled executable.
- It is a hybrid interpreter / JIT compiler for machine code.
- It adds checks around all memory accesses.
  - ⋆ Detects uses of invalid pointers (incl. uninitialized memory)
  - ⋆ Detects memory leaks (at exit)
  - ⋆ Valgrind requires compiling with the "-ggdb" option (gcc / clang)

14.5.4 Pros

- Detects almost all memory issues (that happen at runtime)

14.5.5 Cons

- Large overhead (∼10x slowdown)
- Needs good tests

14.6 Fuzzing

14.6.1 We need good tests

- Dynamic analysis tools are useful
- but only if we have good test cases
- and enough of them
- ⇒ How do we generate good tests?

On a basic level, a fuzzer proceeds as follows:

1. take a (mostly valid) example input file
2. run the tested program with that input file
3. check for crashes
4. modify the input file:
    - random modifications
    - truncations, duplications
    - mergers with other example input files
5. go back to 2

14.6.1.1 Advanced fuzzers

- use test coverage techniques
  to determine which input files are "interesting",
  in that they cover previously-uncovered source code
- use static analysis techniques
  to determine input file modifications that could trigger specific code branches

14.6.1.2 AFL++

- open source project (`https://aflplus.plus/`)
- takes as an input a directory with many (mostly valid) example input files
- generates modified input files that (try to) yield crashes

15 Debugging

15.1 Debugging Techniques

15.1.1 Instrumentation

Instrumentation is a fundamental technique in debugging. It involves inserting additional code to monitor the execution of a program to ensure that the assumptions about the program's behavior hold true.

Key concepts

- The primary goal is to confirm that what we *think* is true is *actually* true.
- It helps in pinpointing the exact location where execution diverges from our expectations.
- Common instrumentation techniques include:
    - ⋆ Assertions: `assert` or `assert()`
    - ⋆ Debugging messages: `print()` or `printf()`
    - ⋆ Machine-readable output for automated debugging tools.

15.1.2 Crash instrumentation example

Instrumentation can be used to identify the point of failure in a sequence of actions:

```
void perform_actions(struct state *s)
{
    action_a(s);
    action_b(s);
    action_c(s);
    action_d(s);
    action_e(s);
}
```

By adding debugging messages, we can observe the execution flow:

```
void perform_actions(struct state *s)
{
    printf("Action A...\n");
    action_a(s);
    printf("Action B...\n");
    action_b(s);
    printf("Action C...\n");
    action_c(s);
    // If the program crashes after "Action C...", we know the issue is
    in action_c()
    printf("Action D...\n");
    action_d(s);
    printf("Action E...\n");
    action_e(s);
    printf("Actions done.\n");
}
```

15.1.3 Machine-readable output example

Machine-readable output is particularly useful for automated testing and debugging frameworks:

```
def matrix_inverse(mtx):
    # ...
    return result
```

To verify the correctness of the output, we can add checks that output errors in a machine-readable format:

```
def matrix_inverse(mtx):
    # ...
    error_matrix = mtx * result - matrix_identity()
    matrix_write(mtx, "mtx.m")
    matrix_write(result, "result.m")
    assert matrix_norm(error_matrix) < 1e-5
    return result
```

Note: The example assumes that there are no undefined behaviors, such as accessing invalid memory, that could cause the program to crash before reaching the problematic code.

15.1.4 How to handle large test cases?

- Suppose the `matrix_inverse()` function has a bug.
- The issue is observed with a specific 2000×2000 matrix.
- Directly visualizing such a large matrix to debug is impractical.

15.1.4.1 Strategies for Instrumentation

- Instrumenting `matrix_inverse()` by printing the matrix at each step can be overwhelming due to the matrix's size.
- An effective approach is needed to handle the large size of the test case.

15.1.5 Test Case Reduction

Test case reduction is a technique to reduce the complexity of the test case while preserving the bug.

Steps for Reduction
Given a matrix $A \in \mathbb{R}^{n \times n}$:

1. Construct a smaller matrix $B \in \mathbb{R}^{m \times m}$ by selecting a square submatrix of A.
2. Test `matrix_inverse()` on B.
3. If `matrix_inverse(B)` fails, replace A with B and repeat the process.
4. Continue iteratively until a sufficiently small matrix that reproduces the bug is found.

Example Approach

- Initially, try removing a random half of the rows and columns of A.
- If the problem persists, reduce the number of rows and columns incrementally.
- If necessary, narrow down to removing a single row and column.

Note: This process can be automated, allowing for systematic and efficient debugging of large test cases.

15.1.6 Code Bisection

Code bisection is a systematic method for identifying the location of a bug in the code by repeatedly dividing the range of code under suspicion and verifying at each division point.

Applying Code Bisection

1. Begin with a range of code where the bug is suspected.
2. Insert debugging statements at the midpoint to determine if the first half is executed without error.
3. Based on the outcome, narrow down the range to the half where the bug occurs.
4. Repeat the process with the narrowed range until the exact location of the bug is identified.

Example of Code Bisection
Consider a function `perform_actions` which executes a series of actions:

```
void perform_actions(struct state *s)
{
    action_000(s);
    ...
    action_999(s);
}
```

To locate the bug, we instrument the code with print statements:

```
void perform_actions(struct state *s)
{
    printf("First action...\n");
    action_000(s);
    ...
    printf("Action 500...\n");
    action_500(s);
    // If a crash occurs after this print statement, the bug is between
     action 500 and 999.
    ...
    action_999(s);
    printf("Actions done.\n");
}
```

Narrowing Down the Range
Through iterative bisection, we can narrow the range further:

```
void perform_actions(struct state *s)
{
    // Inserting additional print statements between the narrowed range
    printf("Action 750...\n");
    action_750(s);
    // The crash between 750 and 999 indicates the half where the bug
   resides.
    ...
}
```

Continuing this process will eventually isolate the action causing the crash, allowing for a targeted investigation and resolution of the bug.

15.1.7 Version bisection

Version bisection is used to identify a change in the codebase that introduced a bug by checking out different commits and testing them.

Using Git for Bisection
The ‘git log‘ command is used to list commits in a concise format, and ‘git bisect‘ can automate the bisection process:

```
git log --oneline
```

A list of commit messages may look like this:

```
9e9e6fc (HEAD -> main, origin/main) Added perf version check.
ff3c21b Changed branch mispredict ratio displayed.
fd49f78 Silently ignore branch events.
85afe03 Support new perf-script brstack format with added spaces.
77f8759 Made perf script output parsing more lenient.
637f374 Version bump.
47b578b Fixed erroneous use of atime, should have been mtime.
1dadd0f Moved objdump cache to /tmp.
60a534a Added caching of objdump output.
6f3c377 Some debugging code.
b2daa9b Updated version.
```

To find the commit that introduced a bug, we can use ‘git bisect‘ to mark the current version as bad and an earlier version that is known to be good. Git will then checkout a commit halfway between the two, and we can compile and test that version:

```
git bisect start
git bisect bad # mark the current version as bad
git bisect good b2daa9b # mark commit b2daa9b as good
```

Note: You would continue this process, marking each tested commit as ‘good‘ or ‘bad‘, until ‘git bisect‘ identifies the commit that introduced the bug.

15.2 Debuggers

- A **debugger** is a tool that allows us to run our code step-by-step (e.g., line by line)
- Between each step, we can examine
    - ⋆ program **output**

    - ⋆ program **state** (i.e., variables)

- Debuggers for interpreted languages are language-specific
- Debuggers for compiled languages work at the assembly level

16 Lecture 16

16.1 Performance

16.1.1 What is performance?

We want computers to perform required actions while minimizing their use of **resources**:

- Time
- Power use (mobile, servers)
- Network use (mobile)
- Memory use (peak RAM usage)
- Storage (solid state drive / hard disk drive)

We care about those resources in proportion to their cost (financial, emotional, etc.)

16.1.2 How do we achieve good performance?

High level to low level:

1. Pick a good algorithm
   - specifically, an algorithm with low computational complexity:
   - $O(\log(n))$ better than $O(n^2)$ better than $O(n^6)$ better than $O(2^n)$
   - we can hope for great improvements, $100\times$ faster, $1000\times$, etc.
   - $\rightarrow$ first thing to try!
2. Pick an algorithm that is fast on the computers you use and implement it well
   - this course!
   - smaller improvements: from a few percent to $50\times$ faster
3. Translate the implementation into efficient instructions (compilers do that well)

16.1.3 What hides inside the big-O?

Let us compare two algorithms:

- Algorithm A has complexity $O(n^2)$
- Algorithm B has complexity $O(n\log(n))$

Specifically,

- Algorithm A performs $2n^2 + 16$ operations
- Algorithm B performs $16n\log(n) + 64$ operations

Graph showing that, for smaller n, $O(n^2)$ is be better even though it has a worse big-O scale.

Algorithm A	Algorithm B
$2n^2 + 16$	$16n\log(n) + 64$

16.1.4 Which algorithm do we choose?

Assume that

- Algorithm A is insertion sort
- Algorithm B is merge sort

16.1.5 Merge sort example

The given charts illustrate the performance characteristics of two sorting algorithms, Algorithm A (Insertion Sort) and Algorithm B (Merge Sort), across different input sizes.

- In the first chart, we see that for smaller values of n, Algorithm A performs competitively, but as n increases, its performance degrades significantly compared to Algorithm B.
- The takeaway is that we should choose the right algorithm based on the size of the input: for smaller datasets, Algorithm A may be more efficient, whereas for larger datasets, Algorithm B is preferable.
- The second set of charts emphasizes that we can do even better by combining algorithms. For small datasets, we can use Algorithm A, and as the dataset grows, we can switch to Algorithm B. This combination approach leverages the strengths of both algorithms.
- The right chart in the second image shows that combining algorithms results in a performance that is close to the best of both worlds: the efficiency of Algorithm A for small datasets and the scalability of Algorithm B for larger ones.

This strategy is often employed in practice, such as in the implementation of sorting functions in standard libraries, which use a hybrid approach (like Timsort in Python and Java) that adapts to the nature of the dataset to provide optimal sorting performance.

16.2 CPU Pipelines

16.3 Back to instructions

Instruction decoding:

```
48 2b 06       sub    rax, QWORD PTR [rsi]
48 af          imul   rax, QWORD PTR [rsi]
48 af af 06    imul   rax, QWORD PTR [rdx]
```

From `48 af af 06`, the CPU needs to understand:

- that it must perform a multiplication (as opposed to, say, a subtraction)
- that one term is the value of a 64-bit register, `rax`
- that the other term comes from memory: the 64-bit value pointer to by `rsi`

16.3.0.1 What happens after instruction decoding?

`48 af af 06 imul rax, QWORD PTR [rdx]]`

- the CPU has Boolean circuitry to compute multiplications
- it must ensure that one of the two inputs of the multiplier is `rax`
- the CPU has Boolean circuitry to access memory
- it must ensure that the input of the memory circuitry is `rsi`
- it sets the second input of the multiplier to the output of the memory circuitry
- it stores the output of the multiplier back to `rax`

It is no longer possible to do all this in a single cycle
(e.g., at 4 GHz, i.e. 4 billion cycles per second, so in 0.25 ns)

16.3.0.2 Imagine that it takes:

- one cycle to decode an instruction
- one cycle to fetch data from memory
- one cycle to perform arithmetic

16.3.0.3 Sequential Execution of CPU Instructions

The provided diagrams illustrate the sequential execution of two assembly instructions in a non-pipelined CPU architecture. Each instruction goes through the decode, memory, and arithmetic stages one at a time without overlap. Considering one cycle for each stage, the execution timeline is as follows:

1. The first instruction, `sub rax, [rdi]`, involves subtracting the value at the memory location pointed to by `rdi` from the `rax` register. The execution stages are:
   - Cycle 1: Decoding the `sub` instruction.
   - Cycle 2: Fetching the operand from memory `[rdi]`.
   - Cycle 3: Performing the subtraction and updating the `rax` register.
2. After the first instruction completes all its stages, the second instruction, `imul rbx, [rsi]`, which multiplies the `rbx` register by the value at the memory location pointed to by `rsi`, begins its execution:
   - Cycle 4: Decoding the `imul` instruction.
   - Cycle 5: Fetching the operand from memory `[rsi]`.
   - Cycle 6: Performing the multiplication and updating the `rbx` register.

