CS100 Lecture 2

Variables I and Arithmetic Types

Contents

Variable declaration

Type of a variable

Every variable in C has a type.

- The type is **fully deterministic** and **cannot be changed**.
- The type is **known even when the program is not run**.
  - $\Leftrightarrow$ The type is known at **compile-time**.
  - $\Leftrightarrow$ C is **statically-typed** [1]. $\Leftrightarrow$ C has a **static type system**.
  - In contrast, Python is **dynamically-typed**.

Statically-typed vs dynamically-typed

C: statically-typed

```c
int a = 42;   // Type of a is int.
a = "hello"; // Error! Types mismatch!
```

The type of a variable

- is explicitly written on declaration, and
- is known at compile-time, and
- cannot be changed.

Python: dynamically typed

```python
a = 42         # Type of a is int.
a = "hello"    # Type of a becomes str.
```

The type of a variable

- can be changed, and
- is not necessarily known until we run the program.

A type-related error in C is *(usually)* a **compile error**:

- It stops the compiler. The executable will not be generated.

Declare a variable

To declare a variable, we need to specify its **type** and **name**.

```
Type name;
```

Example:

```
int x;    // Declares a variable named `x`, whose type is `int`.
double y; // Declares a variable named `y`, whose type is `double`.
```

We may declare multiple variables of a same type in one declaration statement, separated by `,`:

```
int x, y; // Declares two variables `x` and `y`, both having type `int`.
```

Declare a variable

A **variable declaration** can be placed

- inside a function, which declares a **local variable**, or
- outside of any functions, which declares a **global variable**.

```c
#include <stdio.h>

int main(void) {
  // local variables in `main`
  int x, y;
  scanf("%d%d", &x, &y);
  printf("%d\n", x + y);
}
```

```c
#include <stdio.h>

int x, y; // global variables

int main(void) {
  scanf("%d%d", &x, &y);
  printf("%d\n", x + y);
}
```

Local variables vs global variables

Which one do you prefer?

```c
#include <stdio.h>

int main(void) {
  // local variables in `main`
  int x, y;
  scanf("%d%d", &x, &y);
  printf("%d\n", x + y);
}
```

```c
#include <stdio.h>

int x, y; // global variables

int main(void) {
  scanf("%d%d", &x, &y);
  printf("%d\n", x + y);
}
```

What are these variables used for?

```c
#include <stdio.h>
// Other #includes

int x, y; // What are these two variables used for?

int moveSpaceShuttle(SpaceShuttle *shuttle, Coord to, Vehicle *by) {
  // 109 lines
}
int makePreparations(Environment *env, Task tasks[], Time time) {
  // 73 lines
}
LaunchResult launchSpaceShuttle(SpaceShuttle *shuttle, Task tasks[]) {
  // 35 lines
}
// Other 136 functions, 3325 lines in total
int main(void) {
  // 120 lines
}
```

Readability matters

[Best practice] Declare the variable when you first use it!

- If the declaration and use of the variable are too separated, it will become much more difficult to figure out what they are used for as the program goes longer.

[Best practice] Use meaningful names!

- The program would be a mess if polluted with names like `a`, `b`, `c`, `d`, `x`, `y`, `cnt`, `cnt_2`, `flag1`, `flag2`, `flag3` everywhere.
- Use meaningful names: `sumOfScore`, `student_cnt`, `open_success`, ...

Readability is very important. Many students debug day and night simply because their programs are not human-readable.

Use of global variables

One reason for using global variables is to have them shared between functions:

```c
int input;
void work(void) {
  printf("%d\n", input);
}
int main(void) {
  scanf("%d", &input);
  work();
}
```

```c
void work(void) {
  // Error: `input` was not decared
  // in this scope.
  printf("%d\n", input);
}
int main(void) {
  int input;
  scanf("%d", &input);
  work();
}
```

$\Rightarrow$ More about scopes and name lookup in later lectures / recitations.

Initialize a variable

A variable can be **initialized** on declaration.

```
int x = 42; // Declares the variable `x` of type `int`,
            // and initializes its value to 42.
int a = 0, b, c = 42; // Declares three `int` variables, with `a` initialized
                      // to 0, `c` initialized to 42, and `b` uninitialized.
```

This is syntactically **different** (though seems equivalent) to

```
int x;  // Declares `x`, uninitialized.
x = 42; // Assigns 42 to `x`.
```

[Best practice] Initialize the variable if possible. Prefer initialization to later assignment.

