Introduction to Programming

https://www.unive.it/data/course/493929

2. Basics

Giulio Ermanno Pibiri — giulioermanno.pibiri@unive.it
Department of Environmental Sciences, Informatics and Statistics

Academic year 2023/2024

Overview

- Anatomy of a C program
- Tokens: keywords, identifiers, operators, separators, literals
- Arithmetic, relational, logical operators
- Precedence and associativity
- Binary representation and data types
- Variables and expressions
- Basic input/output
- Exercises

Anatomy of a C program

```c
#include <stdio.h>

/* Your first program in C. */

int main() {

    printf("Hello world!\n");

    return 0;
}
```

Anatomy of a C program

library header file with definition of `printf`

```c
#include <stdio.h>

/* Your first program in C. */

int main() {

    printf("Hello world!\n");

    return 0;
}
```

Anatomy of a C program

```c
#include <stdio.h>

/* Your first program in C. */

int main() {

    printf("Hello world!\n");

    return 0;
}
```

library header file with definition of `printf`

multi-line comment (ignored during compilation)

Anatomy of a C program

library header file with definition of `printf`

multi-line comment (ignored during compilation)

entry point of every C program: your program starts executing from here

```c
#include <stdio.h>

/* Your first program in C. */

int main() {

    printf("Hello world!\n");

    return 0;
}
```

Anatomy of a C program

```c
#include <stdio.h>

/* Your first program in C. */

int main() {

    printf("Hello world!\n");

    return 0;
}
```

library header file with definition of `printf`

multi-line comment (ignored during compilation)

entry point of every C program: your program starts executing from here

this function prints a single line `"Hello world!"` and move cursor on next line (`'\n'`)

Anatomy of a C program

```c
#include <stdio.h>

/* Your first program in C. */

int main() {
    printf("Hello world!\n");

    return 0;
}
```

library header file with definition of `printf`

multi-line comment (ignored during compilation)

entry point of every C program: your program starts executing from here

this function prints a single line `"Hello world!"` and move cursor on next line (`'\n'`)

note that `main()` returns a value of type `int`: we return `0` to mean "computation ends correctly"

Tokens in C

- Tokens are the smallest units of every C program.
- They define **what** you can write in your program.
- There are **5 types** of tokens in C.
    - **Keywords**
    - **Identifiers**
    - **Operators**
    - **Separators**
    - **Literals**

Example:

```c
int x = y + 3;
```

Tokens in C

- Tokens are the smallest units of every C program.
- They define **what** you can write in your program.
- There are **5 types** of tokens in C.
  - **Keywords**
  - **Identifiers**
  - **Operators**
  - **Separators**
  - **Literals**

Example:

```c
int x = y + 3;
```

keyword

Tokens in C

- Tokens are the smallest units of every C program.
- They define **what** you can write in your program.
- There are **5 types** of tokens in C.
    - **Keywords**
    - **Identifiers**
    - **Operators**
    - **Separators**
    - **Literals**

Example:

```
int x = y + 3;
```

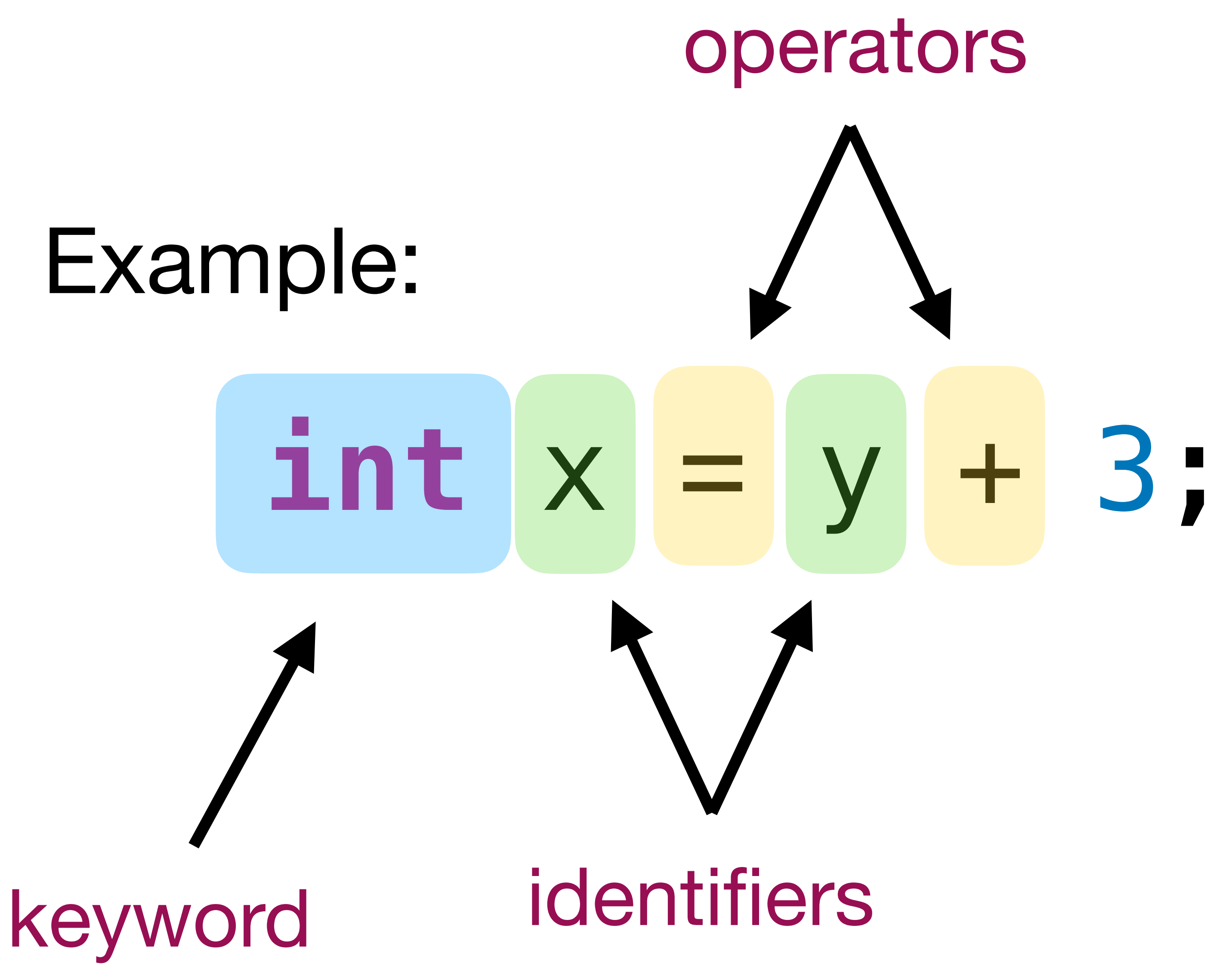
keyword

identifiers

Tokens in C

- Tokens are the smallest units of every C program.
- They define **what** you can write in your program.
- There are **5 types** of tokens in C.
  - **Keywords**
  - **Identifiers**
  - **Operators**
  - **Separators**
  - **Literals**

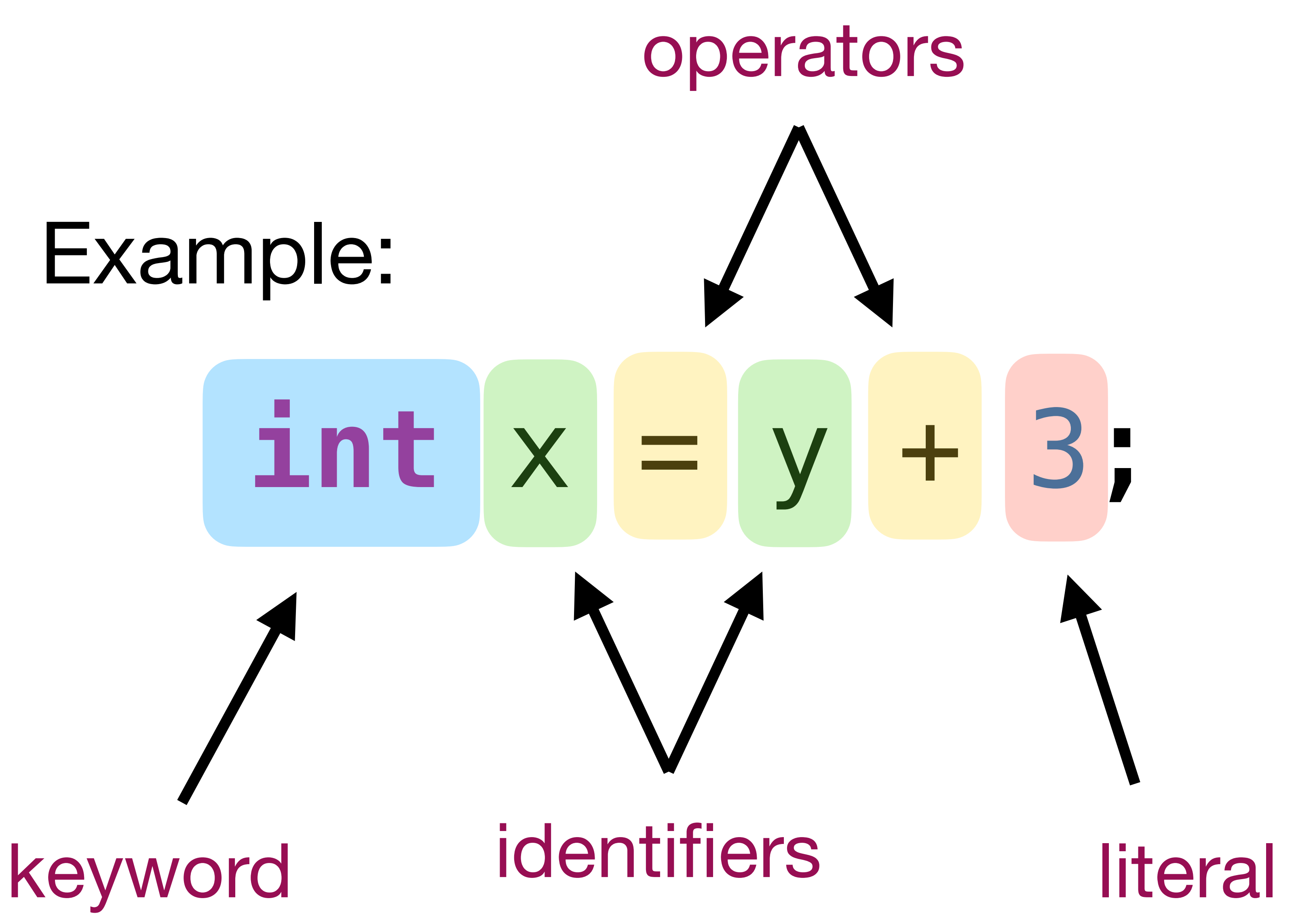
Example:



Tokens in C

- Tokens are the smallest units of every C program.
- They define **what** you can write in your program.
- There are **5 types** of tokens in C.
    - **Keywords**
    - **Identifiers**
    - **Operators**
    - **Separators**
    - **Literals**

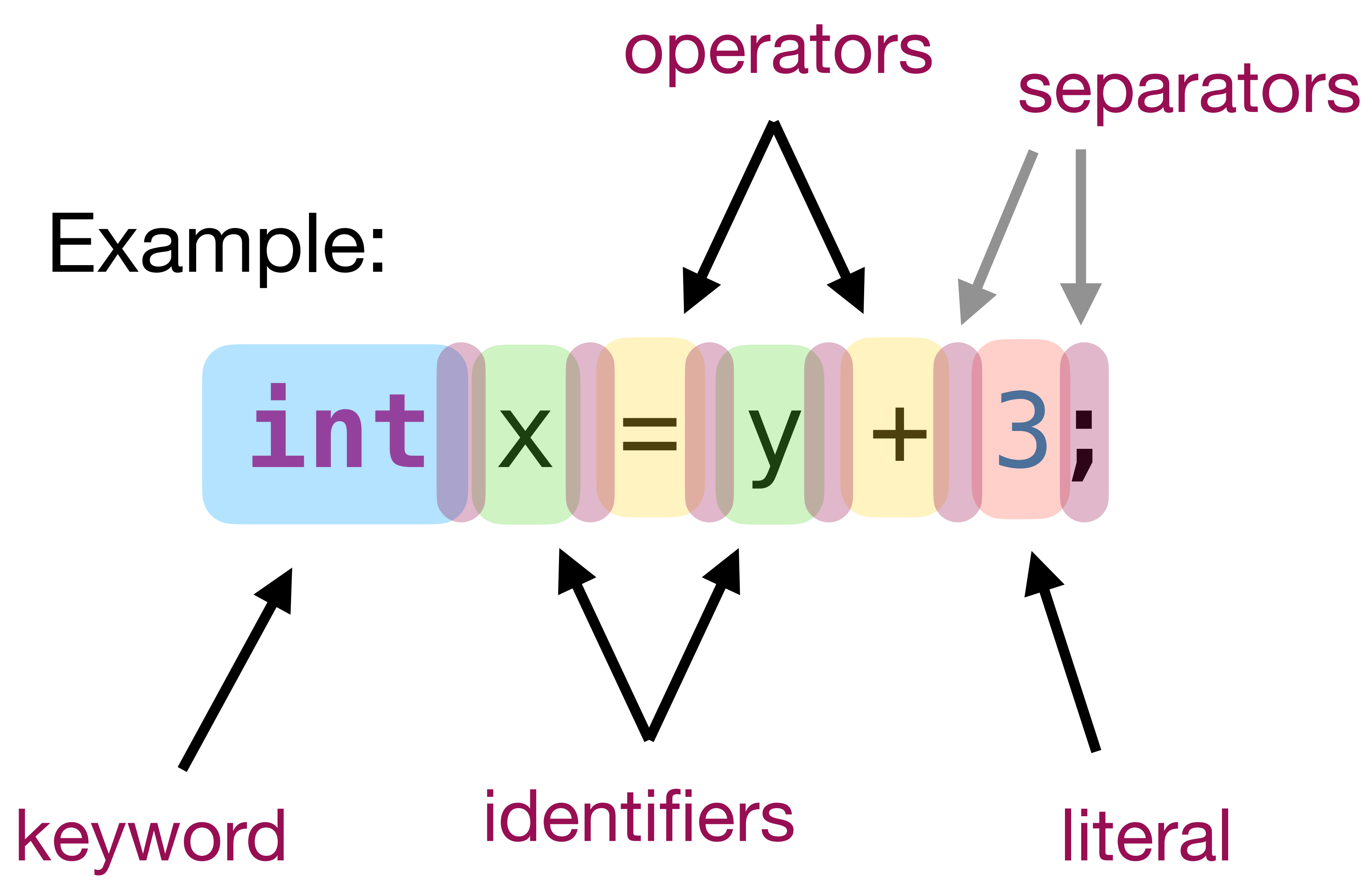
Example:



Tokens in C

- Tokens are the smallest units of every C program.
- They define **what** you can write in your program.
- There are **5 types** of tokens in C.
  - **Keywords**
  - **Identifiers**
  - **Operators**
  - **Separators**
  - **Literals**

Example:



Keywords

- Keywords are **reserved words** in the C language.
- You **cannot** use them to define identifiers.

`auto`	`break`	`case`	`char`	`const`	`continue`	`default`	`do`
`double`	`else`	`enum`	`extern`	`float`	`for`	`goto`	`if`
`int`	`long`	`register`	`return`	`short`	`signed`	`sizeof`	`static`
`struct`	`switch`	`typedef`	`union`	`unsigned`	`void`	`volatile`	`while`

Identifiers

- Identifiers are **names chosen by the programmer** to name variables, functions, or user-defined structures.

- Examples:

  ```
  float foo = 1.00; double PI = 3.14;
  ```

- Rules:

  - an identifier cannot be a keyword;

  - an identifier must begin with a character, `a-z` or `A-Z`, or underscore `_` ;

  - an identifier must not contain any other special character, like + or `]` or `!`, etc.;

  - **case matters**: `foo`, `FOO`, `Foo`, and `FOo` are four **different** identifiers in C.

Operators

- Operators are tokens that are used to perform an **operation**.
- This operation can be:
  - **arithmetic**, e.g., sum or multiply two values;
  - **logical**, e.g., determine the truth value of a composition of expressions (see next);
  - **relational**, e.g., determine if `a < b`;
  - **bitwise**. (We will not cover them.)

Arithmetic operators

- Suppose that `a = 10` and `b = 5`.

`+`	Add two numbers.	`a + b` gives `15`
`*`	Multiply two numbers.	`a * b` gives `50`
`/`	Divide two numbers. **(Beware: integer division might not be what you expect.)**	`a / b` gives `2`
`%`	Modulus operator divide first operand from second and returns remainder.	`a % b` gives `0`

Assignment operator

- The assignment operator = will assign the value of the right-hand side (a constant, an expression, or a variable) to the left-hand side which must be the name of a **variable**.
- Example: `a = 10` assigns the constant `10` to the variable `a`.
- Example: `x = a + 10` assigns the result of the expression `(a + 10)` to the variable `x`. (In this case, the result is `20`.)
- Short-hand assignments.

	Example	Meaning
`+=`	`a += 10`	`a = a + 10`
`-=`	`a -= 10`	`a = a - 10`
`*=`	`a *= 10`	`a = a * 10`
`/=`	`a /= 10`	`a = a / 10`

Relational operators

- A **boolean expression** is an expression that returns either **true** or **false**.
- Example: let `a = 10` and `b = 5`. Then `(a > 5)`, `(a == b)`, and `(b != 3)` are boolean expressions returning, respectively, `true`, `false`, and `true`.
- **Very important:**
  - **In C, the value 0 means false and any value different than 0 means true.**
  - **In C, the value of a boolean expression is converted to the integer 0 if it is false or to the integer 1 if it is true.**
- We will re-consider this in Part 3.

Relational operators

- Relational operators are used to determine the relation between two operands.

- Assume again that `a = 10` and `b = 5`.