Each instruction takes 3 cycles
However, in this model,

- while the memory circuitry is busy fetching QWORD PTR [rsi], the multiplier is idle
- while the multiplier computes the result, the memory is idle
- during instruction decoding, everything else is idle

We can exploit this!

16.3.1 Pipelined execution

The images demonstrate the pipelined execution of instructions in a modern CPU. This execution model allows multiple instructions to be processed concurrently at different stages of the instruction cycle. Given the scenario where each stage (decode, memory access, arithmetic operation) takes one cycle to complete, the pipeline allows for a new instruction to enter the pipeline before the previous instruction has finished, thus improving the overall throughput of the system.

- Initially, the pipeline is filled with the first instruction `sub rax, [rdi]` going through the decode, memory, and arithmetic stages.
- As soon as the decode stage is complete for the first instruction, the second instruction `imul rbx, [rsi]` enters the decode stage while the first instruction is in the memory stage.
- This process continues with the third instruction `add rcx, [rbp]` entering the pipeline.
- Once the pipeline is full, each cycle will see an instruction at each stage of execution, effectively executing multiple instructions simultaneously.

The pipelined execution is visualized in the images, with each horizontal level representing a cycle and each stage of instruction processing moving one step down the pipeline in each subsequent cycle. The advantage of this model is that the CPU does not have to wait for one instruction to complete all its stages before starting the next one, which significantly increases the instruction throughput.

16.3.2 Throughput vs. latency

- Latency:
  - ⋆ Executing each instruction still takes 3 cycles!
- Throughput:
  - ⋆ But on average, we execute up to 1 instruction per cycle.

16.3.3 Data dependencies

The images depict the execution of two assembly instructions that have a data dependency. Data dependencies occur when an instruction requires the result of a previous instruction.

1. The first instruction, `mul rcx, [rdx]`, performs a multiplication operation with the value at the memory location pointed to by `rdx` and stores the result in `rcx`.
2. The second instruction, `add rdx, [rsi]`, adds the value at the memory location pointed to by `rsi` to the register `rdx`.

Given the following conditions:

- Each stage (decode, memory, arithmetic) takes one cycle to complete.
- An instruction must wait for the previous instruction to complete its memory and arithmetic stages if there is a data dependency.

The execution proceeds as follows:

- The `mul` instruction is decoded first, then the corresponding value is fetched from memory, and finally, the multiplication is performed.
- The `add` instruction must wait until the `mul` instruction has completed its arithmetic stage before it can proceed with the memory fetch, as it depends on the updated value of `rdx`.
- This results in a stall in the pipeline, where the `add` instruction's decoding is delayed until the `mul` instruction's result is written back, ensuring the correct value is used for the addition.

The pipelined execution is disrupted by the data dependency, causing delays or stalls, which are illustrated in the sequence of images.

16.3.4 Conditional branching

Conditional branching poses a unique challenge in CPU instruction pipelines. When an `if` statement is encountered, the CPU must determine which path to follow – the true branch or the false branch. The provided diagrams illustrate the step-by-step decision-making process in a simplified CPU pipeline.

1. Initially, the CPU decodes the comparison instruction `cmp rdi, rsi`. This instruction sets up the condition flags based on the comparison of the values in the `rdi` and `rsi` registers.
2. Once the `cmp` instruction is decoded, the CPU moves on to decode the subsequent `jge` (jump if greater or equal) instruction. However, the jump cannot be performed until the result of the `cmp` instruction is known, which requires waiting for the arithmetic stage to complete.
3. If the condition is true, the CPU follows the branch to the instruction labeled `.L1` (in this case, representing the code block `YYY`). Otherwise, it continues to the next instruction in sequence (`zzz`).

4. During this decision phase, the pipeline may stall, waiting for the comparison result to decide the next instruction to fetch and decode. This waiting period is where branch prediction can play a significant role in maintaining pipeline efficiency.

16.3.5 Branch prediction

- in practice, the CPU will try to predict which branch will be taken (based on past choices at that specific instruction)
- and speculatively choose that branch

16.3.5.1 Example

- Initially, the CPU encounters a conditional branch instruction (`jge .L1`) and speculatively decodes the subsequent instructions (YYY) assuming the branch will not be taken.
- Meanwhile, the comparison instruction (`cmp rdi, rsi`) is executed to evaluate the actual condition. Until this condition is evaluated, the CPU continues to decode and execute the speculatively chosen path.
- If the branch prediction is correct, the CPU continues execution without any interruption, leading to higher performance due to reduced waiting time.
- However, if the prediction is incorrect, as depicted in the final images, the CPU must discard or "flush" the speculatively executed instructions (YYY) and redirect execution to the correct path (ZZZ). This action is known as a pipeline flush, which can cause a temporary decrease in performance due to the time taken to correct the path and refill the pipeline with the correct instructions.

The diagrams illustrate a scenario where the CPU's branch predictor incorrectly anticipates the direction of a branch, resulting in several cycles of mispredicted instruction execution. Once the correct path is determined (ZZZ), the pipeline is adjusted, but the mispredicted instructions (YYY) must be invalidated, showcasing the cost of a misprediction.

16.3.6 In practice

- When all goes perfect, processors can actually execute more than one instruction per cycle
- Modern processor pipelines have between 5 and 40 stages
- At each stage, there are multiple circuitry blocks (decoders, arithmetic and logic unit (ALU) "ports", etc.)
- Branch mispredict penalty is typically ≥ 10 cycles
- Main memory latency is 50–200 cycles

16.3.7 How do we write good code?

- These parameters vary widely from CPU to CPU
- Specific characteristics are often not public
- It is almost impossible to predict the number of cycles a given set of instructions will take (in the presence of branches and memory accesses)
- ⇒ Qualitatively: we try to *understand* the phenomena at play
- ⇒ Quantitatively: We *measure* at runtime

16.3.8 Out-of-order execution

Out-of-order execution is an advanced CPU technique that improves the utilization of execution units by allowing instructions behind a slow-running instruction to be executed in advance if the operands they require are available.

- In the given example, there are three assembly instructions: two addition instructions (`add rdx, rdi` and `add rcx, rbx`) and one memory read instruction (`mov rax, [rsi]`).
- The CPU starts by decoding the first instruction. If the operands are immediately available, it may dispatch this instruction to an Arithmetic Logic Unit (ALU) for execution.
- If the subsequent instruction does not depend on the first one's result and its operands are ready, the CPU may decode and dispatch this instruction to another ALU, even if the first instruction has not completed yet.
- The memory read instruction may be dispatched to the memory unit independently if the memory address is ready, potentially allowing all three instructions to be in various stages of execution simultaneously.
- This results in improved overall throughput as the CPU does not idle waiting for instructions to complete in program order. However, the CPU must ensure that the results are written back in the correct order to preserve the program's logical flow.

16.4 Memory

Access to memory ("random access memory" or RAM) is slow
On desktop computers, RAM is typically on distinct integrated circuit (IC) packages, physically centimeters away from the CPU.
Solution: **caching**

16.4.1 Caching

Level 1 ("L1") cache:

- The CPU contains a **small amount of extremely fast memory**
- This memory **requires many of logic gates** on-package
- But it is **always available** (no latency)
- The CPU contains logic to decide which part of the main memory gets stored in its L1 cache
- This continuously changes over time

Level 2 ("L2") cache:

- slower than L1
- but requires fewer logic gates, so we can have more

Level 3 ("L3") cache:

- slower than L2
- but requires fewer logic gates, so we can have more

Zen 4 Cache	**L1I cache**	**L1D cache**	**L2 cache**	**L3 cache**
Cache size	32kB	32kB	1MB	xxx
Associativity	8 way	8 way	8 way	xxx
Cache line size	64 b	64 b	64 b	64 b

Table 1: Cache configuration for Zen 4 architecture.

16.4.2 Typical configuration

- memory transits through in units of one cache line
    - ⋆ 64 bytes on x86_64
    - ⋆ 128 bytes on M1 Macs
- there is no concept of locality beyond cache lines
- every memory access is performed through L1 cache
- when all cache entries are full, we need to overwrite one
    - ⋆ → cache eviction policies e.g. least-recently used (LRU)
- pipelined CPUs feature a memory prefetcher (speculatively fills caches in advance)

16.4.3 How do we write good code?

- Again, cache operation varies widely from CPU to CPU
- It is almost impossible to predict how it will behave with complex instruction streams
- $\Rightarrow$ Qualitatively: we try to **understand** how caches work
- $\Rightarrow$ Quantitatively: We **measure** at runtime

17 Benchmarking and static instrumentation

17.1 More caches and pipelines

Virtualization of memory involves translating virtual addresses to hardware addresses, which is essential for every memory access operation. This translation relies on a page table, which is itself stored in memory. This process might suggest the need for two memory accesses per memory fetch instruction, which is inefficient.
Solution: To optimize this, a portion of the page table is cached in the CPU in the Translation Lookaside Buffer (TLB), reducing the frequency of memory accesses.

17.1.1 Cache latency

Caches are implemented at various levels to mitigate access latency. Below are some typical latencies for different technologies and operations:

Operation	**Typical Latency**	
Instruction	~ 0.25 ns	0.25 ns
RAM	~ 100 ns	100 ns
Solid State Drive (SSD)	~ 0.2 ms	$200,000$ ns
Hard Disk Drive (HDD)	~ 2 ms	$2,000,000$ ns
Wired Ethernet (round-trip)	~ 1 ms	$1,000,000$ ns
WiFi Latency (round-trip)	~ 10 ms	$10,000,000$ ns
Same-city Internet (round-trip)	~ 5 ms	$5,000,000$ ns
Same-continent Internet (round-trip)	~ 25 ms	$25,000,000$ ns
Transatlantic Internet (round-trip)	~ 100 ms	$100,000,000$ ns

17.1.2 Examples of caches

Caches are ubiquitous in computing and networking:

- SSDs have internal RAM caches, typically ranging from 0-4 GB.
- Operating systems cache files in memory to speed up access.
- Large content providers, such as Google, Amazon, Netflix, and Cloudflare, implement extensive caching mechanisms globally.

Real-world Caching Example
Here is an example showing the output of pinging servers located in different parts of the world:

```
ping canada.ca
PING canada.ca (205.193.117.159): 56 data bytes
64 bytes from 205.193.117.159: icmp_seq=0 ttl=228 time=20 ms
... (similar lines omitted for brevity) ...
64 bytes from 205.193.117.159: icmp_seq=4 ttl=228 time=18 ms
```

```
ping google.com.au
PING google.com.au (142.251.209.3): 56 data bytes
64 bytes from mi104s04-in-f3.1e100.net (142.251.209.3): icmp_seq=0 ttl=115 time=12.2 
... (similar lines omitted for brevity) ...
64 bytes from mi104s04-in-f3.1e100.net (142.251.209.3): icmp_seq=4 ttl=115 time=11.8 
```

Key points

- Understand the role of the Translation Lookaside Buffer (TLB) in reducing memory access latency.
- Be able to compare the latencies of different memory and storage technologies, from CPU instructions to transatlantic internet round-trips.
- Recognize the impact of caching at various system levels, including hardware (like SSDs) and software (like operating system file caches).
- Consider the real-world implications of caching, such as the performance differences observed when accessing local vs. remote resources.