⇒ More on initialization in later lectures.

Arithmetic types

Refer to this page for a complete, detailed and standard documentation.

Integer types

Is `int` equivalent to $\mathbb{Z}$?

- Is there a limitation on the numbers that `int` can represent?

Integer types

Is `int` equivalent to $\mathbb{Z}$?

- Is there a limitation on the numbers that `int` can represent?

Experiment:

```c
#include <stdio.h>

int main(void) {
  int x = 1;
  while (1) {
    printf("%d\n", x);
    x *= 2; // x = x * 2
    getchar();
  }
}
```

- On 64-bit Ubuntu 22.04 and compiled with GCC 13, after printing `1073741824` (2^{30}), the output becomes negative, and then `0`.

```
1073741824
-2147483648
0
0
```

Bits and bytes

Information is stored in computers **in binary**.

- $42_{\text{ten}} = 101010_{\text{two}}$.

A **bit** is either 0 or 1.

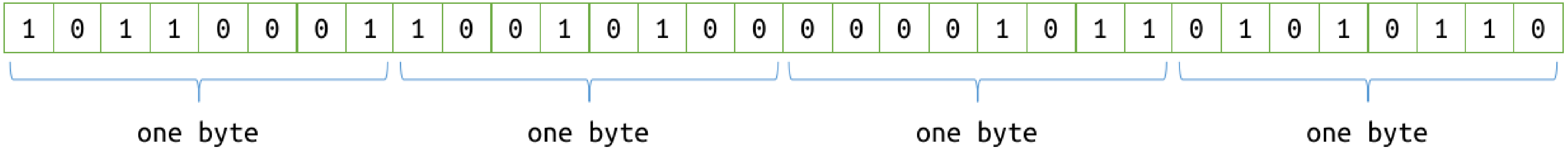
- The binary representation of 42 consists of 6 bits.

A **byte** is 8 bits [2] grouped together like 10001001.

- At least 1 byte is needed to store 42.
- At least 3 bytes are needed to store $142857_{\text{ten}} = 100010111000001001_{\text{two}}$

Bits and bytes

A 32-bit number: $2979269462_{\text{ten}} = 10110001100101000000101101010110_{\text{two}}$.



Suppose now we have n bits.

- How many different values can be represented?
- What is the largest integer that can be represented?
- How do we represent negative numbers? Non-integer values? …

Bits and bytes

Suppose now we have n bits.

- How many different values can be represented?
  - 2^n.
- What is the largest integer that can be represented?
  - $2^n - 1 = \underbrace{111\ldots1}_{n}{}_{\text{two}}$.
- How do we represent negative numbers? Non-integer values? ...
  - There are several different signed number representations, among which **two's complement** is widely used.
  - About floating-point numbers: IEEE754
  - Details are not covered in CS100.

Integer types

An integer type in C is either **signed** or **unsigned**, and has a **width** denoting the number of bits that can be used to represent values.

Suppose we have an integer type of n bits in width.

- If the type is **signed** [3], the range of values that can be represented is $\left[-2^{n-1}, 2^{n-1} - 1\right]$.
- If the type is **unsigned**, the range of values that can be represented is $\left[0, 2^n - 1\right]$.

Integer types

`(signed) short (int)`	`unsigned short (int)`
`signed / int / signed int`	`unsigned (int)`
`(signed) long (int)`	`unsigned long (int)`
`(signed) long long (int)`	`unsigned long long (int)`

Integer types

- The keyword `int` is optional in types other than `int`:
  - e.g. `short int` and `short` name the same type.
  - e.g. `unsigned int` and `unsigned` name the same type.
- "Unsigned-ness" needs to be written explicitly: `unsigned int`, `unsigned long`, ...
- Types without the keyword `unsigned` are signed by default:
  - e.g. `signed int` and `int` name the same type.
  - e.g. `signed long int`, `signed long`, `long int` and `long` name the same type.

Width of integer types

type	width (at least)	width (usually)
`short`	16 bits	16 bits
`int`	16 bits	32 bits
`long`	32 bits	32 or 64 bits
`long long`	64 bits	64 bits

- A signed type has the same width as its `unsigned` counterpart.
- **It is also guaranteed that `sizeof(short)` $\leqslant$ `sizeof(int)` $\leqslant$ `sizeof(long)` $\leqslant$ `sizeof(long long)`.**
  - `sizeof(T)` is the number of **bytes** that `T` holds.