>	If value of left operand is greater than right operand, returns true else returns false	`(a > b)` returns **1**
<	If value of right operand is greater than left operand, returns true else returns false	`(a < b)` returns **0**
==	If both operands are equal returns true else false	`(a == b)` returns **0**
!=	If both operands are not equal returns true else false.	`(a != b)` returns **1**
>=	If value of left operand is greater-than or equal to right operand, returns true else false	`(a >= b)` returns **1**
<=	If value of right operand is greater or equal to left operand, returns true else false	`(a <= b)` returns **0**

Beware of the bugs...

- **Important note**: do not confuse == (relational equality) with = (assignment) !
- **In C, every expression returns a value. Also assignments.**
- Hence, `a = 5` assigns `5` to `a` and the whole expression returns `5`.
- `a == 5` determines if the value of `a` is equal to `5`, hence it returns either `0` or `1`.
- **Q.** Why can this be a problem for you?

Beware of the bugs...

- **Important note**: do not confuse == (relational equality) with = (assignment) !
- **In C, every expression returns a value. Also assignments.**
- Hence, `a = 5` assigns `5` to `a` and the whole expression returns `5`.
- `a == 5` determines if the value of `a` is equal to `5`, hence it returns either `0` or `1`.
- **Q.** Why can this be a problem for you?
- **A.** Because in C, you can write

  `if (a = 5) { ... }` and `if (a == 5) { ... }`

  and **the code compiles!**

  (The compiler will warn you, though. Stay tuned, we will reconsider this in Part 3.)

Logical operators

- Logical operators are used to combine the results of two or more boolean expressions together. Assume again that `a = 10` and `b = 5`.

<table>
<tr><td><code>&&</code></td><td>Used to combine two expressions. If both operands are true or non-zero, returns true else false</td><td><code>((a>=1)&&(a<=10))</code>
returns 1</td></tr>
<tr><td><code>||</code></td><td>If any of the operand is true or non-zero, returns true else false</td><td><code>((a>1)||(a<5))</code>
returns 1</td></tr>
<tr><td><code>!</code></td><td>Logical NOT operator is a unary operator. Returns the complement of the boolean value.</td><td><code>!(a>1)</code> returns 0</td></tr>
</table>

- This will be useful in combination with the `if-and-else` construct (Part 3).

Increment/decrement operator

- The increment/decrement operator is a **unary** operator that increases/decreases an integer-type variable value by **1**.
- It comes in two flavours: **prefix** and **postfix**.
- ++a and −−a are prefix operators; a++ and a−− are postfix operators.

Increment/decrement operator

- The increment/decrement operator is a **unary** operator that increases/decreases an integer-type variable value by **1**.
- It comes in two flavours: **prefix** and **postfix**.
- ++a and −−a are prefix operators; a++ and a−− are postfix operators.
- Assume `a = 5`:

```
b = ++a; // a becomes 6 and b is assigned 6

b = a++; // b is assigned 5 and a becomes 6
```

- More formally: prefix operators **first evaluate and then return** the result; postfix operator **first return and value of the variable and then perform** increment/decrement.

Conditional operator

- Also called "ternary" operator because it accepts three arguments.
- Syntax of using ternary operator: `(condition) ? (true part) : (false part)`
- Example, assuming `a = 10` and `b = 5`:

  ```
  b = (a > 1) ? a : b;
  ```

  will store the value `10` in `b` because `(a > 1)` is true, hence assigning the value of `a` to `b`.
- This can also be achieved using an `if-and-else` statement.

Other operators

- There are also other operators that will be introduced in **next lectures**.

`.`	**Member access** operator	Used to access the members of structures and unions
`->`	**Member access** operator	Used to access the members of structures and unions
`*`	**Dereferencing** operator	Used to dereference the value of a pointer variable
`&`	**Address** of operator	Used to get the actual memory address of a variable

Separators

- There are two types of separators in C: **whitespace** characters (single/multiple spaces or TABs), and semicolons `';'` .

- Whitespace characters are used to separate keywords and identifiers. You can use any number of them (not necessarily one).

Separators

- There are two types of separators in C: **whitespace** characters (single/multiple spaces or TABs), and semicolons `';'` .
- Whitespace characters are used to separate keywords and identifiers. You can use any number of them (not necessarily one).
- For example, these two programs are identical for the compiler.

```c
#include <stdio.h>

/* Your first program in C. */

int main()
{
    printf("Hello world!\n");

    return 0;
}
```

```c
#include <stdio.h> /* Your first program in C. */
int main(){printf("Hello world!\n");return 0;}
```

Separators

- There are two types of separators in C: **whitespace** characters (single/multiple spaces or TABs), and semicolons `';'` .
- Whitespace characters are used to separate keywords and identifiers. You can use any number of them (not necessarily one).
- For example, these two programs are identical for the compiler.

```
#include <stdio.h>

/* Your first program in C. */

int main()
{
    printf("Hello world!\n");

    return 0;
}
```

```
#include <stdio.h> /* Your first program in C. */
int main(){printf("Hello world!\n");return 0;}
```

But please, use a decent/proper indentation style!

Separators

- Semicolons, instead, are used to separate different statements.
- Example:

```
int a = 5 float b = 3.14
printf("\n")
```

does not compile!

Separators

- Semicolons, instead, are used to separate different statements.
- Example:

  ```
  int a = 5 float b = 3.14
  printf("\n")
  ```

  does not compile!

- Separate each statements with semicolons.

  ```
  int a = 5; float b = 3.14;
  printf("\n");
  ```

  Now ok!

Operator precedence

- **Q.** What is the value of this expression: `3 + 5 * 10 / 2` ?

Operator precedence

- **Q.** What is the value of this expression: `3 + 5 * 10 / 2` ?
- **A.** The result can be ambiguous if we do not follow a precise rule.
- We need to know what operators have **precedence** over others.

Operator precedence

- **Q.** What is the value of this expression: `3 + 5 * 10 / 2` ?
- **A.** The result can be ambiguous if we do not follow a precise rule.
- We need to know what operators have **precedence** over others.
- From math, for example, we know that `*` and `/` have higher precedence compared to + (and –).
- So the result of the above expression is `28`, not `40` or something else.

Operator precedence

- **Q.** What is the value of this expression: `3 + 5 * 10 / 2` ?
- **A.** The result can be ambiguous if we do not follow a precise rule.
- We need to know what operators have **precedence** over others.
- From math, for example, we know that `*` and `/` have higher precedence compared to + (and –).
- So the result of the above expression is `28`, not `40` or something else.
- **We can always enforce precedence using parentheses** `(...)`. Always do so if you do not remember operator precedence!

Operator associativity

- If an expression contains two or more operators with the **same precedence**, then we have to fix an order of evaluation:

    either **left-to-right** or **right-to-left**.

- For example, `int x = 5 * 4 / 2 * 3;` is evaluated from **left to right**.

- But `x = y = z;` is evaluated from **right to left** instead.

- Always consult a table like the one here

    https://en.cppreference.com/w/c/language/operator_precedence.

Literals

- **Constant values** in a C program are known as literals.
- There are four different types of literals:
  - **integer**, such as `7244ul`, `-2345L`;
  - **float**, such as `0.314`, `.456`, `-5.3e-11`;
  - **character**, such as `'a'`, `'b'`, `'c'`, ... , `'\n'`, etc. (any character enclosed between two **single** quotes);
  - **string**, any sequence of characters enclosed between **double** quotes, such as `"hello"` or `"welcome to Introduction to Programming :)\n”`.

Literals

- **Constant values** in a C program are known as literals.
- There are four different types of literals:
  - **integer**, such as `7244ul`, `−2345L`;
  - **float**, such as `0.314`, `.456`, `−5.3e−11`;
  - **character**, such as `'a'`, `'b'`, `'c'`, ... , `'\n'`, etc. (any character enclosed between two **single** quotes);
  - **string**, any sequence of characters enclosed between **double** quotes, such as `"hello"` or `"welcome to Introduction to Programming :)\n"`.
- **Note**: in C, a string always terminates with the special character `'\0'` which is used to indicate the **end** of the string. We will see again this in Part 6. Hence `'A'` and `"A"` are two different things!