17.1.3 Examples of pipelines

- Storage devices:
  - ⋆ SSDs typically access data in "pages" of 4096 bytes
  - ⋆ 0.2ms SSD latency would imply a max speed of 20 MB/s
  - ⋆ instead SSDs routinely read and write 500 MB/s
- Networks:
  - ⋆ Network packets are typically 1500 bytes
  - ⋆ 10ms WiFi latency would imply 150 KB/s
  - ⋆ instead most WiFi networks do at least 10 MB/s
- Browsers:
  - ⋆ Google Chrome maintains up to 6 connections per domain

17.2 Benchmarking

When profiling applications in Unix-like systems, the 'time' command is used to measure the duration of program execution.

Time Command Output
Running the 'time' command with an application gives us three key pieces of information:

- **real**: This is the elapsed "real" time or wall-clock time from start to finish of the call.
- **user**: This time is spent in user mode, that is, the time the CPU takes to run the application code.

- **sys**: This is the time spent in system mode, which means the time the CPU takes to run OS kernel operations on behalf of the application.

It's important to note that the sum of user and system times ($user + sys$) will typically be less than the real time, as real time also includes time spent waiting for I/O or other applications.

17.2.1 Variance in Execution Time

Execution time can vary between runs due to a variety of factors, such as system load, I/O overhead, or CPU scheduling. Variance is observed when running the same command multiple times:

```
time ./application
```

For example, running a command to read from '/dev/random' may yield different 'real', 'user', and 'sys' times upon repeated executions.

Examining Variance
To understand the stability and performance of a system or application, examine the variance in execution time over multiple runs. For instance, running the following command multiple times:

```
time head -n 1000000 /dev/random > /dev/null
```

will provide different execution times, which is an example of the inherent variance in system performance.

17.2.2 Reasons for variance

Variability in execution time can often be attributed to several factors, which include but are not limited to:

- **Power and Temperature Throttling:** Modern CPUs can adapt their operating speed to avoid overheating, leading to performance changes.
- **Interactions with Devices:** The operating system has in-memory caches for files, and storage devices have internal memory caches, which can affect the timing of operations.
- **Other Processes:** The necessity to share system resources among multiple processes can introduce variability in the execution time of a program.

17.2.3 Effect of file caches

File caching by the operating system can have a significant effect on the performance of file operations. Below is an example showing the execution of the 'md5sum' command on a large file, first without file caches and then with the file cached in memory:

```
time md5sum 2GB_file
```

In this example, the second run of ‘md5sum‘ on the same file is faster due to the effect of file caching.

Key Points

- Understand how CPU power management can affect execution times.
- Be aware of the role of the OS and hardware in caching data, and how it impacts performance measurements.
- Remember that concurrent processes compete for resources and can cause measurable differences in performance.
- Practice interpreting output from system monitoring tools to diagnose performance issues.

17.2.4 Inaccuracies

- executable startup is slow
- initialization adds overhead
- input and output are slow

17.2.4.1 Executable startup is slow

The process of starting an executable can introduce overhead that affects the accuracy of performance benchmarking, especially for applications that only run for a few milliseconds.

- Compiling and running an empty program in C still incurs a measurable startup time.
- Interpreted languages like Python also have a significant startup time even for executing a simple exit command.

17.2.4.2 Initialization adds overhead

Initial setup or parsing steps in applications add overhead, which might dominate the execution time for short-running processes.

- For example, timing the execution of a simplex algorithm might inadvertently measure the time taken by the file parser rather than the algorithm itself.

17.2.4.3 Input and output are slow

I/O operations can be a major source of slowdown in applications.

- A Python function calculating the Riemann zeta function demonstrates this when its output is printed iteratively.

- The execution time drastically increases when printing results inside the loop compared to a single print at the end.

Takeaway
It is essential to understand that benchmarking should focus on the algorithm or the specific task of interest, avoiding extraneous operations that do not contribute to the performance measurement of the targeted task.

17.2.5 Recommendations for Accurate Benchmarking

- Isolate the core functionality from initialization and termination procedures when timing an application.
- Minimize I/O operations during timing, especially when dealing with high-performance code.
- Be aware of the environment and system state, as concurrent processes and system load can impact measured performance.

17.2.6 Aggregate measures

- if we benchmark our code on different inputs, we may want to use
  - ⋆ total time / average time
  - ⋆ geometric mean
  - ⋆ or other aggregate measures
  - ⋆ or some visualization (bar graphs, performance profiles, etc.)
- but beware: all aggregate measures are biased

	Input 1	Input 2	Input 3	Average
Version A	2530s	2300s	12s	1614s
Version B	2535s 1.002x	2304s 1.002x	6s 0.5x	1615s

17.3 Static instrumentation

When it comes to performance optimization, it is crucial to focus on specific parts of the code. Static instrumentation is a method for inserting code to measure the execution time of particular segments directly.

17.3.1 Why Benchmark Specific Parts of Code?

Benchmarking specific parts of code is important for several reasons:

- To bypass the time consumed by executable startup, initialization, and I/O operations, which can obscure the true performance of the code segments.
- To accurately measure the performance of code sections that execute quickly and might otherwise be overshadowed by the startup overhead.

- To identify bottlenecks, which are the parts of the code that significantly impact the overall performance.

17.3.1.1 Adding Timing Instrumentation

To obtain accurate measurements, developers should add timing code around the functions or blocks of interest. This allows for granular analysis of where most of the execution time is being spent.

17.3.2 About bottlenecks

Donald Knuth famously said, "Premature optimization is the root of all evil." This highlights the importance of identifying true bottlenecks before optimization:

- A small section of code could be responsible for a disproportionate amount of the execution time, which could be a target for optimization.
- Focusing on optimizing non-critical parts of the code without understanding the actual performance impact can lead to wasted effort and complexity.

17.3.2.1 Identifying Bottlenecks with Static Instrumentation

For instance, consider a program with three functions where static instrumentation has revealed the following time consumption:

- `function_A()` takes up 12% of the time, with 500 lines of code.
- `function_B()` consumes 60% of the time, but it only has 20 lines of code.
- `function_C()` uses 18% of the time for its 80 lines of code.
- All other code combined takes up the remaining 10% of the time.

With this information, it is clear that `function_B()` is the primary bottleneck and should be the focus of optimization efforts. Despite its small size, it has the greatest impact on performance.

17.3.3 Using time.time()

In Python, ‘time.time()‘ is used to get the wall-clock time at various points during execution. Here’s an example of how to use it:

```
import time
t0 = time.time()
initialize()
# ... run various functions ...
cleanup()
t5 = time.time()
print(f"Total time: {t5 - t0}")
```

This method is simple but has limitations due to the granularity and precision of the time measured.

17.3.4 Using clock_gettime()

In C, ‘clock_gettime()‘ provides a more precise measurement of execution time. It can be used like this:

```
struct timespec t0, t1, t2, t3, t4, t5;
clock_gettime(CLOCK_MONOTONIC, &t0);
initialize();
# ... run various functions ...
cleanup();
clock_gettime(CLOCK_MONOTONIC, &t5);
print_all_clocks(&t0, &t1, &t2, &t3, &t4, &t5);
```

‘$CLOCK_MONOTONIC$‘ ensures that the time measured is not affected by discontinuous jumps in the system time.

17.3.5 Cumulative time

To measure cumulative time across multiple iterations or function calls, you can sum the time differences for each specific part:

```
for i in range(1000000):
    t0 = time.time()
    function_A()
    tA += time.time() - t0
    # ... similarly for functions B and C ...
```

17.3.5.1 Caveats in Timing

- Measuring time itself takes time, which can introduce an overhead (40 ns for ‘time.time()‘).
- The actual time may fluctuate due to various system factors.

17.3.6 Microbenchmarks

For functions that execute very quickly, microbenchmarking can be used to get a more accurate measure by running the function many times and measuring the total time:

```
tA, tB, tC = 0
for i in range(5000000):
    t0 = time.time()
    function_A()
    tA += time.time() - t0
    # ... similarly for functions B and C ...
```

17.3.6.1 Key Takeaways

- When benchmarking, it's important to isolate the part of the code you're interested in.
- Take into account the overhead introduced by the timing mechanism itself.
- Use cumulative timing and microbenchmarking for a more precise performance analysis, especially for quick-executing code.

17.3.7 Microbenchmarks limitations

- It may not make sense to call `function_A()` in isolation
  - ⋆ Take `sin(x)` for example: which value of x do we choose?
  - ⋆ Always the same?
    - ∗ Are we sure `sin(0)` takes as much time as `sin(0.1)`?
  - ⋆ A random value for x?
    - ∗ What if generating pseudo-random values takes more time than `sin()`?
- What about caches?
  - ⋆ Caches will be "hot" (already filled with relevant data)
  - ⋆ Microbenchmarking presents an over-optimistic picture of memory access times

17.4 Automated instrumentation: profilers

17.4.1 gprof

'gprof' is a profiling tool that helps identify sections of code that take up significant execution time. It automatically instruments the code, collects data on the program's performance, and generates a report.

17.4.1.1 Using gprof

To use 'gprof', follow these steps:

1. Compile the program with profiling enabled using '-pg' option:
   `gcc -O3 -o app app.c -pg`
2. Run the application normally:
   `./app`
3. Generate the performance report:
   `gprof app`

The generated report includes a flat profile and a call graph that show the time spent in each function.

17.4.1.2 Understanding gprof Output

The ‘gprof‘ output report contains:

- A flat profile with the time spent in each function.
- A call graph showing the relationships between functions and the time spent in each call.

17.4.2 Pros and Cons of gprof

Pros:

- It is easy to use and integrates well with existing code bases.
- Provides exhaustive profile information, including time spent in each function and the call hierarchy.
- Generally introduces low overhead to the application’s runtime.

Cons:

- The overhead can increase when bottlenecks are in small, short functions due to the granularity of the profiling.
- The accuracy of time measurement may be limited, and the actual time spent can be affected by the profiling itself.

17.4.2.1 Key Takeaways

- Understand how to compile and run a program with ‘gprof‘ to collect profiling data.
- Be familiar with interpreting the ‘gprof‘ report to identify performance bottlenecks.
- Recognize the advantages of using automated tools like ‘gprof‘ for profiling, as well as their limitations.

18 Lecture 18

18.1 Hardware performance counters

The simplest hardware-aided performance-measuring tool is: the **time stamp counter (TSC)**

- Introduced by **Intel** with the Pentium architecture (1993)
- Similar feature available on **ARM** since ARMv7 (1996)
- Special integer register
- Incremented by one at a constant rate (e.g. every clock cycle)
- Reading this register has high latency (>10 cycles)
- Useful for microbenchmarks and instrumentation
- `time.time()` / `clock_gettime()` use this internally

18.1.1 More complex performance counters

Since then, Intel and ARM have added many more performance counters:

- executed ("retired") instructions
- branches
  - ⋆ successfully predicted
  - ⋆ mispredicted branches
- memory accesses
  - ⋆ found in L1 cache
  - ⋆ L1 misses, found in L2 cache
  - ⋆ L2 misses, found in (last-level) L3 cache
  - ⋆ L3 misses, found in main memory
- TLB (page table cache) hits
- TLB misses
- Pros
  - ⋆ always measured
  - ⋆ no performance penalty
  - ⋆ no interference with normal execution
- Cons
  - ⋆ only an aggregate measure (totals)

18.1.2 Linux perf

`perf stat ./application` gives various performance metrics for ./application, like cycles, instructions, etc.