Implementation-defined behaviors

The standard states that the exact width of the integer types is **implementation-defined**.

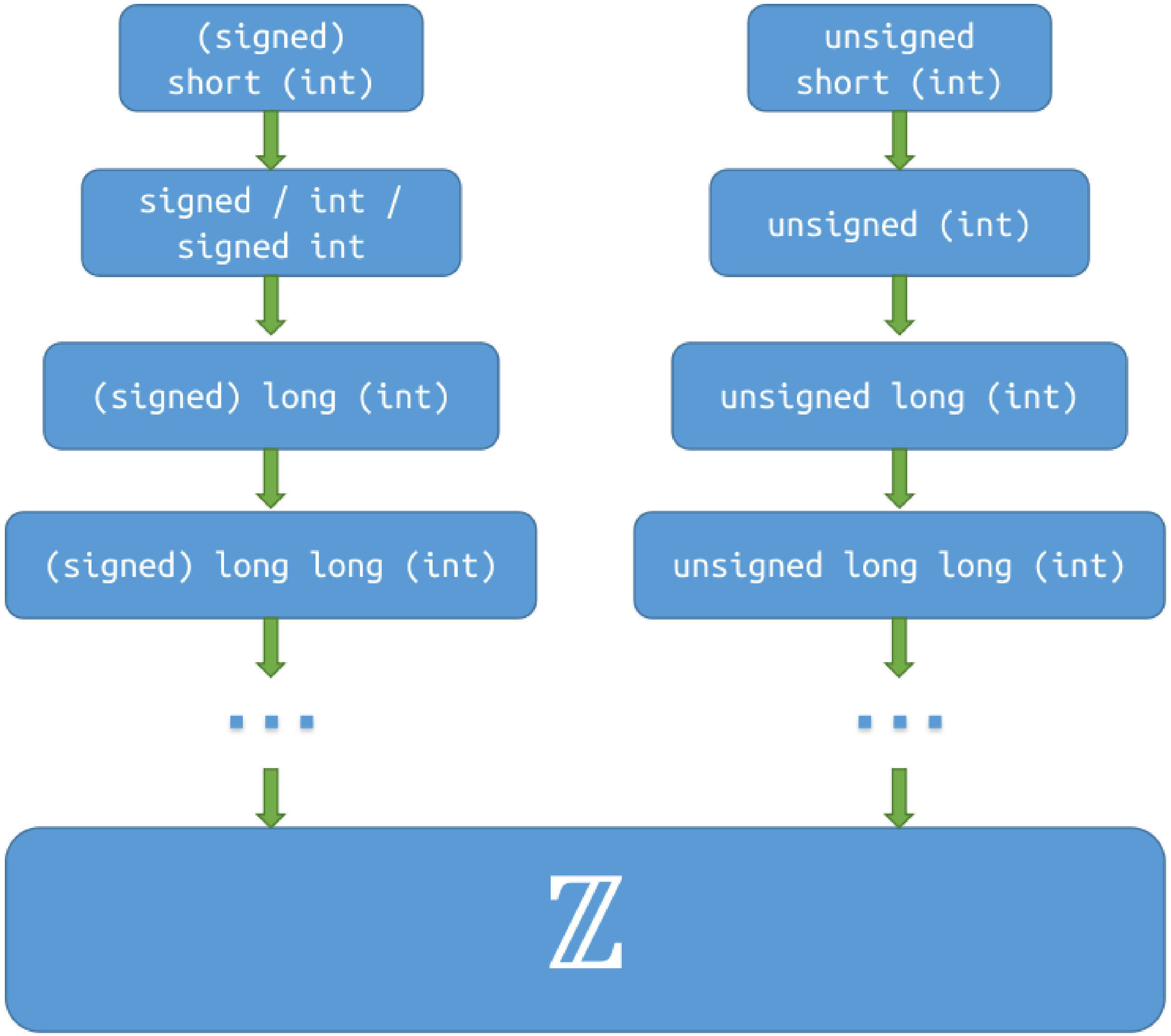
- **Implementation**: The compiler and the standard library.
- An implementation-defined behavior depends on the compiler and the standard library, and is often also related to the hosted environment (e.g. the operating system).

Which one should I use?

`int` is the most optimal integer type for the platform.

- Use `int` for integer arithmetic by default.
- Use `long long` if the range of `int` is not large enough.
- Use smaller types (`short`, or even `unsigned char`) for memory-saving or other special purposes.
- Use `unsigned` types for special purposes. We will see some in later lectures.

