

CS100 Lecture 2

Variables and Arithmetic Types

Contents

- Variable declaration
- Arithmetic types
 - Bits and bytes
 - Integer types
 - Real floating types
 - Character types
 - Boolean type

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**. \Leftrightarrow C has a **static type system**.
 - In contrast, Python is **dynamically-typed**.

*Note: The type of every variable is determined at compile-time *except for variable-length arrays (since C99)*.

Statically-typed vs dynamically-typed

Python: dynamically typed

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

C: statically-typed

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

The type of a variable

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

The type of a variable

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

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**.

```
#include <stdio.h>

int x, y; // global variables

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

```
#include <stdio.h>

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

Local variables vs global variables

Which one do you prefer?

```
#include <stdio.h>

int x, y; // global variables

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

```
#include <stdio.h>

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


What are these variables used for?

```
#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.

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:

```
#include <stdio.h>

int main(void) {
    int x = 1;
    while (1) {
        printf("%d\n", 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 ¹ 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

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? ...

A 16-bit number: $45460_{\text{ten}} = 1011000110010100_{\text{two}}$.

1	0	1	1	0	0	0	1	1	0	0	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

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?
 - $2^n - 1 = \underbrace{111 \dots 1}_n_{\text{two}}$.
- 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**², the range of values that can be represented is $[-2^{n-1}, 2^{n-1} - 1]$.
- If the type is **unsigned**, the range of values that can be represented is $[0, 2^n - 1]$.

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)
<code>short</code>	16 bits	16 bits
<code>int</code>	16 bits	32 bits
<code>long</code>	32 bits	32 or 64 bits
<code>long long</code>	64 bits	64 bits

- A signed type has the same width as its `unsigned` counterpart.
- It is also guaranteed that `sizeof(short) ≤ sizeof(int) ≤ sizeof(long) ≤ 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**.

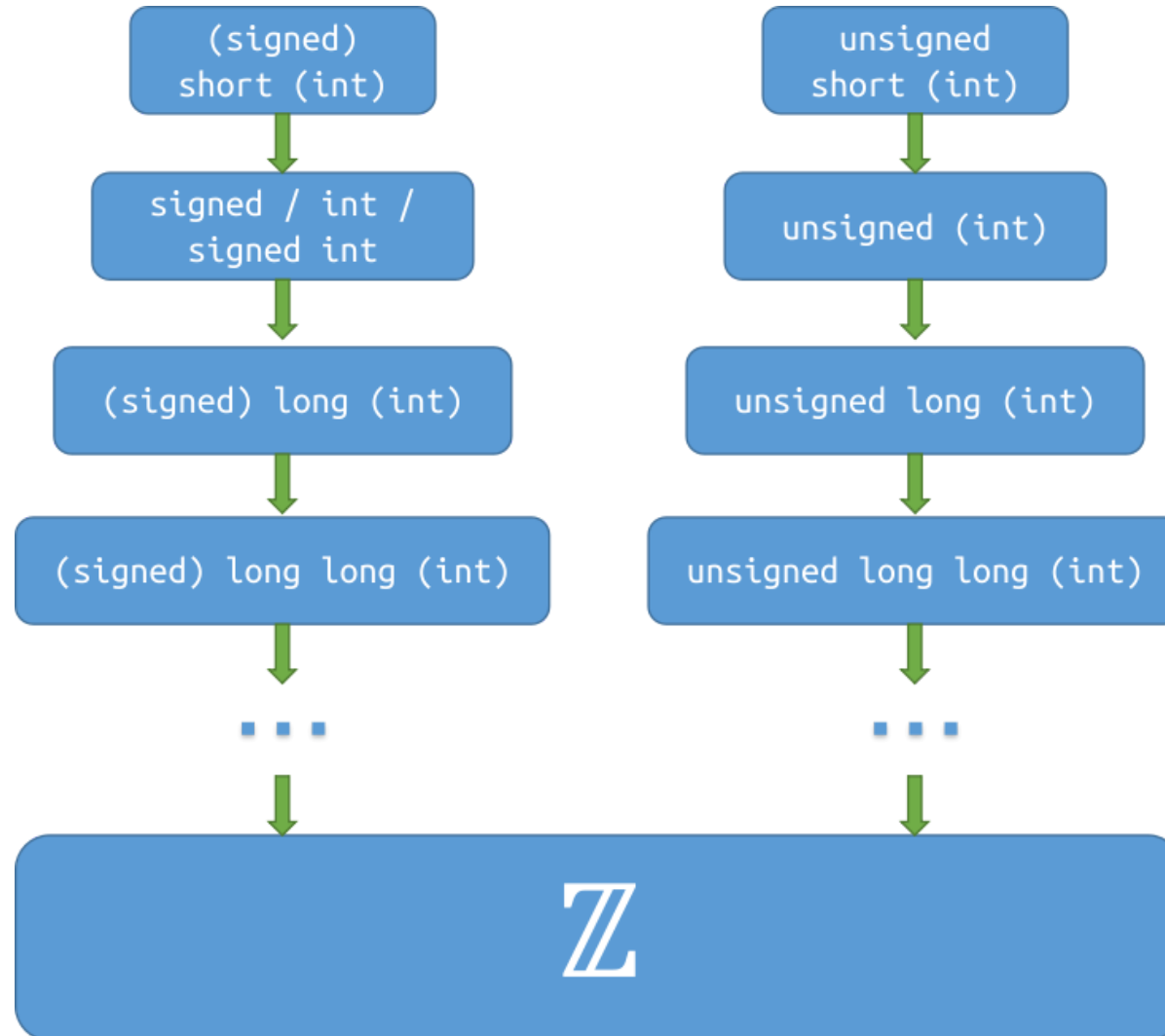
- **Implementation:** The compiler and the standard library.
- An implementation-defined behavior depends on the compiler and the standard library, and is often also related to the hosted environment (e.g. the operating system).