18.2 Stochastic Instrumentation

18.2.1 Limitations of performance counters

- How could we find **hot spots**
  (small groups of instructions that the application spends a lot of time running)
- What about performance counts (cache misses, mispredicted branches,...)
  at those **hot spots**?
- Instrumentation is expensive (and affects accuracy)

18.2.1.1 Solution: stochastic instrumentation

- every N cycles (e.g., every 1,000,000th cycle / every 0.1ms), a **sample** is taken
- the **sample** records:
  - ⋆ which instruction is currently being executed
  - ⋆ optionally, what it is waiting for (instr. decoding, pipeline bubble, memory access,...)
  - ⋆ optionally, instruction addresses of the last few branches
  - ⋆ optionally, whether those branches were successfully predicted

18.2.2 Stochastic instrumentation

- Pros
  - ⋆ no performance penalty
  - ⋆ no interference with normal execution
  - ⋆ accuracy naturally increases on hotspots
- Cons
  - ⋆ none

18.2.3 Analysis applications

- Linux
  - ⋆ `perf record / perf report`
  - ⋆ KDAB hotspot
- MacOS: Apple XCode Instruments
- Windows: Visual Studio ("dynamic instrumentation" / "collection via sampling")

- Intel-specific: vTune
- AMD-specific: uProf

18.2.4 Bottom-up analysis

[TBD] - Placeholder for the bottom-up analysis image from the Intel VTune Profiler

18.2.5 Flame graphs

[TBD] - Placeholder for the flame graphs image

19 Data structures: Arrays, Linkned Lists

19.1 Abstract Data Types and Data Strucutres

An abstract data type (ADT) is a model for data containers that defines the type's behavior in terms of possible values, operations, and behavior of those operations. Examples include:

- In Python: `list`, `dict`, `set`, etc.
- In C++: `std::vector`, `std::unordered_map`, etc.

ADTs specify the operations supported but not the implementation details, such as how data is stored or how the operations are executed.

19.1.1 Data Structure Implementation

A data structure is a concrete implementation of an ADT that details how data is arranged in memory and the specific algorithms used for operations, which allows us to compute the computational complexity.

19.2 Lists

Lists represent one of the simplest forms of ADTs, essentially a collection of ordered elements that support operations like:

- Storing multiple elements together.
- Optionally appending, inserting, or deleting elements.
- Accessing or modifying all elements in sequence or at a specific index (with varying efficiency).

19.3 Arrays

19.3.1 Static arrays

Static arrays are fixed-size, contiguous memory data structures that provide:

- Constant-time ($O(1)$) access or modification of elements via an index.
- Linear-time ($O(n)$) traversal to access or modify all elements due to the direct addressing capability.

19.3.2 Dynamic arrays

Dynamic arrays extend static arrays to allow variable size n, which introduces additional complexity:

- Theoretical complexity $O(n)$ for operations that change the size of the list.
- This affects the complexity of appending, inserting, or deleting elements.

19.3.2.1 Key Takeaways

- ADTs define the what, not the how, of data manipulation.
- Data structures implement the specifics of ADTs and determine the efficiency of operations.
- Understanding the difference between ADTs and data structures is crucial for choosing the right implementation for the needed operations, especially considering the performance implications.

19.3.3 Size increase

- An array occupies the bytes in memory:
  - ⋆ from "array_address"
  - ⋆ to array_address $+ n \times$ element_size $- 1$
- Increasing n has $O(n)$ complexity, because the memory at

  $$\text{array\_address} + n \times \text{element\_size}$$

  may be occupied by other data
- In that case, the dynamic array must be relocated elsewhere in memory (changing array_address)
- All $n\times$element_size bytes must be copied to the new location, hence $O(n)$ complexity

19.3.4 Size decrease

- Conversely, if the memory before and/or after an array is free,
  - ⋆ we may want to move the array
  - ⋆ in order to create a larger block of free memory
- Not doing this may cause "memory fragmentation"

In theory:

operation	complexity
access/modify element at arbitrary index	$O(1)$
increase n	$O(n)$
decrease n	$O(n)$
append an element	$O(n)$
discard last element	$O(n)$
insert an element	$O(n)$
delete an element	$O(n)$

In practice:

- Almost all implementations ignore fragmentation due to shrinking (no move when decreasing $n > 0$)

operation	complexity
access/modify element at arbitrary index	$O(1)$
increase n	$O(n)$
decrease n	$O(1)$
append an element	$O(n)$
discard last element	$O(1)$
insert an element	$O(n)$
delete an element	$O(n)$

19.3.5 Over-allocation

Over-allocation is a strategy used in managing dynamic arrays where more memory is allocated than is immediately necessary.

- We distinguish between two quantities:
  - ⋆ The user-visible size n: the number of elements the user thinks the array can hold.
  - ⋆ The allocated size a: the actual amount of memory reserved for the array.
- When the user requests an increase in the array size to n', if $n' \leq a$, the dynamic array can accommodate the new size without needing additional memory allocation.
- The allocated size a is not incremented by small amounts (no $a' = a + 1$).
- Instead, to optimize performance and minimize frequent allocations, a is typically increased exponentially, often doubling ($a' = 2a$) when the array needs to grow. This approach reduces the number of times the array must be resized and copied over the lifetime of the array.

19.3.5.1 Benefits of Over-Allocation

- **Efficiency:** By allocating more space than immediately necessary, we reduce the frequency of memory allocation operations, which are costly in terms of performance.
- **Growth Management:** Over-allocation manages the growth of an array more smoothly, preventing the system from reallocating memory for every single element added once the initial capacity is exceeded.

19.3.5.2 Implications of Over-Allocation

- While over-allocation can improve efficiency, it does mean that the data structure may use more memory than what is strictly required for its current elements.
- This trade-off between memory usage and performance is often acceptable, especially since modern computing devices typically have memory resources to spare.

19.3.6 Exponential memory allocation

Exponential memory allocation is a strategy used to manage the growth of data structures such as dynamic arrays efficiently. This method helps minimize the number of memory reallocations, which are computationally expensive operations.

- In dynamic arrays, we differentiate between the user-visible size (n) and the allocated memory size (a).
- When an array's size needs to be increased from n to n', if n' is less than or equal to a, no additional memory allocation is required.
- The allocated size (a) is not incremented linearly but rather grows exponentially, typically doubling when a size increase is necessary.

19.3.6.1 Illustration of Exponential Allocation

- Initially, an array may have an allocated size (a) larger than the user-visible size (n).
- As elements are added and n grows, if n exceeds a, the array is reallocated with twice the previous allocated size ($a' = 2a$).
- This approach is visualized in the sequence of diagrams, where the allocated size (a) increases to accommodate the growing number of elements (n).

19.3.6.2 Advantages of Exponential Allocation

- **Efficiency:** Reduces the frequency of memory allocation calls, which can be costly in terms of performance.
- **Predictability:** Provides a predictable growth pattern, simplifying the management of memory for dynamic arrays.
- **Space Utilization:** By allocating more space upfront, the overhead of reallocation is reduced, utilizing memory more effectively in the long run.

19.3.6.3 Exponential Allocation in Practice

- When an array with $n = 3$ grows to $n = 4$, the previously allocated size ($a = 4$) is sufficient.
- However, when n reaches 5, the array is reallocated with $a = 8$, doubling the previous allocation to maintain the exponential growth strategy.
- This pattern continues as n grows, with the next reallocation occurring when n exceeds 8, increasing a to 16.

19.3.6.4 key concepts

- Understand how exponential allocation minimizes the number of memory allocations needed as an array grows.
- Be prepared to explain the trade-off between immediate memory usage and long-term efficiency gains.
- Recognize the patterns of growth in allocated size versus user-visible size in dynamic arrays.

19.3.7 Pros and cons of exponential allocation

Pros:

- Reduces the frequency of memory reallocations.
- Ensures that the allocated size a is always within a factor of the actual size n, specifically $a \leq 2n$.
- Provides an $O(1)$ amortized time complexity for increasing the size of the array.

Cons:

- Can lead to memory waste as the allocated space might not always be fully utilized.
- The wasted space is bound by the equation $a = 2^{\lceil \log_2(n) \rceil}$, which can be significant for large arrays.

19.3.8 Complexity of exponential allocation

- Starting with an empty array and incrementing its size n times leads to at most $k = \lceil \log_2(n) \rceil$ memory allocation moves.
- The total cost of these moves is a power series sum: $1 + 2 + 4 + \ldots + 2^{k-1} = 2^k - 1$.
- This sum is less than or equal to $2n$, ensuring that the total time complexity for n size increments is $O(n)$.
- The amortized time complexity for each size increment is $O(1)$.

19.3.8.1 Operational Complexities

- Access or modification at an arbitrary index: $O(1)$.
- Increase or decrease n: $O(1)$ amortized.
- Append an element: $O(1)$ amortized.
- Discard the last element: $O(1)$.
- Insert or delete an element: $O(n)$ due to the need to shift elements.

operation	**complexity**
access/modify element at arbitrary index	$O(1)$
increase n	$O(1)$ amortized
decrease n	$O(1)$
append an element	$O(1)$ amortized
discard last element	$O(1)$
insert an element	$O(n)$
delete an element	$O(n)$

19.3.8.2 Key concepts

- Understand the balance between efficient allocation and potential memory waste.
- Be able to calculate the upper bound of memory allocation and the amortized cost for array resizing.
- Recognize the time complexities associated with different array operations

19.3.9 Virtual memory

Virtual memory changes the complexity landscape for operations on dynamic arrays by allowing us to remap memory instead of physically moving it.

19.3.9.1 Virtual Memory in Asymptotic Complexity

- The asymptotic complexity of changing the size n of an array involves an $O(n)$ operation due to memory moves.
- However, with virtualized memory, physical byte movement is unnecessary.
- Instead, the page table remaps the physical memory associated with a virtual address to a new virtual address, which in practice, can make memory moves essentially $O(1)$.

19.3.10 Remapping Memory Using the Page Table

- **Pros:** Memory move becomes $O(1)$ in practice due to virtual memory's ability to remap addresses instead of moving bytes.
- **Cons:** It requires a system call to the OS kernel to change the page table, which introduces:
  - ⋆ A context switch, potentially polluting the CPU caches.
  - ⋆ A significant fixed cost due to the system call overhead.
- This operation is therefore only performed when n grows very large, avoiding frequent system calls.
- To manage growth, a' is set to $a + K$ for some large K, balancing between memory efficiency and system call overhead.

For very large n (in the order of multiple megabytes), the operational complexities can be optimized:

- Access or modification at an arbitrary index remains $O(1)$.
- Increasing or decreasing n can be $O(1)$ with virtual memory's remapping capabilities.
- Appending or discarding the last element is $O(1)$.
- Inserting or deleting an element within the array still has $O(n)$ complexity due to the need to shift elements in the array.

operation	**complexity**
access/modify element at arbitrary index	$O(1)$
increase n	$O(1)$
decrease n	$O(1)$
append an element	$O(1)$
discard last element	$O(1)$
insert an element	$O(n)$
delete an element	$O(n)$

19.3.10.1 Key Concepts

- Understand the role of virtual memory in optimizing array resizing operations.
- Discuss the pros and cons of using virtual memory to remap addresses instead of moving data physically.
- Recognize the impact of system calls and context switches on the performance of dynamic arrays.

19.4 Linked Lists

Linked lists are fundamental data structures that implement a collection of elements, allowing for variable size n.

Linked lists support:

- Inserting, deleting, and modifying elements in $O(1)$ time complexity, regardless of the position in the list.
- Accessing or modifying all elements in order with $O(n)$ time complexity.

Linked lists do not natively support:

- Accessing or modifying an element at an arbitrary index, known as "random access," which would have $O(n)$ complexity if implemented sequentially.