Which one is the real world, the integer types or $\mathbb{Z}$?



Real floating types

"Floating-point": The number's radix point can "float" anywhere to the left, right, or between the significant digits of the number.

Real floating-point types can be used to represent *some* real values.

- Real floating-point types $\neq \mathbb{R}$.

Real floating types

C has three types for representing real floating-point values:

- `float`: single precision. Matches IEEE754 binary32 format if supported.
- `double`: double precision. Matches IEEE754 binary64 format if supported.
- `long double`: extended precision. A floating-point type whose precision and range are at least as good as those of `double`.

Details of IEEE754 formats are not required in CS100.

Range of values can be found in this table.

Which one should I use?

Use `double` for real floating-point arithmetic by default.

- In some cases the precision of `float` is not enough.
- Don't worry about efficiency! `double` arithmetic is not necessarily slower than `float`.

Do not use floating-point types for integer arithmetic!

`scanf`/`printf`

Refer to the table in this page.

type	format specifier
`short`	`%hd`
`int`	`%d`
`long`	`%ld`
`long long`	`%lld`

type	format specifier
`unsigned short`	`%hu`
`unsigned`	`%u`
`unsigned long`	`%lu`
`unsigned long long`	`%llu`

- `%f` for `float`, `%lf` for `double`, and `%Lf` for `long double`.

Exercise

Write the "A+B" program for real numbers. Which type do you decide to use? How do you read and print the values?

Exercise

Write the "A+B" program for real numbers. Which type do you decide to use? How do you read and print the values?

```c
#include <stdio.h>

int main(void) {
  double a, b;
  scanf("%lf%lf", &a, &b);
  printf("%lf\n", a + b);
  return 0;
}
```

Character types

The C standard provides three **different** character types: `signed char`, `unsigned char` and `char`.

Let `T` $\in$ {`signed char`, `unsigned char`, `char`}. It is guaranteed that

`1 == sizeof(T) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`.

- **`T` takes exactly 1 byte.**

Question: What is the valid range of `signed char`? `unsigned char`?

Character types

Question: What is the valid range of `signed char`? `unsigned char`?

- `signed char`: $[-128, 127]$.
- `unsigned char`: $[0, 255]$.

What? A character is an integer?

ASCII (American Standard Code for Information Interchange)

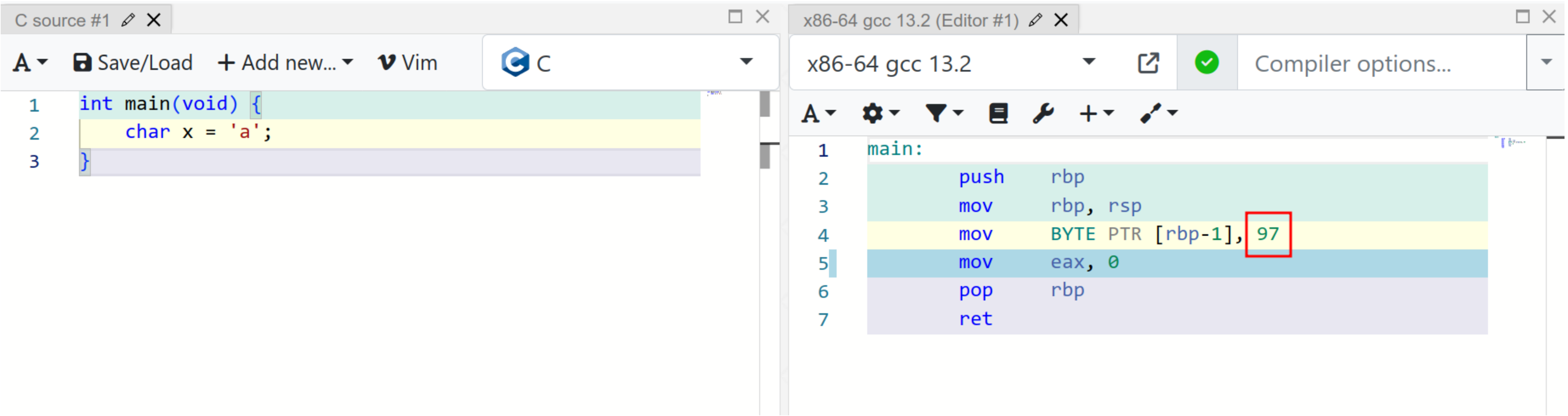
A character is represented in computers as its ASCII code, which is a small integer.