Comments

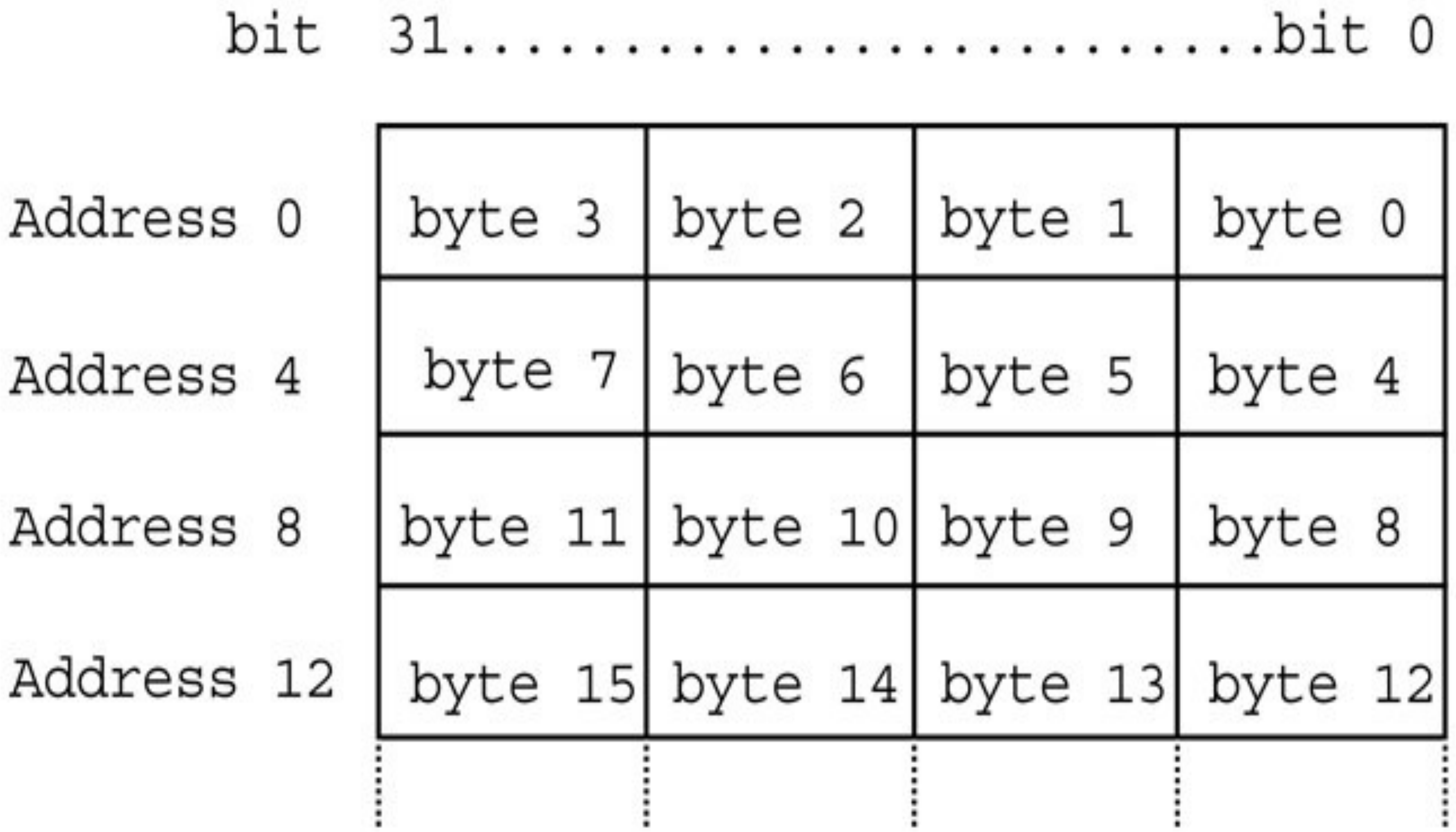
- We have **single-line** comments and **multi-line** comments.
- Comments are ignored by the compiler but make the code more readable. A good comment **should explain** what a certain piece of code does.

```c
#include <stdio.h>

int main() {
    int b = 13;  // base of the rectangle
    int h = 4;   // height of the rectangle

    /*
        Compute perimeter and area of a rectangle
        whose base is b and hight is h.
    */
    int perimeter = 2 * (b + h);
    int area = b * h;

    /* Print results. */
    printf("Perimeter is %d and area is %d.\n", perimeter, area);

    return 0;  // return with success
}
```

Representation of numbers and text

Binary numbers and computers

- Computers have storage units called binary digits or **bits**.
  - High voltage: bit `1`
  - Low voltage: bit `0`
- A group of 8 consecutive bits is called a **byte**.
- The computer memory stores digital information as multiple of bytes.



Different binary units

Unit	Value	Storage
Bit	1 or 0	
Byte (B)	8 Bits	Character
Kilobyte (KB)	1024 Bytes = 2^{10} Bytes	Half page of text
Megabyte (MB)	1024 Kilobytes = 2^{20} Bytes	About 2 mins MP3 file
Gigabyte (GB)	1024 Megabytes = 2^{30} Bytes	About one hour Movie
Terabyte (TB)	1024 Gigabytes = 2^{40} Bytes	128 DVD Movies
Petabyte (PB)	1024 Terabyte = 2^{50} Bytes	7 billion Facebook photos
Exabyte (EB)	1024 Petabyte = 2^{60} Bytes	50,000 years of DVD
Zettabyte (ZB)	1024 Exabyte = 2^{70} Bytes	Global internet traffic per year (2016)

Note. In general, kilo = 1000 = 10^3, but in Computer Science: kilo (K) = 1024 = $2^{10} \approx 10^3$ = 1000.

Binary representation

- How many things can n bits represent?
- With 1 bit: 2^1 things.
- By doubling the bits, that is, with 2 bits: $2^2 = 4$ things.

  ...

- In general, with n bits we can thus represent: 2^n things.

1 Bit	2 Bits	3 Bits	4 Bits	5 Bits
0	00	000	0000	00000
1	01	001	0001	00001
	10	010	0010	00010
	11	011	0011	00011
		100	0100	00100
		101	0101	00101
		110	0110	00110
		111	0111	00111
			1000	01000
			1001	01001
			1010	01010
			1011	01011
			1100	01100
			1101	01101
			1110	01110
			1111	01111
				10000
				10001
				10010
				10011
				10100
				10101
				10110
				10111
				11000
				11001
				11010
				11011
				11100
				11101
				11110
				11111

Example to represent 7 things (days of the week)

- To represent 7 distinct objects we need at least 3 bits because with 3 bits we can represent up to 2^3 = 8 distinct objects, so 3 is the minimum number of bits we can use.

- In general we need $\lceil \log_2 n \rceil$ bits per object for n distinct objects.

`000`	Mon
`001`	Tue
`010`	Wed
`011`	Thu
`100`	Fri
`101`	Sat
`110`	Sun
`111`	Non used

Representing text

- Text is written in letters, and the Latin alphabet we use is made of 26 letters.
- **Q.** How many bits would you need?
- We can list them all and assign to each a binary string.
- Character set: a list of characters and the codes used to represent each one that computer manufacturers agreed to standardize.
- **ASCII** character set:
  - American Standard Code for Information Interchange
  - 7 bits version allows 128 unique characters
  - See UTF-8 as more modern standard character encoding

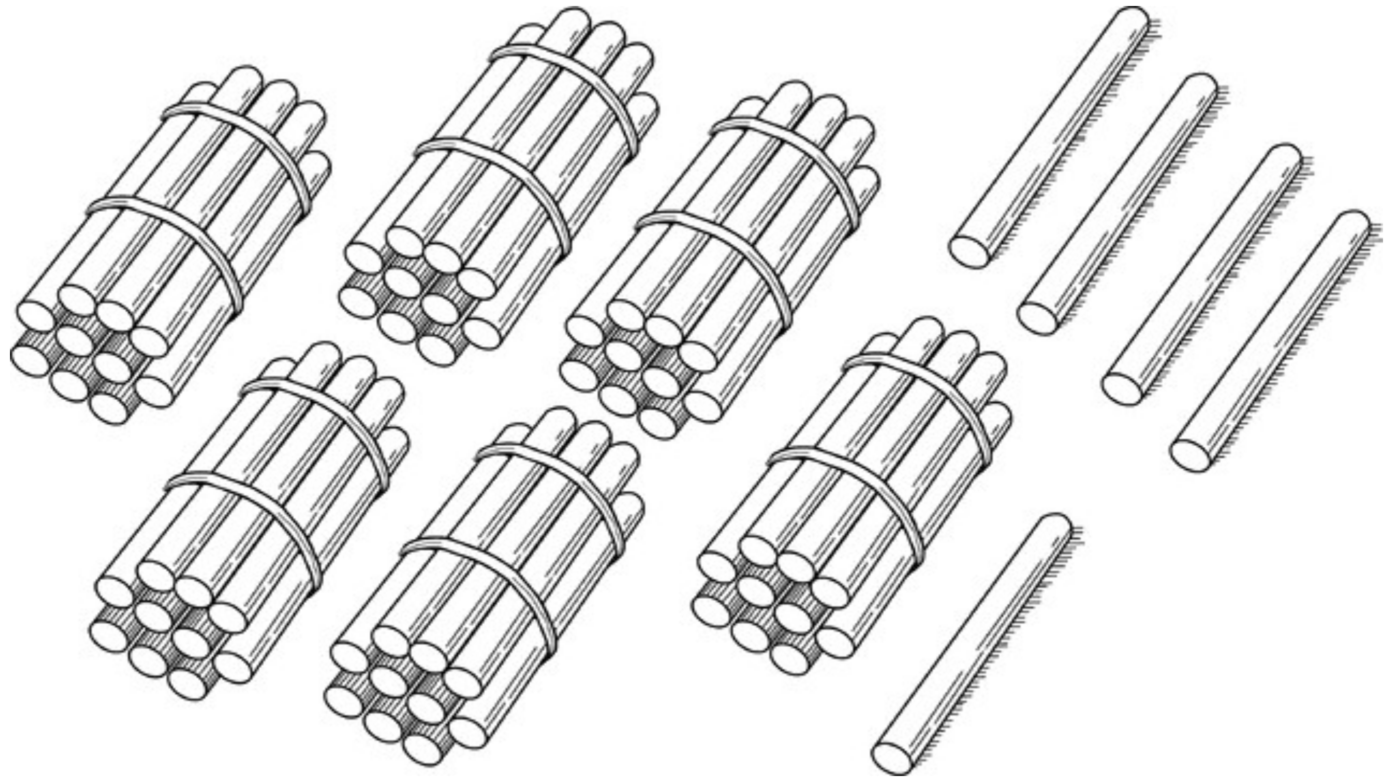
ASCII TABLE

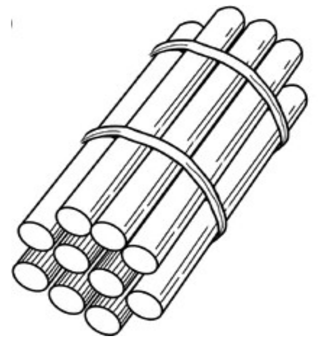
Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Representing integers: positional notation

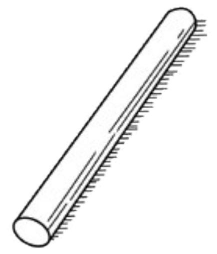
- Positional notation is used to represent numbers in everyday life.