19.4.1 Doubly-linked lists

Doubly-linked lists are an extension of linked lists where each element has a reference to both the previous and the next element.

```
struct element {
    struct payload data;
    struct element *prev;
    struct element *next;
};
```

Functionality such as insertion is demonstrated in the provided code snippet, showing how new elements are linked to their predecessors and successors.

19.4.1.1 Comparative Operations

operation	**dynamic array**	**doubly-linked list**
access/modify element at arbitrary index	$O(1)$	$O(n)$
increase n	$O(1)$	$O(1)$
decrease n	$O(1)$	$O(1)$
append an element	$O(1)$	$O(1)$
discard last element	$O(1)$	$O(1)$
insert an element	$O(n)$	$O(1)$
delete an element	$O(n)$	$O(1)$

19.4.2 Memory management considerations

Memory allocation is slow

```
struct element *x = malloc(sizeof(struct element));
```

compared to dynamic arrays' fast case

```
if (new_n <= d->sz) {
    d->n = new_n;
    return SUCCESS;
}
```

operation	**dynamic array**	**doubly-linked list**
access/modify element at arbitrary index	$O(1)$	$O(n)$
increase n	$O(1)$	$O(1)$
decrease n	$O(1)$	$O(1)$
append an element	$O(1)$	$O(1)$
discard last element	$O(1)$	$O(1)$
insert an element	$O(n)$	$O(1)$
delete an element	$O(n)$	$O(1)$

19.4.3 Memory caches considerations

Cache usage is an important aspect of performance optimization. For linked lists, memory caches do not provide the same benefits as for dynamic arrays due to the non-contiguous memory allocation of elements.

- With deep pipelines and good branch prediction, a processor can pre-fetch elements in a dynamic array, but it cannot do so as effectively with linked lists due to data dependencies.

While linked lists have fewer applications than one might expect, they can be extremely useful in situations where their specific advantages can be leveraged.

19.4.4 More options

Beyond simple linked lists, other data structures offer different performance trade-offs:

- Indirection, such as a dynamic array of pointers, allows for a level of indirection which can simplify memory management.
- In-memory tree data structures, like binary trees or n-ary trees, provide hierarchical organization and can be more efficient for certain types of operations.

```
struct nary_node {
    struct payload data;
    struct nary_node *children[MAX_CHILDREN];
};

struct dll_node {
    struct payload data;
    struct dll_node *prev_sibling;
    struct dll_node *next_sibling;
    struct dll_node *first_child;
};
```

19.4.4.1 Key concepts

- Understand the trade-offs between different data structures regarding memory allocation and access patterns.
- Recognize that linked lists can incur additional overhead due to memory allocation and may not benefit as much from cache optimization.
- Consider alternative data structures like indirection or in-memory trees for more complex needs or to improve performance.

20 Lecture 20

20.1 Abstract Data Types and Data Structures

- An abstract data type
  - ⋆ Specifies supported operations
- A data structure is an implementation of an abstract data type
  - ⋆ Specifies data layout in memory
  - ⋆ Specifies algorithms for operations

20.1.1 Lists

- support storing multiple elements together
- and optionally append, insert, delete, random access, . . .
- implementations:
  - ⋆ dynamic arrays
    - ∗ everything $O(1)$ in practice except insert/delete $O(n)$
  - ⋆ linked lists
    - ∗ everything $O(1)$ (but slower than arrays) except random access $O(n)$

20.2 Stacks / LIFO

- A stack is an ordered collection of elements
- supports two operations:
  - ⋆ "push": add an element
  - ⋆ "pop": retrieve-and-remove the last-added element
    - ∗ $\Rightarrow$ last in, first out (LIFO)

20.2.1 Static array implementation of a stack

- Useful only when there is a hard limit on the number of elements
- We maintain a static array
- and a stack pointer (or top index)
- this is how "the" stack is implemented
  (for storing function arguments, local variables and return addresses)

20.2.1.1 Example

The images illustrate this process:

1. Initially, the stack is empty, and the stack pointer points to the base of the stack.
2. As elements A, B, and C are pushed onto the stack, the stack pointer is incremented after each push to point to the next empty slot.
3. When elements are popped from the stack, the stack pointer is decremented, and the elements are removed in the reverse order they were added (Last In, First Out - LIFO).
4. If elements are pushed after popping, the stack pointer moves upwards again to fill the spaces that were previously occupied.

20.2.2 Linked list implementation of a stack

- **Pro:** No hard limit on number of elements
- **Con:**
    - ⋆ Memory allocation for every `push`
    - ⋆ Memory freed for every `pop`

20.2.3 Dynamic array implementation of a stack

- **Pros:**
    - ⋆ No hard limit on number of elements
    - ⋆ Memory management overhead is small
- **Con:**
    - ⋆ No pointer stability

20.2.3.1 Example

In a dynamic array stack, managing element pointers becomes a non-trivial task due to possible reallocations. Here's an insight into pointer stability during stack operations:

- The stack begins empty. We then push elements A, B, C, D, and E onto the stack, which increases in size dynamically. The stack pointer marks the current top of the stack.
- Taking a pointer ‘p‘ to an element, say C, allows direct manipulation of its value. But this raises a concern: what happens to ‘p‘ when the stack grows?
- If new pushes cause the stack to exceed its capacity, it must expand, often necessitating a copy of elements to a new memory block. This reallocation invalidates previous element pointers, such as ‘p‘ pointing to C, as they now reference the old memory location.

- The visual sequence illustrates changing C to C' via pointer 'p'. After stack expansion due to additional pushes, if the elements are reallocated, 'p' may still point to where C used to be, not to C' in the new stack memory layout.
- The final state, after expansion and modification through 'p', should reflect the changed value of C' at the correct position in the stack, ensuring that 'p' is updated to point to the new location of C'.

20.2.4 Arena

- Known as arena allocator, region-based allocator, zone-based allocator, obstack
- Implemented as a list of static array stacks

20.2.4.1 Further explanation

An arena allocator, also known as a region-based allocator, zone-based allocator, or obstack, is a memory management model particularly suitable for situations where objects are allocated and deallocated in a predictable pattern.

- The arena allocator manages memory in large blocks or "arenas."
- Each arena can contain a stack or a pool of objects.
- Memory within an arena is allocated sequentially, and objects are typically deallocated all at once.
- This model is highly efficient when objects have similar lifetimes, as it minimizes fragmentation and overhead associated with individual deallocation.

The images illustrate the process of allocating and deallocating objects within an arena. The operations shown include pushing elements onto the stack and then popping them off, which corresponds to allocating and deallocating memory in the arena.

- Initially, the stack pointer points to the base of the arena, indicating that no objects are allocated.
- As objects are pushed onto the stack, the stack pointer moves up, marking the end of the used space.
- When objects are popped off the stack, the stack pointer moves down, indicating that space is now available for future allocations.
- The allocation is fast and efficient since it involves only incrementing or decrementing the stack pointer.

Memory allocation in an arena is thus done in a LIFO order, and all objects within an arena can be deallocated at once by resetting the stack pointer to the base of the arena.

20.2.5 Stack implementations

Implementation	Size	Requires allocations	Pointer stability
static array	constant	no	yes
dynamic array	can grow	when growing	no
linked list	can grow	every push	yes
arena	can grow	when growing	yes

20.2.6 More options

Combination of other data structures and indirection can be used, depending on desired properties.

20.3 queues / FIFO

A queue is an ordered collection of elements

- supports two operations:
    - ⋆ `enqueue`: add an element
    - ⋆ `dequeue`: retrieve-and-remove the earliest-added element
- ⇒ first in, first out (FIFO)

20.3.1 Ring buffer implementation of a queue

- Useful only when there is a hard limit on the number of elements
- We maintain a static array
- and two pointers/indices: head and tail

20.3.1.1 Example

These operations are illustrated through the following example:

1. Start with an empty queue.
2. **Enqueue A:** The queue is now: A.
3. **Enqueue B:** The queue is now: A, B.
4. **Dequeue:** Remove A. The queue is now: B.
5. **Enqueue C:** The queue is now: B, C.
6. Continue this process, adding to the tail and removing from the head.

As we enqueue elements, they take their place at the tail end of the queue. When we dequeue, we remove the element from the head, adhering to the FIFO principle.

20.3.2 Implementation detail

When head == tail:
 the queue is empty?   or the queue is full?

- maintain a variable with the number of elements currently in the queue
- or keep incrementing head and tail, and index the static array as

  array[head % size]   and
  array[tail % size]

20.3.3 Applications of ring buffers

- Audio playback/recording devices
- Video capture devices
- Special case (double-buffering, i.e. size = 2) for computer graphics
- Network devices (routers, switches)

20.3.4 More options

- use dynamic arrays
- use linked lists
- use indirection
- ...
- depending on specific needs

20.4 Priority queues

- A priority queue is a collection of elements, each with an associated priority
- supports two operations:
  - ⋆ "push": add an element-priority tuple
  - ⋆ "pop": retrieve-and-remove the highest-priority element

20.4.1 Implementation of a priority queue

- Store element-priority tuples in an array or in a linked list
- "push": $O(1)$ of the underlying data structure
- "pop": scan all elements, find max priority, $O(n)$

20.4.2 Binary heap implementation of a priority queue

[TBD]

- Binary heaps represent a priority queue as
  - ⋆ a binary tree (every node has at most two children)
  - ⋆ that is complete (every level full, except possibly the deepest)
- Every node is labeled by the corresponding element's priority
- Tree has the heap property:
  - ⋆ Priority of any node $\geq$ priority of its children
  - ⋆ $\Leftrightarrow$ Priority of any node $\geq$ priority of all its descendants

20.4.3 Binary heap push

- Step 0: Add new element at the first free slot on the deepest level
- Step 1:
  - ⋆ If its priority is *not higher* than its parent's,
    - ∗ the heap property is *satisfied*, we are done
  - ⋆ If its priority is *higher* than its parent's,
    - ∗ swap them,
    - ∗ go back to Step 1, looking at the pushed element's new position

20.4.4 Binary heap pop

[TBD]

- Step 0: Replace root with last element (on deepest level)
- Step 1:
  - ⋆ If its priority is **not lower** than its children's,
    - ∗ the heap property is **satisfied**, we are done
  - ⋆ If its priority is **lower** than one of its children's,
    - ∗ swap with the highest-priority child,
    - ∗ go back to Step 1, looking at the pushed element's new position

20.4.5 Binary heap operations

- Push: $O(\log_2(n))$
- Find max: $O(1)$
- Pop: $O(\log_2(n))$

20.4.6 Complete binary data structure

- Binary heaps are complete binary trees
- We can avoid allocation for every "push" by storing nodes in an array
- Depth ℓ of the tree has at most 2^ℓ nodes, $\forall \ell$
- Depth ℓ of the tree has exactly 2^ℓ nodes, except for the deepest level

depth	0	1	1	2	2	2	2	3	3	3	3	3	3	3	
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- There are exactly $(2^\ell - 1)$ nodes of with depth $< \ell$

20.4.7 Storage scheme

[TBD]

depth	0	1	1	2	2	2	3	3	3	3	3	3	3	3	
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- If a node has index j
- its children are stored at indices $2j + 1$ and $2j + 2$
- its parent is stored at index $\lfloor (j - 1)/2 \rfloor$

20.4.8 Binary heap with array storage

- Superior to in-memory tree (with pointers)
  - ⋆ We avoid allocation for every "push"
  - ⋆ We avoid data dependencies (load node data to get pointer to parent/children)
- Still,
  - ⋆ Push and pop operations are tough for branch predictor
  - ⋆ Jumps to indices $(2j + 1)$, $(2j + 2)$ or $\lfloor (j - 1)/2 \rfloor$ are not cache-friendly for large j

20.4.9 Priority queue: special case

- Assume that
  - ⋆ priorities are distinct integers $p \in \{0, \ldots, P - 1\}$
  - ⋆ we always push at a priority $\leq$ current max priority
- Then,
  - ⋆ we allocate a static array of size P
  - ⋆ Push: store in array at index p $\quad O(1)$
  - ⋆ Pop: sweep array backwards $\quad O(P/n)$ amortized

20.4.9.1 Further Explanation

- A priority queue is implemented as a static array, where the index represents the priority and the array element at that index represents the queue element at that priority level.
- The queue supports two main operations:
  - ⋆ **Push operation:** This inserts an element into the queue at the position that corresponds to its priority. The operation is constant time, $O(1)$, because it places the element directly at the index of its priority.
  - ⋆ **Pop operation:** This removes the element with the highest priority that is less than or equal to the current max priority. To find this element, the array is swept backwards from the current max priority index. The amortized complexity of this operation is $O(P/n)$, where P is the size of the priority range and n is the number of operations, due to the fact that each element can be popped only once and the cost is spread over multiple pop operations.
- The maximum priority element is always kept updated to enable efficient pop operations. After a pop, the max priority is updated to the next highest priority that has an element in the queue.
- This implementation allows for efficient insertion of elements in a priority-wise fashion and also ensures that the removal of the highest priority element is done with a reasonable amortized cost.