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	*[ENG 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]*

ASCII (American Standard Code for Information Interchange)

A character is represented in computers as its ASCII code, which is a small integer.

- We only consider the so-called *ASCII characters* here.



A character is **nothing but** an integer! In C, there is no "conversion" between characters and ASCII code!

ASCII (American Standard Code for Information Interchange)

Important things to remember:

- [`'0'`, `'9'`] $= [48, 57]$.
- [`'A'`, `'Z'`] $= [65, 90]$.
- [`'a'`, `'z'`] $= [97, 122]$.

Example: Given a lowercase letter, return its uppercase form.

```
char to_uppercase(char x) {
  return x - 32;
}
```

[Best practice] Avoid magic numbers

What is the meaning of `32` here? $\Rightarrow$ a magic number.

```
char to_uppercase(char x) {
  return x - 32;
}
```

Write it in a more human-readable way:

```
char to_uppercase(char x) {
  return x - ('a' - 'A');
}
```

Escape sequence

Some special characters are not directly representable: newline, tab, quote, …

We use escape sequences, e.g.

escape sequence	description
`\'`	single quote
`\"`	double quote
`\\`	backslash

escape sequence	description
`\n`	newline
`\r`	carriage return
`\t`	horizontal tab

Character types

`char`, `signed char` and `unsigned char` are **three different types**.

- Whether `char` is signed or unsigned is **implementation-defined**.
- If `char` is signed (unsigned), it represents the same set of values as the type `signed char` (`unsigned char`), but **they are not the same type**.
  - In contrast, `T` and `signed T` are the same type for `T` $\in$ {`short`, `int`, `long`, `long long`}.