- Example: **65**



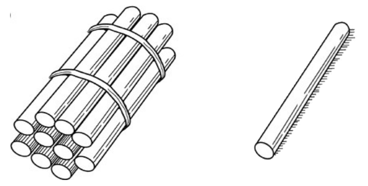


6 x



5 x

- **Note.** 10 is the **base** of this representation, which needs **10** symbols (0,1,2,3,4,5,6,7,8,9), whose value depends on the **position**.



6	5

$$65 = 5 \cdot 10^0 + 6 \cdot 10^1$$

Positional notation: general case

- You use an alphabet of b symbols.
- Each symbol d_i or digit is such that $0 \leq d_i < b$.
- A number x seen as a n-digit vector: $d_{n-1} \ \dots \ d_1 d_0$.
- The **value** of x in this base system is a linear combination of the powers of b:

$$x = \sum_{i=0}^{n-1} d_i \cdot b^i = d_0 \cdot b^0 + d_1 \cdot b^1 + \ \cdots \ + d_{n-1} \cdot b^{n-1}$$

- Example: $x = 647 = 6 \cdot 10^2 + 4 \cdot 10^1 + 7 \cdot 10^0$ with $b = 10$ and $n = 3$ $(d_0 = 7,\ d_1 = 4,\ d_2 = 6)$.

Binary integer numbers

- **Decimal** numbers have base 10 and need 10 symbols:
  `0, 1, 2, 3, 4, 5, 6, 7, 8, 9`
- **Hexadecimal** numbers have base 16 and need 16 symbols:
  `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F`
- **Binary** numbers have base 2 and only need 2 symbols:
  `0, 1`
- The space of representable integer numbers does **not** depend on the base, you can convert any number from a base to another.

Converting from binary to decimal

- **Q.** Given the binary number $[x]_2 = 1101110$, how do we convert it to decimal $[x]_{10}$?

Converting from binary to decimal

- **Q.** Given the binary number $[x]_2 = 1101110$, how do we convert it to decimal $[x]_{10}$?

- **A.** Just apply the formula! In this case $b = 2$, so each digit $0 \leq d_i < 2$ (or, $d_i \in \{0,1\}$).

Converting from binary to decimal

- **Q.** Given the binary number $[x]_2 = 1101110$, how do we convert it to decimal $[x]_{10}$?

- **A.** Just apply the formula! In this case $b = 2$, so each digit $0 \leq d_i < 2$ (or, $d_i \in \{0,1\}$).

$x =$	1	1	0	1	1	1	0	← digits
	6	5	4	3	2	1	0	← positions

Converting from binary to decimal

- **Q.** Given the binary number $[x]_2 = 1101110$, how do we convert it to decimal $[x]_{10}$?

- **A.** Just apply the formula! In this case $b = 2$, so each digit $0 \leq d_i < 2$ (or, $d_i \in \{0,1\}$).

$x =$	1	1	0	1	1	1	0	$\longleftarrow$	digits
	6	5	4	3	2	1	0	$\longleftarrow$	positions

$$
\begin{aligned}
[x]_{10} &= 0 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5 + 1 \cdot 2^6 \\
&= 2 + 4 + 8 + 32 + 64 = 110
\end{aligned}
$$

Converting from decimal to binary

- **Q.** Given the decimal number $[x]_{10} = 134$, how do we convert it to binary $[x]_2$?
- **A.** Divide x by 2 until we obtain a quotient (q) of 0. At each step i we obtain the digit d_i as the reminder of $\lfloor x/2 \rfloor$.

  (Since we divide by 2, the reminder d_i is either `0` or 1).

Converting from decimal to binary

Example for $x = 134$.

Converting from decimal to binary

Example for $x = 134$.

$i = 0 : q = \lfloor 134/2 \rfloor = 67 \ (d_0 = 0)$

Converting from decimal to binary

Example for $x = 134$.

$$i = 0 : q = \lfloor 134/2 \rfloor = 67 \ (d_0 = 0)$$

$$i = 1 : q = \lfloor 67/2 \rfloor = 33 \ (d_1 = 1)$$

Converting from decimal to binary

Example for $x = 134$.

$$i = 0 : q = \lfloor 134/2 \rfloor = 67 \ (d_0 = 0)$$

$$i = 1 : q = \lfloor 67/2 \rfloor = 33 \ (d_1 = 1)$$

$$i = 2 : q = \lfloor 33/2 \rfloor = 16 \ (d_2 = 1)$$

Converting from decimal to binary

Example for $x = 134$.

$$i = 0 : q = \lfloor 134/2 \rfloor = 67 \ (d_0 = 0)$$
$$i = 1 : q = \lfloor 67/2 \rfloor = 33 \ (d_1 = 1)$$
$$i = 2 : q = \lfloor 33/2 \rfloor = 16 \ (d_2 = 1)$$
$$i = 3 : q = \lfloor 16/2 \rfloor = 8 \ (d_3 = 0)$$

Converting from decimal to binary

Example for $x = 134$.

$$i = 0 : q = \lfloor 134/2 \rfloor = 67 \ (d_0 = 0)$$

$$i = 1 : q = \lfloor 67/2 \rfloor = 33 \ (d_1 = 1)$$

$$i = 2 : q = \lfloor 33/2 \rfloor = 16 \ (d_2 = 1)$$

$$i = 3 : q = \lfloor 16/2 \rfloor = 8 \ (d_3 = 0)$$

$$i = 4 : q = \lfloor 8/2 \rfloor = 4 \ (d_4 = 0)$$

Converting from decimal to binary

Example for $x = 134$.

$$i = 0 : q = \lfloor 134/2 \rfloor = 67 \ (d_0 = 0)$$

$$i = 1 : q = \lfloor 67/2 \rfloor = 33 \ (d_1 = 1)$$

$$i = 2 : q = \lfloor 33/2 \rfloor = 16 \ (d_2 = 1)$$

$$i = 3 : q = \lfloor 16/2 \rfloor = 8 \ (d_3 = 0)$$

$$i = 4 : q = \lfloor 8/2 \rfloor = 4 \ (d_4 = 0)$$

$$i = 5 : q = \lfloor 4/2 \rfloor = 2 \ (d_5 = 0)$$

Converting from decimal to binary

Example for $x = 134$.

$$i = 0 : q = \lfloor 134/2 \rfloor = 67 \ (d_0 = 0)$$

$$i = 1 : q = \lfloor 67/2 \rfloor = 33 \ (d_1 = 1)$$

$$i = 2 : q = \lfloor 33/2 \rfloor = 16 \ (d_2 = 1)$$

$$i = 3 : q = \lfloor 16/2 \rfloor = 8 \ (d_3 = 0)$$

$$i = 4 : q = \lfloor 8/2 \rfloor = 4 \ (d_4 = 0)$$

$$i = 5 : q = \lfloor 4/2 \rfloor = 2 \ (d_5 = 0)$$

$$i = 6 : q = \lfloor 2/2 \rfloor = 1 \ (d_6 = 0)$$

Converting from decimal to binary

Example for $x = 134$.

$i = 0 : q = \lfloor 134/2 \rfloor = 67 \ (d_0 = 0)$

$i = 1 : q = \lfloor 67/2 \rfloor = 33 \ (d_1 = 1)$

$i = 2 : q = \lfloor 33/2 \rfloor = 16 \ (d_2 = 1)$

$i = 3 : q = \lfloor 16/2 \rfloor = 8 \ (d_3 = 0)$

$i = 4 : q = \lfloor 8/2 \rfloor = 4 \ (d_4 = 0)$

$i = 5 : q = \lfloor 4/2 \rfloor = 2 \ (d_5 = 0)$

$i = 6 : q = \lfloor 2/2 \rfloor = 1 \ (d_6 = 0)$

$i = 7 : q = \lfloor 1/2 \rfloor = 0 \ (d_7 = 1)$

Converting from decimal to binary

Example for $x = 134$.

$$i = 0 : q = \lfloor 134/2 \rfloor = 67 \ (d_0 = 0)$$
$$i = 1 : q = \lfloor 67/2 \rfloor = 33 \ (d_1 = 1)$$
$$i = 2 : q = \lfloor 33/2 \rfloor = 16 \ (d_2 = 1)$$
$$i = 3 : q = \lfloor 16/2 \rfloor = 8 \ (d_3 = 0)$$
$$i = 4 : q = \lfloor 8/2 \rfloor = 4 \ (d_4 = 0)$$
$$i = 5 : q = \lfloor 4/2 \rfloor = 2 \ (d_5 = 0)$$
$$i = 6 : q = \lfloor 2/2 \rfloor = 1 \ (d_6 = 0)$$
$$i = 7 : q = \lfloor 1/2 \rfloor = 0 \ (d_7 = 1)$$

$$\longrightarrow \quad [134]_2 = d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0 = 10000110$$

Converting from decimal to binary - alternative method

How to store the number 57 with 8 bits?