20.4.10 Implementation details

- good for branch predictor
- great for caches
- we can store, additionally, an array of P bits ("bitmap")
  - ⋆ bit p set to one if there is an element with priority p
  - ⋆ makes "pop" operations essentially 64x faster

20.4.11 Applications of bitmap priority queues

- Linux kernel: scheduling parallel tasks
- Linear algebra: sparse matrices

20.5 Sort Operations

20.5.1 Heap sort

- Push n elements to heap: $O(n \log n)$
- Pop n elements one by one: $O(n \log n)$

$$\Rightarrow O(n \log n) \text{ worst case}$$

20.5.2 Comparison sort methods

Method	**Average**	**Worst case**	**Additional storage**
Quicksort	$O(n\log(n))$	$O(n^2)$	none
Merge sort	$O(n\log(n))$	$O(n\log(n))$	n
Heap sort	$O(n\log(n))$	$O(n\log(n))$	none

20.5.3 Special case 1

- Assume that
  - ⋆ we sort n elements with priorities $S \subseteq \{0, \dots, P-1\}$
  - ⋆ no two elements have the same priority (hence $P \geq n$)
- Then,
  - ⋆ we represent the elements as a bitmap priority queue
  - ⋆ Push: $O(n)$
  - ⋆ Pop: $O(P)$

$\Rightarrow O(n+P)$

20.5.4 Special case 2: counting sort

- Assume that
  - ⋆ we sort n elements with priorities $S \subseteq \{0, \dots, P-1\}$
  - ⋆ $P \leq n$ (we may have duplicates)
- Then,
  - ⋆ we allocate a static array `count` of size P
  - ⋆ we allocate a static array `result` of size n
  - ⋆ we count the number of occurences of each priority: $O(n)$
  - ⋆ we sweep `count` backwards to determine offsets: $O(P) = O(n)$
  - ⋆ we construct the sorted `result` list: $O(n)$

$\Rightarrow O(n)$

21 Tries, hash tables, spatial data structures

21.1 Associative Arrays

Associative arrays, also known as maps or dictionaries, are fundamental data structures in computer science and programming.

Definition and Operations

- Associative arrays are collections of key-value pairs or tuples.
- Keys can be any string of bits, such as integers or strings.
- Values are data associated with the keys.
- Operations supported by associative arrays include:
    - ⋆ Insertion: Adding a new key-value pair.
    - ⋆ Deletion: Removing an existing key-value pair.
    - ⋆ Lookup: Finding the value associated with a given key.

21.1.1 Naive implementation

A naive implementation of an associative array can be achieved using a simple list of key-value tuples.

21.1.2 Complexity

The complexity of operations in a naive implementation varies based on the underlying data structure:

	Insertion	Deletion (after lookup)	Lookup
Linked list	$O(1)$	$O(1)$	$O(n)$
Dynamic array	$O(1)$	$O(n)$	$O(n)$

21.2 Implementations using a total order on keys

21.2.1 Total order on keys

Key comparison

- Associative arrays require a mechanism to compare keys (e.g., $key_i \leq key_j$).
- In practice, key comparison is feasible by interpreting keys as large integers.
- Specialized comparison operators may be more efficient for constant-sized keys.
- Key space could potentially be infinite, accommodating arbitrary-sized keys.

21.2.2 Sorted dynamic array of (key, value) tuples

- Sorted dynamic arrays maintain keys in a total order (e.g., $key_0 \leq key_1 \leq \cdots \leq key_n$).
- Lookup operations can use binary search, leading to $O(\log n)$ complexity.

Data Structure	**Insertion**	**Deletion (after lookup)**	**Lookup**
Linked list	O(1)	O(1)	O(n)
Dynamic array	O(1)	O(n)	O(n)
Sorted dynamic array	O(n)	O(n)	$O(\log(n))$

21.2.3 Binary search tree

- A binary search tree (BST) maintains a total order invariant within its structure.
- Each node i guarantees $key_j \leq key_i$ for all j in its left subtree and $key_j > key_i$ for all j in its right subtree.
- The shape of the BST can vary greatly depending on the insertion order, which can impact performance.

21.2.4 Self-balancing binary search tree

- AVL trees
- Red-black trees
- B-trees, splay trees, treaps, ...

Data Structure	**Insertion**	**Deletion (after lookup)**	**Lookup**
Linked list	O(1)	O(1)	O(n)
Dynamic array	O(1)	O(n)	O(n)
Sorted dynamic array	O(n)	O(n)	$O(\log(n))$
Binary search tree	O(n)	O(n)	O(n)
AVL tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$
Red-black tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$

- Cache behavior: ok, not great (similar to other binary tree structures, e.g., heap)

21.3 Implementations using keys bits - tries

21.3.1 Trie

- A trie, also known as a prefix tree, is a tree structure that uses static arrays of size 2^T for routing.
- Keys are divided into chunks or "letters" of T bits.
- Each chunk provides an index into a node's static arrays.
- The path from the root to a leaf represents the complete key.

21.3.1.1 Trie Operations

A Trie is a specialized tree used to store associative arrays, where the keys are usually strings. Inserting a key into a trie involves several steps:

1. A key is represented by a series of characters. In the context of a trie, each character can be thought of as a node in the tree.
2. To insert a key, we start at the root of the trie and traverse the tree by following the path defined by the key's characters.
3. At each step, we look at the current character in the key and move to the corresponding child node.
4. If a child node corresponding to the current character does not exist, we create a new node.
5. We continue this process until all characters in the key have been processed.
6. Once we reach the end of the key, we mark the final node as an end node, which signifies the completion of a key insertion.

21.3.1.2 Trie Insertion Example

Consider the insertion of the hexadecimal keys $0x9f2$, $0x8cd$, and $0x532$ with corresponding values $V1$, $V2$, and $V3$ into a trie where each node represents 4 bits of the key.

1. For the key $0x9f2$ with value $V1$, we would insert the 4-bit segments (2, f, 9) one by one, creating a path in the trie.
2. Next, for the key $0x8cd$ with value $V2$, we would insert the segments (d, 8, c) into the trie, potentially sharing nodes with the previous key if any 4-bit segments are the same.
3. Finally, for the key $0x532$ with value $V3$, we would insert the segments (2, 3, 5) into the trie, again sharing nodes with any previously inserted keys that have matching segments.

21.3.1.3 Complexity Analysis

- Tries are highly efficient for lookups, insertions, and deletions.
- They provide a means to search for keys in a dataset of strings in $O(m)$ time complexity, where m is the length of the key to be searched.
- They are particularly useful for implementing dictionaries with prefix-based lookups.

21.3.2 Key space

The key space of an associative array refers to the range of all possible keys that can be used within the array. The concept of *dense* and *sparse* key spaces is crucial in the context of data structure efficiency.

- Let K be the set of all possible keys, and n the number of tuples in the associative array.
- A key space is considered *sparse* if $n \ll K$, meaning that the number of actual keys used is much less than the possible number of keys.
- If the number of used keys is comparable to the number of possible keys, the key space is considered *dense*.

21.3.3 "Dense" key space

- In a *dense* key space, most of the potential keys have values associated with them.
- Operations such as insertion, deletion, and lookup can be done in $O(\log Tn)$ time, where T is a factor related to the trie branching (typically the size of the alphabet or character set).
- For a dense key space, a static array can be more efficient as the operations become $O(1)$.

21.3.4 "Sparse" key space

- Tries are particularly useful for *sparse* key spaces where a static array of the entire key space would be impractically large.
- The complexity of trie operations in a sparse key space is not dependent on the number of entries but on the key size and the trie branching factor T.
- A sparse key space can lead to high memory overhead in the worst case, where every leaf node might contain only a single tuple.

A trie provides a balance between the two extremes, offering more efficient memory usage in sparse key spaces without sacrificing the performance benefits of a static array in dense key spaces.

21.4 Implementations using key bits - hash tables

Hash tables are a way to implement associative arrays by mapping a large, potentially sparse key space into a smaller, dense index space using a hash function.

21.4.1 Hash function

- A hash function h is a mapping from the key space K to an index space U, where $U \subseteq \mathbb{N}$ and the size of U is much smaller than K.
- A good hash function distributes keys evenly across the index space, minimizing the chance of collisions.

- Examples of hash functions include taking a subset of the key bits or using modular arithmetic, such as $h(k) = k \mod m$ for some integer m.

21.4.2 Hash table

- A hash table uses a static array of size $|U|$ to store values.
- Each key-value tuple (k, v) is stored in the array at the index given by $h(k)$.
- Due to the surjective nature of hash functions, collisions can occur where multiple keys are hashed to the same index.

21.4.3 Dealing with collisions

When implementing associative arrays using hash tables, collisions occur when two different keys hash to the same index. Here are some strategies to handle collisions:

21.4.3.1 Collision Resolution Strategies

- The hash table can be a static array of linked lists. When a collision occurs, the colliding elements are added to the list at the hashed index.
- Alternatively, a static array of dynamic arrays (or vectors) can be used, where colliding elements are appended to the dynamic array at the hashed index.

21.4.3.2 Search Complexity in Case of Collisions

- When collisions occur, a linear search within the linked list or dynamic array at the hash index is performed.
- If c is the maximum number of collisions at a hash index, the worst-case search complexity is $O(c)$.
- In the worst case where all elements collide at the same index, c can be equal to n, making the complexity $O(n)$.

21.4.3.3 Insertion and Lookup Operations

- To insert a key-value pair (k, v), hash the key to find the index and add the pair to the appropriate data structure at that index.
- To lookup a value for a key, hash the key to find the index, and then search through the data structure at that index to find the value.

21.4.4 Open addressing as a collision resolution method

Open addressing is a collision resolution method in hash tables. When a collision occurs, it finds another place within the hash table.

21.4.4.1 Insertion Process

1. Compute the hash index $i = h(\text{key})$.
2. If `array[i]` is empty, place the $(\text{key}, \text{value})$ tuple there. Process done.
3. Otherwise, if `array[i]` is occupied (collision), then let $i = (i + 1) \mod |U|$, and repeat from step 2.