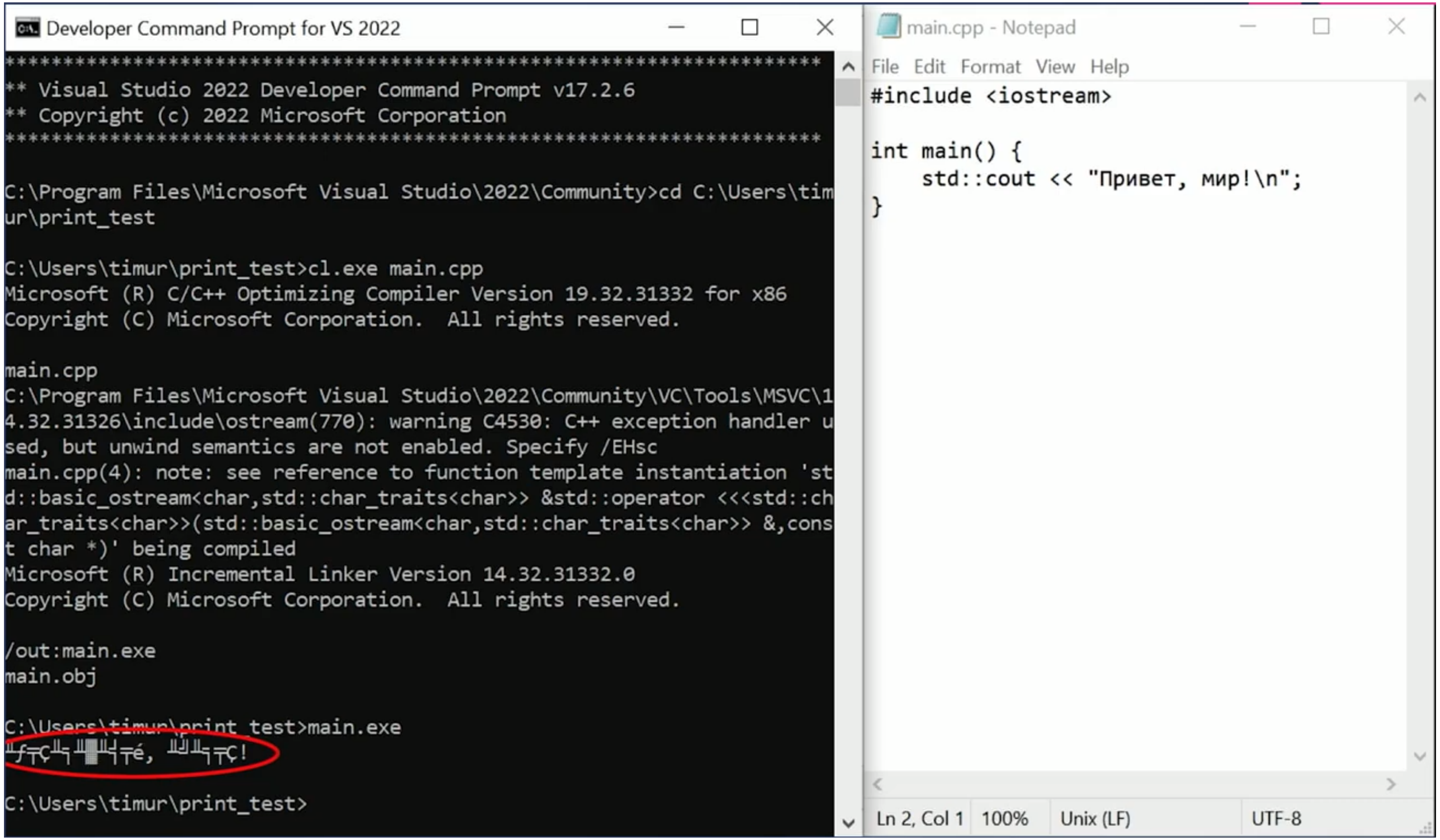
Character types

For almost all cases, use `char` (or, sometimes `int`) to represent characters.

`signed char` and `unsigned char` are used for other purposes.

To read/print a `char` using `scanf`/`printf`, use `%c`.

Sad story: Handling non-ASCII characters? ...