We need to find the *0/1* coefficients of $2^0, 2^1, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7$ so that their sum is 57.

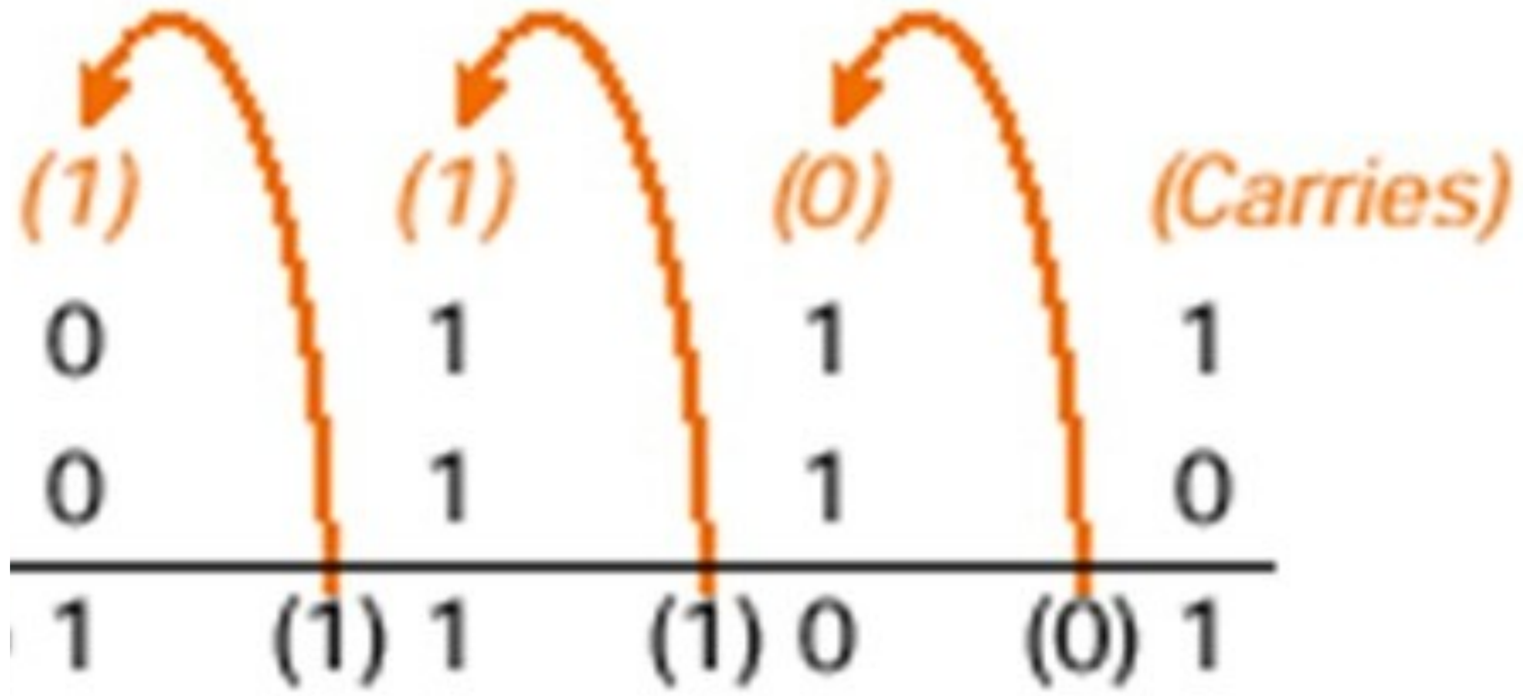
Subtractions method:

Input	Powers of 2	Is larger than or equal to 2^i?	Coefficient
57	$2^7 = 128$	$57 - 128 < 0$	0
57	$2^6 = 64$	$57 - 64 < 0$	0
57	$2^5 = 32$	$57 - 32 = 25 \geq 0$	1
25	$2^4 = 16$	$25 - 16 = 9 \geq 0$	1
9	$2^3 = 8$	$9 - 8 = 1 \geq 0$	1
1	$2^2 = 4$	$1 - 4 < 0$	0
1	$2^1 = 2$	$1 - 2 < 0$	0
1	$2^0 = 1$	$1 - 1 = 0 \geq 0$	1

The binary representation of 57 is **00111001**.

Binary arithmetic

- Remember that there are only 2 digit symbols in binary, `0` and `1`. Therefore `1 + 1 = 0` with a *carry.*
- This is analogous to decimal arithmetic: 5 + 5 = 0 with a carry.
- Example of 7 (`111`) + 6 (`110`) = 13 (`1101`):



Integer numbers in code

- An integer number is generally represented in a 4-byte word (32 bits), in which the first bit represents the sign, and the rest represents the number.

- So we have 31 bits for the number: 2^{31} possible values, from 0 to $2^{31} - 1$.

- **Problem:** we end up in representing the 0 value twice! And we want to represent 0 only once.

x	Unsigned value	Signed value
`000`	0	+0
`001`	1	+1
`010`	2	+2
`011`	3	+3
`100`	4	-0
`101`	5	-1
`110`	6	-2
`111`	7	-3

Example with $n = 3$ bits.

2's complement representation

- Given a number whose sign bit is 1: the 2's complement of x with n bits is $u(x) - 2^n$, where $u(x)$ indicates the unsigned value of x.
- So the range of an integer represented with n bits is: $-2^{n-1} \leq x < 2^{n-1}$.
- That is: $-2^2 \leq x < 2^2$ for $n = 3$ or $-2^{31} \leq x < 2^{31}$ for $n = 32$.

x	Unsigned value	Signed value	2's complement
`000`	0	+0	+0
`001`	1	+1	+1
`010`	2	+2	+2
`011`	3	+3	+3
`100`	4	-0	-4
`101`	5	-1	-3
`110`	6	-2	-2
`111`	7	-3	-1

Example with $n = 3$ bits.

Real numbers to binary

- Using a byte, we can use a simple approach (fixed-point representation): 4 bits for the integer part, and 4 bits to represent the fractional part
  $xxxx\,.\,yyyy$

- We can represent only $16 = 2^4$ numbers < 1:
  `0.0000`, `0.0001`, `0.0010`, `0.0011`, …, `0.1110`, `0.1111`

- The smallest number we can represent is `0000.0001`, i.e.,
  $2^{-4} = 1/16 = 0.0625$

- We are wasting many bits to represent useless numbers (like the zeros). Anyway recall that real numbers are infinite while bit configurations are finite.

Real numbers to binary

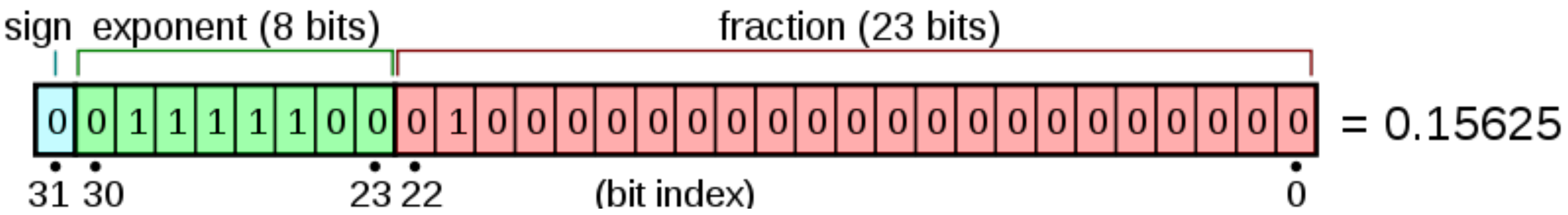
- Scientific notation allows us to express any real number as:

$$\pm d\,.\,dddddd \times 10^n.$$

- Using computers: single precision (32 bits) floating-point representation.
    - **1 bit** for the **sign**
    - **8 bits** for the **exponent** (note: that all 0s and all 1s are reserved for *zero* and *NaN* — not a number — so you can represent 254 numbers: exponent ranges from -126 to 127)
    - **23 bits** for the *significand*

Real numbers to binary

- The encoding of float numbers is a bit tricky:



- And the conversion to decimal is given by:

$(0.15625)_{10} = (0.00101)_2$
(1/8+1/32) shifted by 3 positions

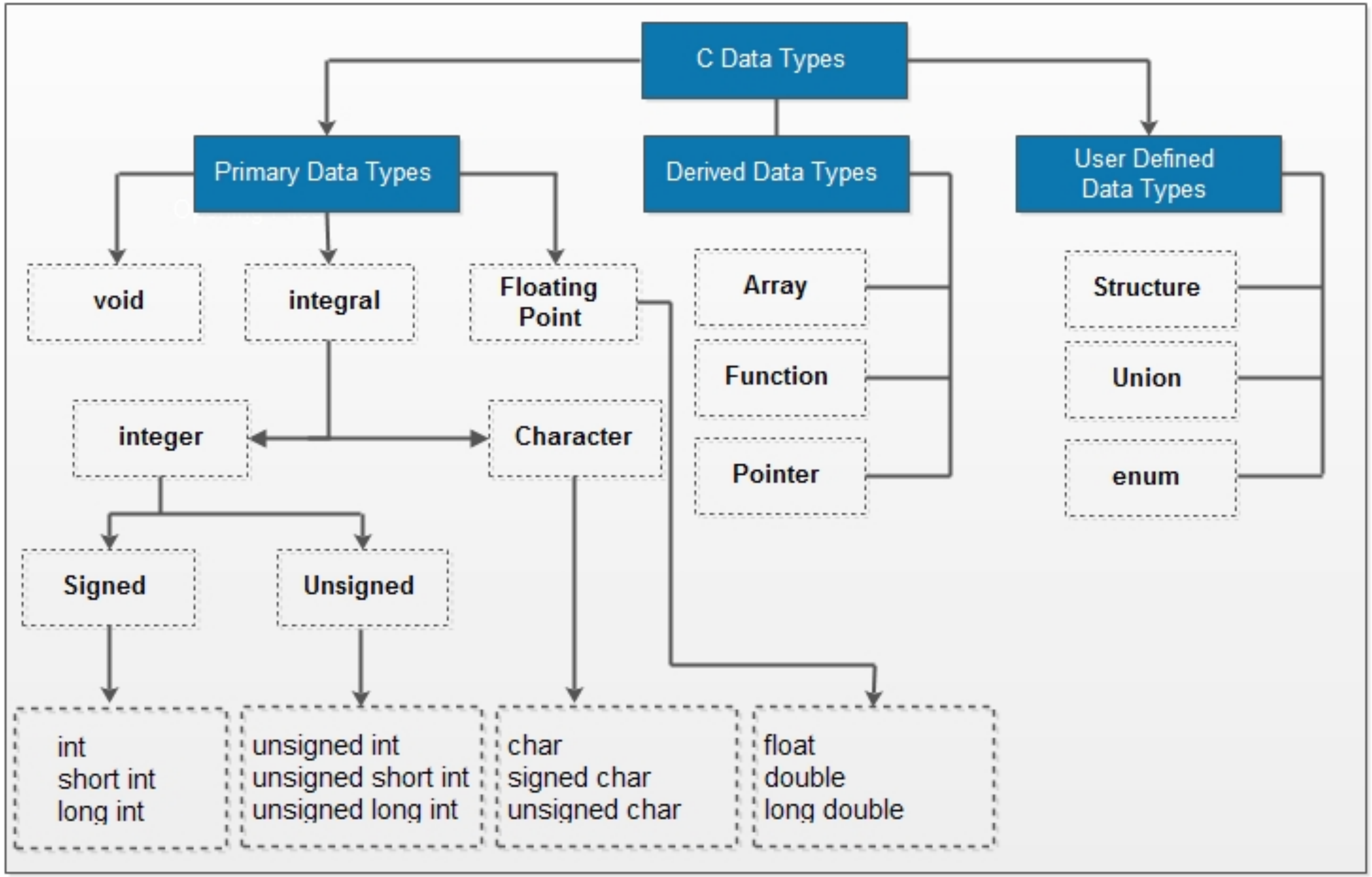
$$(-1)^{b_{31}} \times 2^{(b_{30}b_{29}\ldots b_{23})_2 - 127} \times (1.b_{22}b_{21}\ldots b_0)_2$$

- So the largest modulo you can represent is (roughly) $2^{127} \times 2^0 \approx 3.402 \times 10^{38}$
- And the smallest positive number is $2^{-126} \times 2^{-23} = 2^{-149} \approx 1.401 \times 10^{-45}$.
- Recall: real numbers are continuous. You can not represent every number in the range from largest to smallest, there is an infinite number of them.
- **Note**: changing the base changes the space of representable numbers:
  - You can not represent 1/3 in base 10 (need infinite digits)
  - You can trivially represent it in base 3: `0.1` $(0 \cdot 3^1 + 1 \cdot 3^{-1})$

Data types in C

https://ecomputernotes.com/what-is-c/types-and-variables/data-types-in-c

- A data type defines:
  - how data is organised in memory,
  - the number of bytes it takes,
  - and the range of values a variable of a given type is allowed to take.



Data types in C

Data type	Size	Range
`char`	1 byte	-128 to 127
`signed char`		
`unsigned char`	1 byte	0 to 255
`short`	2 bytes	–32,767 to 32,767
`signed short`		
`signed short int`		
`unsigned short`	2 bytes	0 to 65,535
`unsigned short int`		
`int`	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
`signed int`		
`unsigned int`	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
`long`	4 bytes	-2,147,483,648 to 2,147,483,647
`signed long`		
`signed long int`		
`unsigned long`	4 bytes	0 to 4,294,967,295
`unsigned long int`		
`long long`	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`long long int`		
`signed long long`		
`signed long long int`		
`unsigned long long`	8 bytes	0 to 18,446,744,073,709,551,615
`unsigned long long int`		
`float`	4 bytes	1.2E-38 to 3.4E+38
`double`	8 bytes	2.3E-308 to 1.7E+308
`long double`	12 bytes	3.4E-4932 to 1.1E+4932

`sizeof` operator

- The operator sizeof is a special operator in C that returns the exact **number of bytes** taken by a type in memory. It always returns an integer (`long unsigned` on my Mac).

```c
#include <stdio.h>

int main() {
    printf("sizeof(char) = %lu\n\n", sizeof(char));

    printf("sizeof(short) = %lu\n", sizeof(short));
    printf("sizeof(int) = %lu\n", sizeof(int));
    printf("sizeof(long) = %lu\n", sizeof(long));
    printf("sizeof(long long) = %lu\n\n", sizeof(long long));

    printf("sizeof(float) = %lu\n", sizeof(float));
    printf("sizeof(double) = %lu\n", sizeof(double));
    printf("sizeof(long double) = %lu\n", sizeof(long double));

    return 0;
}
```

Variables and expressions

Variables

- Variables are **names given to memory addresses** that hold our data in memory. Remember: C is a high-level programming language, so we refer to a quantity in memory via a symbolic name.

Computer		Programmers		
Address	**Content**	**Name**	**Type**	**Value**
90000000	00	sum	int (4 bytes)	000000FF(255_{10})
90000001	00			
90000002	00			
90000003	FF			
90000004	FF	age	short (2 bytes)	FFFF(-1_{10})
90000005	FF			
90000006	1F	averge	double (8 bytes)	1FFFFFFFFFFFFFFF ($4.45015E-308_{10}$)
90000007	FF			
90000008	FF			
90000009	FF			
9000000A	FF			
9000000B	FF			
9000000C	FF			
9000000D	FF			

https://www3.ntu.edu.sg/home/ehchua/programming/cpp/cp4_PointerReference.html

Variables

- In C, all variables must be declared specifying a **type**. (Technically, we say that C is a strongly-types programming language.)
- The variable name is an identifier and, as such, must follow the naming rules for identifiers (see previous slides).
- Two more rules:
  - You cannot have two variables with the same name in the same **scope.** We will cover scopes in Part 4. For the moment being, you cannot declare two variables with the same name in the `main()`.
  - A variable cannot have the same name of a global identifier.

Variables

- **Recommendations**:
  - Always choose **semantic names** for variables, not `foo`, `bar`, `pluto`, `paperino`, etc.
  - Example: if you are using a variable to hold the temperature of a room, then use something like `float room_temp = 23.4;`

Variables

- **Recommendations**:
  - Always choose **semantic names** for variables, not `foo`, `bar`, `pluto`, `paperino`, etc.
  - Example: if you are using a variable to hold the temperature of a room, then use something like `float room_temp = 23.4;`
  - Always **remember to initialise** your variables! Bugs due to un-initialised variables are very common and lead to unexpected results...but are easy to fix.

Expressions

- An **expression** in C is a combination of variables, operators, and literals.
  - Examples:

```
a + 15;

(a*a + b*b) / 2 + 1;

(a < b) + 3;
```

- **Important**: Every expression returns a value of a given **type**.
- So, the type of the expression `a + 15;` depends on the type of the variable `a`.
  (More on this later.)

Constants

- **Constants are values that cannot change.** We can have a constant of any type.
- In particular, a constant behaves like a normal variable but its value cannot change.
- There are two ways we can have constants in C:
  - using the **`const`** qualifier;
  - using the **`#define`** directive.
- Examples:

```
const float PI = 3.14159;

#define PI 3.14159
```

Typecasting

- Typecasting is the process of **converting the type of a variable or of an expression to another**.