Example of Open Addressing

```
h(k) = k mod 16
Insert (0x9f2, V1) -> h(0x9f2) = 0x2
Insert (0x8cd, V2) -> h(0x8cd) = 0xd
Insert (0x532, V3) -> h(0x532) = 0x2

// Array after insertion
// Indices:    0 1 2 3 4 5 6 7 8 9 a b c d e f
// Values:     . . V1V3. . . . . . . . . V2. .
```

21.4.4.2 Handling Collisions

- If a collision is found at index i, check the next index $i + 1$.
- Continue checking in a circular manner until an empty slot is found.
- If the table is full, the insertion fails (or rehashing is needed).

21.4.4.3 Lookup After Collision

- To lookup a value for a key after collisions, start at $h(\text{key})$ and search sequentially until:
  - ⋆ The key is found, or
  - ⋆ An empty slot is encountered, indicating the key is not in the table.

21.4.4.4 Lookup in Hash Tables with Open Addressing

The lookup process for retrieving the value associated with a key in a hash table that uses open addressing is as follows:

1. Compute the initial index i using the hash function: $i = h(\text{key})$.
2. Check if the key at the computed index matches the key we are looking for:
   - If `array[i]` matches the key, return `array[i]`.
   - If `array[i]` is empty, return **not found**.
   - If another key is found at `array[i]`, compute the next index $i = (i + 1) \mod |U|$, and go back to the previous step.

21.4.5 Probing

Probing is a technique used in hash tables to resolve collisions. It involves finding the next available slot or bucket when a collision occurs. Here's how it works during insertion:

1. **Compute initial index**: Let $h_0 = h(k)$ be the hash of key k, and initialize $j = 0$.
2. **Check the slot**: If the slot $array[i(h_0, j)]$ is empty, insert (k, v) there and terminate the insertion.
3. **Collision resolution**: If the slot is occupied, increment j and find the next slot using a probing function $i(h_0, j)$, then repeat step 2.

The probing function $i(h_0, j)$ can vary:

- **Linear probing**: $i(h_0, j) = (h_0 + j) \mod |U|$.
- **Quadratic probing**: $i(h_0, j) = (h_0 + Kj + Lj^2) \mod |U|$ for some constants K and L.

21.4.6 Good hash functions

- Naive hash functions can lead to high collision rates, even with random keys.
- Effective hash functions take a non-uniform distribution of keys over the key space K and map it to a uniform-looking distribution over a set U, minimizing collisions.
- Examples of good hash functions are Fowler–Noll–Vo (FNV), djb2, SipHash.
- These functions usually output 32-, 64-, or 128-bit numbers, which are then mapped to the table size using modulo operation: $h(k) = h_0(k) \mod |U|$.

21.4.7 Complexity of hash table operations

Performance is influenced by many factors, including:

- The density $\frac{n}{|U|}$, key distribution, hash function choice, and probing method all affect performance.
- As table density approaches 1, it's often necessary to increase $|U|$ and rebuild the hash table, known as "rehashing".

21.4.8 In practice

- Keeping the collision rate low ensures that operations like insert, delete, and lookup can remain $O(1)$ in average case.
- The initial access to a hash table often results in a cache miss, which impacts performance.
- With open addressing, collisions can affect the constant time performance due to the need for probing.

21.5 Associative Arrays: performance

Choice of Data Structure:

- There is no universally superior data structure for all scenarios; the choice between self-balancing trees, tries, and hash tables is highly dependent on the specific data and application requirements.
- It is often beneficial to perform benchmarking to determine the most appropriate data structure.

21.5.0.1 Hash Tables

- Hash tables may perform better when:
    - ⋆ Hashing operations are low-cost.
    - ⋆ Collisions are infrequent.
    - ⋆ The scale of the dataset (n) is predictable.

21.5.0.2 Self-balancing Trees

- Self-balancing trees provide:
    - ⋆ Robustness with better worst-case non-amortized complexity, particularly during operations like rehashing.

21.5.0.3 Tries

- Tries can be more efficient when working with keys that have a structured pattern, such as:
    - ⋆ Virtual address translation in page tables.
    - ⋆ IP address routing for networks.
    - ⋆ Tokenizers in natural language processing (e.g., GPT-type models).

21.5.1 Combinations of Data Structures

- Combining data structures can yield the benefits of multiple approaches and is common practice. Examples include:
    - ⋆ Hash tables implemented as a static array of self-balancing trees.
    - ⋆ Depth-k tries with self-balancing trees at the leaf nodes.

21.6 Spatial data structures

Spatial data structures are designed to manage multidimensional data, such as vectors in $\mathbb{R}^m$. They support various operations:

- Insertion: Adding a vector $\mathbf{x} \in \mathbb{R}^m$.
- Deletion: Removing a vector.

- Finding the closest vector to a given vector $\mathbf{y} \in \mathbb{R}^m$.
- Locating nearest neighbors for each inserted vector.
- Identifying k nearest neighbors for each inserted vector.
- Discovering all vectors within a certain distance d from each inserted vector.

21.6.1 The problem

Given a set of vectors, the goal is to find all pairs of vectors that are within a distance d from each other. A naive approach to this problem has a time complexity of $O(n^2)$, described by the following pseudocode:

```
R := {}
for i = 0 to n - 1:
    for j = i + 1 to n - 1:
        if ||x^i - x^j|| <= d:
            R := R union {(i, j)}
```

21.6.2 Grids

A grid-based approach divides the space into a finite number of cells, which can significantly reduce the number of comparisons needed.

Pros and Cons of Using Grids

- **Pros:**
  - ⋆ Quadratic complexity is limited to within individual grid cells.
- **Cons:**
  - ⋆ Requires finite bounds $L \leq \mathbf{x}_i \leq U$ for all vectors.
  - ⋆ Fixed cell size, which can lead to:
    - ∗ A variable number of vectors in each cell.
    - ∗ Many cells being empty, especially in sparse regions.

21.6.3 Quadtrees and octrees

Quadtrees and octrees are tree data structures which partition a two-dimensional and three-dimensional space respectively into successively smaller regions.

- They adaptively divide the space based on the distribution of the objects within.
- Each node in a quadtree corresponds to a square region of space and has four children, which correspond to the four quadrants of the square.
- Similarly, each node in an octree corresponds to a cubic region and has eight children.

21.6.4 k-d trees

k-d trees are a binary tree used to organize points in k-dimensional space.

- Nodes which are not leaves represent hyperplanes that divide the space into two parts, known as half-spaces.
- Points to the left of this hyperplane are represented by the left subtree of that node and points to the right by the right subtree.
- The tree alternates between axes used to partition the space.

21.6.5 Quadtrees, octrees, k-d trees: Pros and Cons

- **Pros:**
  - ⋆ No need for finite bounds $L \leq \mathbf{x}_i \leq U$ for all points.
  - ⋆ Cell size adapts to the distribution of the data.
- **Limitations:**
  - ⋆ Cells have a fixed shape, which can be inefficient for high-dimensional data.
  - ⋆ As the number of dimensions grows, the number of cells can grow exponentially, which is known as the curse of dimensionality.

21.6.6 Binary space partitioning

Binary Space Partitioning (BSP) is a method for recursively subdividing a space into convex sets by hyperplanes. This method is commonly used in computer graphics, computational geometry, and for developing spatial data structures.

- BSP is a generic process of partitioning a space into two parts at each step.
- It is typically implemented as a binary tree where each node represents a subregion of space partitioned by a hyperplane.
- **Pros:**
  - ⋆ Variable cell shape, which allows for efficient space division according to the distribution of objects.
- **Cons:**
  - ⋆ Computation of separating hyperplanes can be costly.
- **Limitations:**
  - ⋆ Not suitable for high-dimensional data due to the curse of dimensionality.

21.6.7 Locality-sensitive hashing

Locality-Sensitive Hashing (LSH) is an algorithmic technique that hashes similar input items into the same “buckets” with high probability. The buckets are used to partition the space in a way that preserves locality.

- The goal is to maximize the probability that similar items map to the same bucket.
- LSH is used for approximate nearest neighbor search in high-dimensional spaces.

Designing a Locality-Sensitive Hash Function

- The hash function $h\colon \mathbb{R}^m \to \mathbb{R}$ should ensure that if two points $\mathbf{x}, \mathbf{y}$ are close in the input space, their hash values $h(\mathbf{x}), h(\mathbf{y})$ should be close as well.
- It is often used in conjunction with other data structures to handle high-dimensional data efficiently.

22 Lecture 22

22.1 Parallel Computation

22.2 Paralleism that does not require programmer intervention

22.2.1 Pipelines

- CPU pipelines can be viewed as implementing some form of parallelism in the sense that multiple executions are being executed simultaneously.
- For example, one instruction's arithmetic is performed (in an ALU) while the next is being decoded.
- However, from the programmer's perspective, everything must happen as if there was no parallelism at all.

22.2.2 Multitasking

- Multitasking allows multiple executables to run "simultaneously" (even on a single processor)
- Regularly, the **scheduler** (part of the OS kernel) decides which task gets to run on a processor.

22.3 Multitasking on a single-core processor

- Multitasking on a single-core processor allows multiple tasks to be handled seemingly in parallel by quickly switching between them. This is known as context switching.
- A single-core CPU can only execute one task at a time. However, it gives the illusion of parallelism by rapidly alternating between tasks, which is managed by the operating system's scheduler.
- Tasks can be in different states, such as *running*, where a task is actively using the CPU, or *sleeping*, where a task is inactive and waiting for some event or the passage of a certain amount of time.
- When multitasking, the scheduler decides which task should be run next. This decision is based on various factors such as task priority, fairness, I/O requirements, and more.
- The scheduler uses a context switch to save the state of the current running task and load the state of the next task to run. The state of a task includes information like the program counter, registers, and memory allocation.
- This rapid switching is fast enough that users perceive multiple applications are running simultaneously, even though only one is being processed at any instant.
- The scheduler is called:

- ⋆ at regular intervals f times per second, by default:
    - ∗ Linux: $f = 1000$ Hz (see CONFIG_HZ)
    - ∗ macOS: $f = 100$ Hz (see sysctl kern.clockrate)
    - ∗ Windows 10: $f = 64$ Hz (see timeBeginPeriod())
- ⋆ when an task performs a system call (`open()`, `write()`, `exit()`, ...)
- ⋆ when a "hardware interrupt" happens:
    - ∗ keyboard received a keypress
    - ∗ network device received data
    - ∗ storage device finished writing
    - ∗ sound/video device ready to receive next buffer
    - ∗ ...

22.3.1 Preemptive multitasking

- When the scheduler decides to interrupt a running process (e.g. to run another)
    - ⋆ the process is said to "preempted"
    - ⋆ it becomes "runnable"
- When a process executes a system call,
    - ⋆ it starts "sleeping"
    - ⋆ after the requested operation is performed,
        - ∗ in some cases, it will run again
        - ∗ in other cases, it becomes runnable and will only run when a CPU is available
    - ⋆ many system calls can take a long time to perform ("blocking" system calls):
        - ∗ `read()`, `write()`, `recv()`, `send()`
- At any given time, most tasks are sleeping
    - ⋆ waiting for data (e.g. from network)
    - ⋆ waiting for user interaction (e.g. keyboard or touch input)
    - ⋆ waiting on a timer (tasks that run at regular interval)
- The only tasks that are normally running/runnable are those performing CPU-intensive operations
    - ⋆ graphics rendering
    - ⋆ audio/video/data compression and decompression
    - ⋆ computations
    - ⋆ etc.