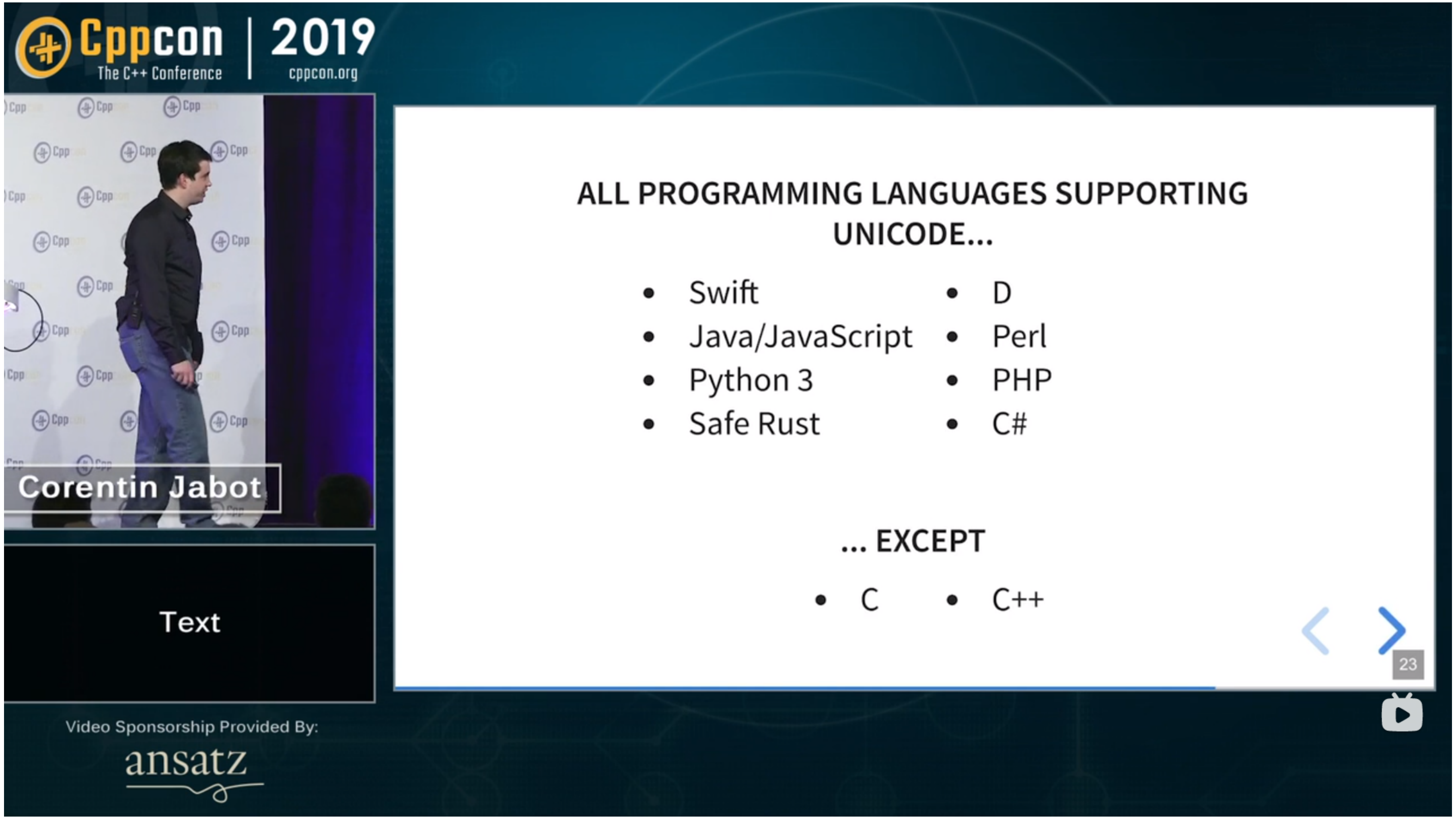


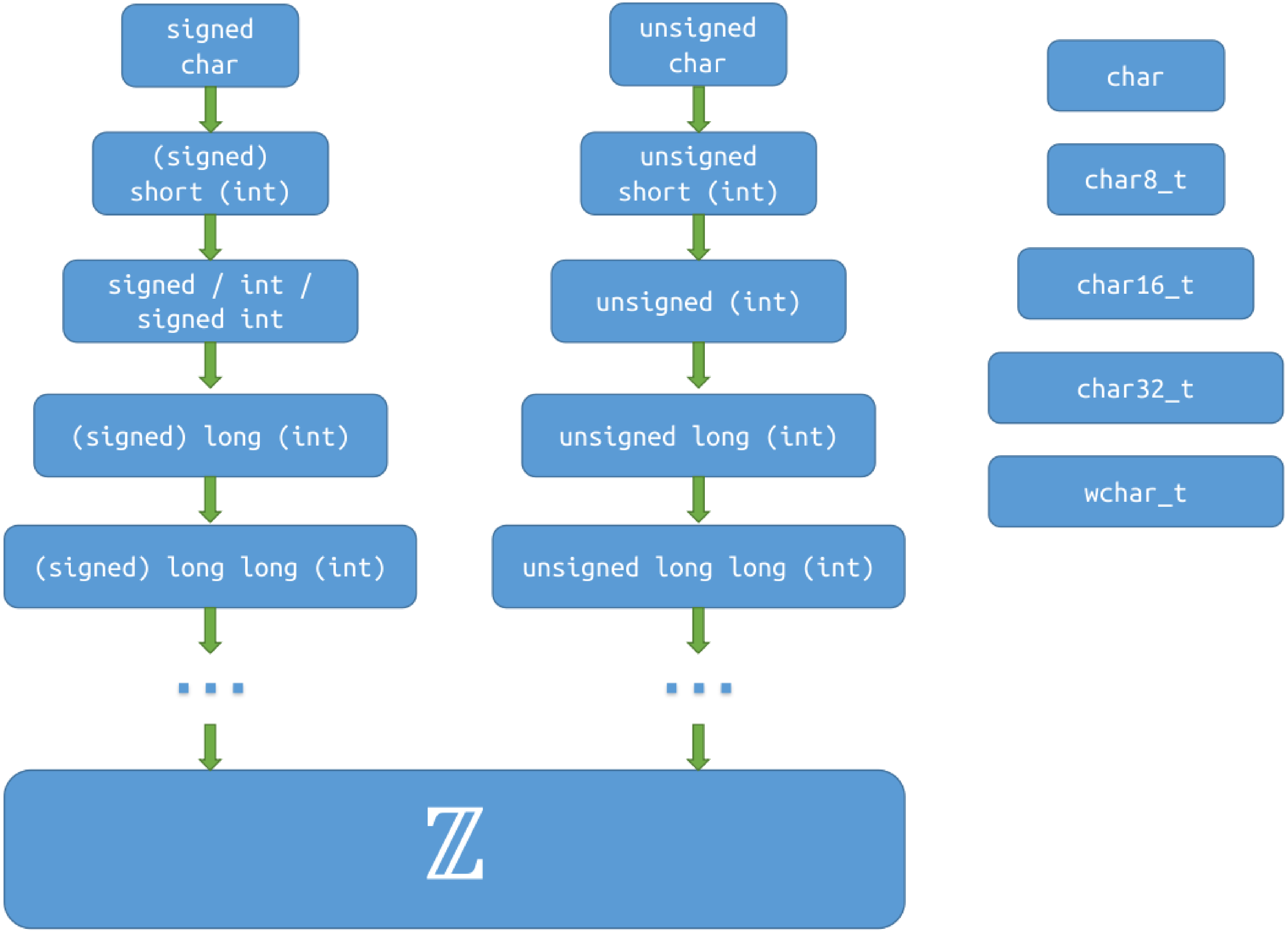
Sad story: Handling non-ASCII characters? ...