- **Q.** Why might this be needed?

Typecasting

- Typecasting is the process of **converting the type of a variable or of an expression to another**.

- **Q.** Why might this be needed?

- Consider the problem of computing the average between two or more `int` numbers.

```
int x, y, z;
x = 23;
y = 45;
z = 66;
float avg = (x + y + z) / 3;
```

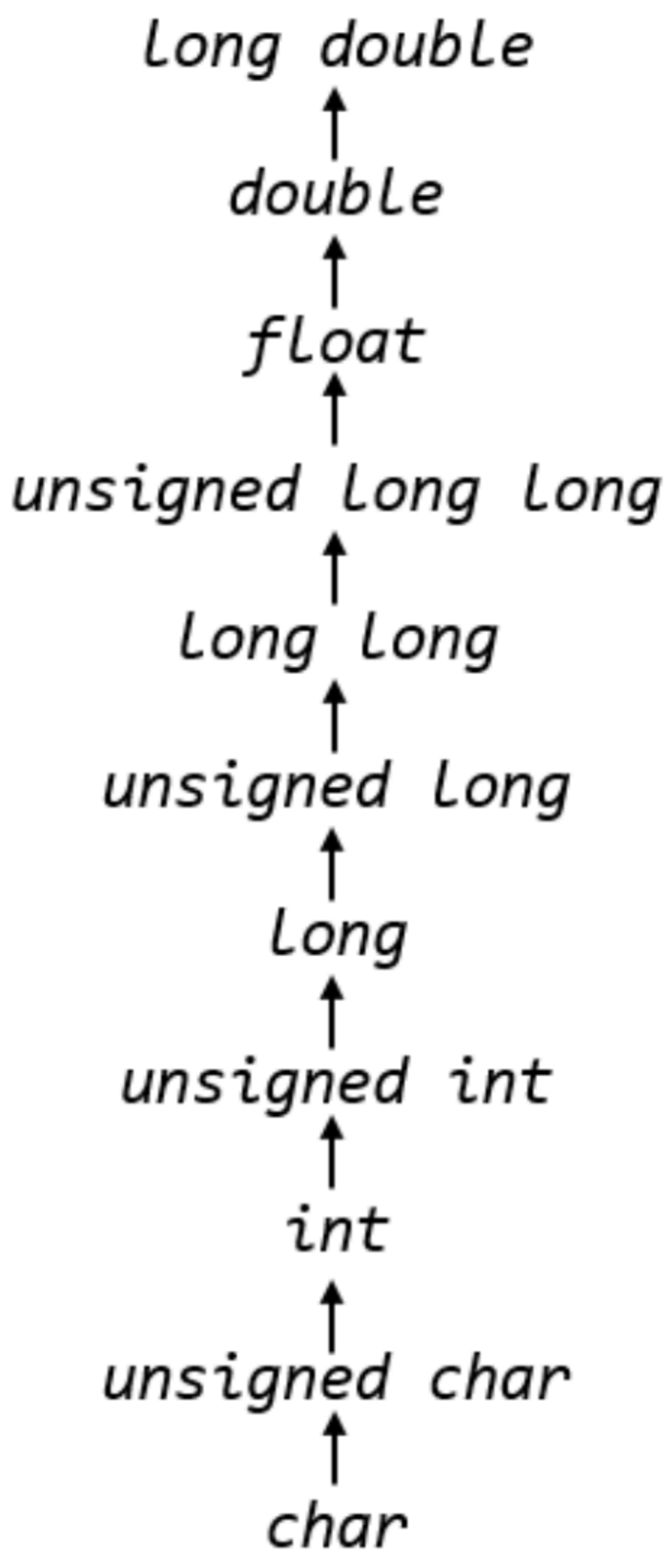
- Easy, no? Let's try it.

Typecasting

- We can have **explicit** and **implicit** typecasting.
- Explicit is when we manually perform typecasting with

  `(new-type)(expression)`

  Implicit is when an **implicit conversion** happens.
- Reference: https://en.cppreference.com/w/c/language/conversion
- Let's exploit implicit conversion rules in the previous example for computing the average.



implicit conversion

Basic input/output

Basic input/output

- The library **`stdio`** of C has several functions to deal with input/output (I/O) operations.
- To use them, we need to write

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

  at the beginning of our program.
- These functions are
  - `getchar()`
  - `putchar()`
  - `scanf()`
  - `printf()`

Basic input/output

- `getchar()` and `putchar()` consume and print respectively a single `char`.
- Let's consider this minimal example.

```
char c = getchar();
putchar(c);
```

- `scanf()` and `printf()` are more powerful instead, and can read and print respectively many values at a time (see next).

Basic input/output

- Wait, but the standard (https://en.cppreference.com/w/cpp/io/c/putchar) says:

Defined in header `<cstdio>`

```
int putchar( int ch );
```

Writes a character ch to `stdout`. Internally, the character is converted to `unsigned char` just before being written.

- **Q.** Why does `putchar` accepts an `int` instead of a `char`?

Basic input/output

- Wait, but the standard (https://en.cppreference.com/w/cpp/io/c/putchar) says:

Defined in header `<cstdio>`

```
int putchar( int ch );
```

Writes a character ch to `stdout`. Internally, the character is converted to `unsigned char` just before being written.

- **Q.** Why does `putchar` accepts an `int` instead of a `char`?

- **A.** Remember the correspondence between `int`s and `char`s given by the **ASCII** table!

`scanf`

- The function `scanf` has the following signature:

  `scanf("list-of-format-specifiers", list-of-memory-addresses);`

  where `"list-of-format-specifiers"` is a list of strings, each starting with `'%'` and specifying the type of the variable to be read (technically: how the bytes should be interpreted).

- We have seen many format specifiers a few slides ago. Most popular are `"%d"` for `int`, `"%c"` for `char`, `"%lu"` for `long unsigned`, `"%s"` for strings (`char*`).

- Let's consider some examples.

```
scanf("%d %f %c", &myint, &myfloat, &mychar);
```

- The operator `'&'` takes the memory address of a variable. Function `scanf` needs the memory address of a variable to fill the variable with the user input. This will become clearer when we will talk about pointers in C (Part 5).

printf

- The signature of `printf` is

  `printf("string containing format-specifiers", list-of-variables);`

- Example.

```
printf("myint is %d, myfloat is %f, and mychar is %c\n",
        myint, myfloat, mychar);
```

- **Q.** We do not need to pass to `printf` the memory addresses of the variables. Why?

Format specifiers (again)

- These are special strings that are used in `printf()` and `scanf()` to, respectively, print and read variables of the wanted type.

Format specifier	Description	Supported data types
`%c`	Character	`char`
`%d`	Signed Integer	`short`
`%e %E`	Scientific notation of float	`float`
`%f`	Floating point	`float`
`%g %G`	Similar as %e or %E	`float`
`%hi`	Signed Integer (short)	`short`
`%hu`	Unsigned Integer (short)	`unsigned short`
`%i`	Signed Integer	`short`
`%l %ld %li`	Signed Integer	`long`
`%lf`	Floating point	`double`
`%Lf`	Floating point	`long double`
`%lu`	Unsigned integer	`unsigned int`
`%lli %lld`	Signed Integer	`long long`
`%llu`	Unsigned Integer	`unsigned long long`
`%o`	Octal representation of Int	`short`
`%p`	Pointer	`void*`
`%s`	String	`char*`
`%u`	Unsigned Integer	`unsigned int`
`%x %X`	Hexadecimal	`short`
`%n`	Prints nothing	

Exercises

Exercises

1. Write a C program to perform input/output of all basic data types.
2. Write a C program to enter two numbers and find their sum.
3. Write a C program to enter two numbers and perform all arithmetic operations.
4. Write a C program to enter base and height of a rectangle and find its perimeter.
5. Write a C program to enter base and height of a rectangle and find its area.
6. Write a C program to enter radius of a circle and find its diameter, circumference, and area.

Exercises

7. Write a C program to enter a length in centimeters and convert it into meters and kilometers.

8. Write a C program to enter a temperature in Celsius and convert it to Fahrenheit.
   (Conversion formula: ˚F = ˚C $*$ 1.8 + 32.)

9. Write a C program to enter a temperature in Fahrenheit and convert it to Celsius.
   (Conversion formula: ˚C = (˚F − 32) / 1.8.)

10. Write a C program to convert a number of days into years, months, and days.

11. Write a C program that, given two integers `x` and `y`, computes power x^y.
    (Hint: use the function `pow` of the library `math.h`.)

12. Write a C program to enter any number and calculate its square root.
    (Hint: use the function `sqrt` of the library `math.h`.)

Exercises

13. Write a C program to enter two angles of a triangle and find the third angle.

14. Write a C program to enter base and height of a triangle and find its area.

15. Write a C program to enter base and height of a right triangle and find its *hypotenuse*. (Pitagora's theorem.)

16. Write a C program to calculate area of an equilateral triangle.

17. Write a C program to enter marks of five subjects and calculate total, average and percentage.

18. Write a C program to enter principal (P), interest rate (r), and terms of loan in years (n) and calculate **simple** and **compound** Interest. (https://www.investopedia.com/terms/s/simple_interest.asp)