22.3.2 Multitasking on a multi-core processor

- Multitasking on a multi-core processor refers to the ability of the CPU to perform multiple tasks at the same time by utilizing multiple cores within a single CPU unit.
- Each core can independently run a task, allowing for real parallel execution of processes. This improves the overall efficiency and performance of the system, especially for multi-threaded applications.
- In the illustrated example, five tasks (task 0 to task 4) are distributed across four CPU cores (CPU 0 to CPU 3). This distribution allows each task to execute in parallel:
  - ⋆ Task 0 runs on CPU 0
  - ⋆ Task 1 runs on CPU 1
  - ⋆ Task 2 runs on CPU 2
  - ⋆ Task 3 runs on CPU 3
  - ⋆ Task 4 could either be queued or run concurrently if one of the other tasks completes or is in a waiting state (not shown in the images).
- This parallelism enables a multi-core processor to handle more tasks in the same amount of time compared to a single-core processor, which must switch contexts to give the illusion of parallelism.
- From a hardware perspective:
  - ⋆ A CPU corresponds to a single integrated circuit ("IC") package
  - ⋆ A computer can (rarely) have multiple CPUs
    - ∗ Typically only found in datacenters, rarely more than 2
  - ⋆ Each CPU can have multiple *cores*
    - ∗ generally 2-8 cores on laptops
    - ∗ up to 128 on datacenter CPUs
- From a software perspective:
  - ⋆ Everything that can run a task is generally called a "CPU"
  - ⋆ Only the kernel's scheduler will (sometimes) care about CPU vs. core
  - ⋆ All other software is unaware of the difference
- A CPU can have multiple copies of some logic blocks
- very common for arithmetic and logic units (ALUs)

22.4 Simultaneous Multithreading SMT

- From a hardware perspective:
  - ⋆ With Simultaneous Multithreading (SMT) (a.k.a. Hyperthreading),
  - ⋆ each core can run multiple (generally 2) tasks (“threads”)
  - ⋆ but they share many logic blocks (in particular ALUs)
  - ⋆ SMT *works well* when those logic blocks would otherwise be idle
  - ⋆ SMT is *ineffective* when those logic blocks are the bottleneck
- From a software perspective:
  - ⋆ Everything that can run a task is generally called a “CPU”
  - ⋆ Only the kernel’s scheduler will (sometimes) care about CPU vs. core vs. *thread*
  - ⋆ All other software is unaware of the difference
  - ⋆ “Thread” has a different meaning in software

22.5 SIMD

- SIMD stands for Single Instruction Multiple Data
- new, larger registers (in addition to the general purpose ones): “vector registers”

bits	255..224	223..192	191..160	159..128	127..96	95..64	63..32	31..0
256	ymm0							
64	fp64 #3		fp64 #2		fp64 #1		fp64 #0	
32	fp32 #7	fp32 #6	fp32 #5	fp32 #4	fp32 #3	fp32 #2	fp32 #1	fp32 #0
16								
8								

- but
  - ⋆ SIMD registers cannot be treated as big integers
  - ⋆ individual “lanes” (8-, 16-, 32- or 64-bit parts) generally cannot be accessed individually

22.6 SIMD registers

- On Intel (and AMD) ISAs:
  - ⋆ SSE (1999): 8 128-bit registers xmm0 - xmm7
  - ⋆ AVX (2011): 16 256-bit registers ymm0 - ymm15
  - ⋆ AVX-512 (2016, but not yet generally available): 32 512-bit registers zmm0 - zmm31
- On ARM:
  - ⋆ Neon (2005): 16 128-bit registers Q0 - Q15

22.7 Example

The code example illustrates the concept of SIMD (Single Instruction, Multiple Data) in the context of vectorized operations for performance optimization in computing.

- SIMD is a parallel computing architecture that allows a single processor instruction to perform the same operation on multiple data points simultaneously.
- In the given C function `add_one`, each element of a 4-element float array is incremented by one. This operation, although simple, can be vectorized to exploit data-level parallelism provided by SIMD.
- The corresponding SIMD-enabled assembly instructions perform the addition in a parallel fashion:
    - ⋆ The `vbroadcastss` instruction replicates a single float value into all four elements of a SIMD register (`xmm0`), preparing the value `1.0` for parallel operations.
    - ⋆ The `vaddps` instruction performs parallel addition of the four `1.0` values in `xmm0` to the four elements of the input array pointed to by `rdi`, storing the result back into `xmm0`.
    - ⋆ The `vmovups` instruction writes the result from the SIMD register back to the memory location of the array, completing the parallel increment operation.
- This SIMD approach is beneficial for computational efficiency, as it minimizes the number of sequential operations and leverages the processor's ability to handle multiple data points in a single instruction cycle.

22.8 Counter-example

The counter-example demonstrates a scenario where SIMD (Single Instruction, Multiple Data) instructions are not sufficient to handle all operations in a vectorized manner due to the dependency of each operation on the result of the previous one.

- In the C function `many_ops`, different arithmetic operations are applied to the elements of a float array, with each operation depending on the result of the previous one, which introduces data dependencies.
- The assembly code shows a sequence of SIMD instructions corresponding to the C code operations. However, due to the dependencies, each operation must be completed before the next can begin, preventing the full utilization of SIMD parallelism.
- The instructions include `vaddss` and `vdivss` for addition and division on scalar single-precision floating point values, respectively. These operations require the results from previous instructions, hence they cannot be parallelized.
- This example illustrates that while SIMD is powerful for operations that can be performed in parallel without dependencies, it is not universally applicable to all types of data processing, especially those with a sequence of dependent operations.

22.9 How to use SIMD

- Rely on compilers (*"autovectorization"*)
- Write assembly code
- Use compiler *"intrinsics"*
  - ⋆ Intrinsics look like C functions
  - ⋆ but the compiler knows how to translate them to specific assembly code
  - ⋆ **Intel intrinsics guide**
  - ⋆ **ARM intrinsics**

$\Longrightarrow$ refer to the intrinsics guide

22.10 Thread-level concurrency

22.11 Processes and threads

- When the OS runs an executable, it gets its own *process*
- A single executable (if run multiple times) can have multiple independent processes
- Memory is virtualized: each process has its own view of the memory it owns
- A process can create ("spawn") multiple *threads*
- Like processes, each thread is an individual task from the point of view of the scheduler
- Within a process, threads share a same view of the process memory
- Pro: Communication between threads is extremely efficient
  - ⋆ Just write something to memory,
  - ⋆ let other threads read it through the same pointer
- Con: Because memory is shared, synchronizing threads is **very complex**

22.11.1 Wrong code (1)

The code attempts to implement a single-item buffer with:

- A `push` function that stores a value in the buffer and sets a `ready` flag to indicate the buffer is full.
- A `pop` function that retrieves a value from the buffer and clears the `ready` flag to indicate the buffer is empty.

The `ready` flag is used for synchronization, ensuring each item is only pushed when the buffer is empty and popped when full.
The C compiler is free to reorder this:

```
void push(int value)
{
    while (ready == 1) {
        // wait
    }
    buffer = value;
    ready = 1;
}
```

into this:

```
void push(int value)
{
    buffer = value;
    while (ready == 1) {
        // wait
    }
    ready = 1;
}
```

The C compiler is free to infer that this loop:

```
while (ready == 1) {
    // wait
}
```

has either zero or infinitely many iterations without side effects (UB); thus remove the loop!

22.11.2 Wrong code (2)

The code shows a flawed implementation for synchronizing a single-producer, single-consumer scenario using a shared buffer:

- The `push` function writes to the buffer if it's not marked as ready, and sets it as ready afterward.
- The `pop` function reads from the buffer if it's marked as ready, and resets the flag afterward.
- The use of `volatile` suggests an attempt to prevent compiler optimization from reordering the access to the `ready` flag.
- However, this code is incorrect due to race conditions caused by lack of atomicity in flag checks and updates, potentially leading to simultaneous access to the buffer by both `push` and `pop`.

22.11.3 Solution

- low-level: compiler intrinsics for "atomic" operations: combined operations that are performed as a single unit no thread will every see the memory in an intermediate state
- high-level: use libraries that correctly implement some primitives: locks, queues, etc.
  - ⋆ Posix threads ("pthreads"; Linux, MacOS)
  - ⋆ OpenMP (Open Multi-Processing; portable)

22.12 Distributed Computing

- In distributed computing, processes do not share memory
- They must communicate by explicitly sending data to each other (`send()`, `recv()`, etc.) typically over the network
- **Con:** Communication is much slower than multithreading
- **Pros:**
  - ⋆ Easier to implement and reason about
  - ⋆ Scales to higher levels of parallelism
    - ∗ As of today, off-the-shelf computers can have up to 2 processors $\times$ 128 cores $\times$ 2 SMT threads = 512 concurrent software threads
    - ∗ With distributed computing, networked computers can work together in parallel
- **Libraries:**
  - ⋆ Message Passing Interface (MPI)
  - ⋆ ...

22.13 4. Hardware Acceleration

22.13.1 Graphics processing units (GPUs)

- GPUs were designed to perform the same simple, repetitive operations
  - ⋆ on many pixels ("fragment shaders"), or
  - ⋆ on many 3D coordinates ("vertex shaders")

22.13.2 Examples (GLSL)

```
float box(in vec2 st, in vec2 size){
    size = vec2(0.5) - size*0.5;
    vec2 uv = smoothstep(size,
                         size+vec2(0.01),
                         st);
    uv *= smoothstep(size,
                     size+vec2(0.01),
                     vec2(1.0)-st);
    return uv.x*uv.y;
}
```

22.13.3 Examples (CUDA)

```
inline __device__ float3 roundAndExpand(float3 v, ushort *w) {
    v.x = rintf(__saturatef(v.x) * 31.0f);
    v.y = rintf(__saturatef(v.y) * 63.0f);
    v.z = rintf(__saturatef(v.z) * 31.0f);

    *w = ((ushort)(v.x) << 11) | ((ushort)(v.y << 5)) | (ushort)v.z;
    // approximate integer bit expansion.
    v.x = 0.032258064516f;
    v.y = 0.015873015873f;
    v.z = 0.032258064516f;
    return v;
}
```

- GPUs were designed to perform the same simple, repetitive operations
    - on many pixels (”fragment shaders”), or
    - on many 3D coordinates (”vertex shaders”)
- they generally adopt a SIMT (”single instruction, multiple threads”) model
    - hundreds of threads working on different sets of data
    - but running the exact same instructions
- good fit for long loops performing repetitive operations
- bad fit for if/then/else

22.13.4 How do we use GPUs?

- GPUs are programmed in special-purpose languages
- Typically, all GPU code is compiled
    - during application startup,
    - by the device driver

- for the specific GPU device installed (amount and subdivision of threads, memory, etc.)

- Two dominant players in the GPU market: nVidia and AMD
- Three major GPU programming languages:
  - CUDA (nVidia, proprietary),
  - ROCm (AMD, open-source),
  - OpenCL (cross-platform, open-source)

22.13.5 Matrix Multiplication Tutorial

22.13.5.1 N = 8192

8192 × 8192 matrix multiplication
precision: fp64 ("double")
CPU: AMD Ryzen 7900 x3d

matmul_1	straightforward implementation	2932.059 s	1x	
matmul_2	transpose B matrix	357.569 s	8x	
matmul_3	block multiply	67.105 s	44x	
matmul_4	same code as matmul_3, SIMD	32.876 s	89x	
matmul_5	OpenBLAS	15.555 s	188x	1x
matmul_6	OpenBLAS, 24 threads	1.962 s	1494x	8x

22.13.5.2 N = 32768

32768 × 32768 matrix multiplication
precision: fp32 ("float") - total 4 GB per matrix
CPU Ryzen 7900 x3d (released Feb 2023)

matmul_7	OpenBLAS, 1 thread	550.350 s	1x
matmul_8	OpenBLAS, 24 threads	50.577 s	11x
matmul_9	cuBLAS, *nVidia* A100 (Apr 2021)	13.152 s	42x
matmul_9	cuBLAS, *nVidia* H100 (Mar 2022)	? s	84x?