Even though the standard provides `wchar_t`, `char8_t` (since C23), `char16_t` and `char32_t` to handle Unicode characters, there are still a lot of problems.

C++23 has some improvement.

That's why Python people laugh at us ...




signed char
(signed) short (int)
signed / int / signed int
(signed) long (int)
(signed) long long (int)
...
unsigned char
unsigned short (int)
unsigned (int)
unsigned long (int)
unsigned long long (int)
...
$\mathbb{Z}$
char
char8_t
char16_t
char32_t
wchar_t

Boolean type: `bool` (since C99)

A type that represents true/false, 1/0, yes/no, ...

To access the name `bool`, `true` and `false`, `<stdbool.h>` is needed. (until C23)

Example: Define a function that accepts a character and returns whether that character is a lowercase letter.

Before C99, using `int`, `1` and `0`:

```c
int is_lowercase(char c) {
  if (c >= 'a' && c <= 'z')
    return 1;
  else
    return 0;
}
```

Since C99, using `bool`, `true` and `false`:

```c
bool is_lowercase(char c) {
  if (c >= 'a' && c <= 'z')
    return true;
  else
    return false;
}
```

Boolean type: `bool` (since C99)

Before C99, using `int`, `1` and `0`:

```c
int is_lowercase(char c) {
  if (c >= 'a' && c <= 'z')
    return 1;
  else
    return 0;
}
```

Since C99, using `bool`, `true` and `false`:

```c
bool is_lowercase(char c) {
  if (c >= 'a' && c <= 'z')
    return true;
  else
    return false;
}
```

Both return values can be used as follows:

```c
char c; scanf("%c", &c);
if (is_lowercase(c)) {
  // do something when c is lowercase ...
}
```

[Best practice] Simplify your code

Just return the result of the condition expression.

```c
int is_lowercase(char c) {
  return c >= 'a' && c <= 'z';
}
```

```c
bool is_lowercase(char c) {
  return c >= 'a' && c <= 'z';
}
```

We will introduce the operators (`&&`, `<=`, `>=`) involved here in later lectures.

Summary

- Variable declaration
  - Type + name
  - Multiple variables in one declaration statement
  - Global vs local
  - Initialization

Summary

- Arithmetic types

signed char	unsigned char	char	bool
(signed) short (int)	unsigned short (int)	char8_t	float
signed / int / signed int	unsigned (int)	char16_t	double
(signed) long (int)	unsigned long (int)	char32_t	long double
(signed) long long (int)	unsigned long long (int)	wchar_t	

Summary

- Arithmetic types
  - Width, signed-ness, valid range
  - Which type to choose
  - Characters: ASCII code, escape sequence
  - Boolean

Exercise

Write a simple calculator that handles input of the form `x op y`, where `x` and `y` are floating-point numbers and `op` $\in$ { `'+'`, `'-'`, `'*'`, `'/'` }. You may use a group of `if`-`else` statements like this:

```
if (op == '+') {
  // ...
} else if (op == '-') {
  // ...
} else if (op == '*') {
  // ...
} else if (op == '/') {
  // ...
} else {
  // report an error
}
```

Notes

[1] The type of every expression in C is determined at compile-time except for *variable-length arrays* (since C99).

[2] A byte is 8 bits on most platforms, but we do have exceptions: 36-bit computing.

[3] There are several different signed number representations, but all popular machines and almost all compilers use **two's complement**. Before C23 and C++20, the C/C++ standards allow for all possible representations, so the minimal valid range for a n-bit integer is $\left[-2^{n-1}+1, 2^{n-1}-1\right]$, which is the range for *one's complement* and *sign-and-magnitude*. Since C23 and C++20, the only representation allowed is two's complement, so the valid range is guaranteed to be $\left[-2^{n-1}, 2^{n-1}-1\right]$. In CS100 we still assume that two's complement is used, even though we are based on C17/C++17.