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?

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

$1 == \text{sizeof}(T) \leq \text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long}) \leq \text{sizeof}(\text{long long})$.

- 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`?

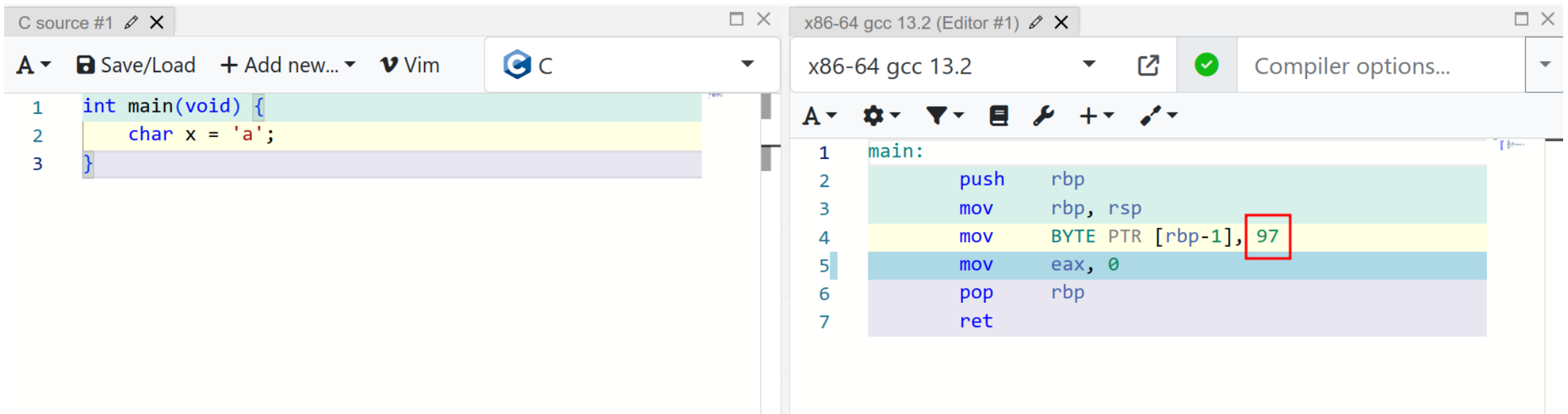
- `signed char`: $[-128, 127]$.
- `unsigned char`: $[0, 255]$.

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.

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



The image shows a side-by-side comparison of C source code and its compiled assembly. On the left, a window titled 'C source #1' displays a C program:

```
1 int main(void) {  
2     char x = 'a';  
3 }
```

 On the right, a window titled 'x86-64 gcc 13.2 (Editor #1)' shows the corresponding assembly code. The assembly includes standard stack frame setup and a return instruction. Line 4, `mov BYTE PTR [rbp-1], 97`, is highlighted in yellow, and the value `97` is enclosed in a red box. This demonstrates that the character 'a' is stored as the integer value 97 in memory.

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
<code>\'</code>	single quote
<code>\"</code>	double quote
<code>\\</code>	backslash

escape sequence	description
<code>\n</code>	newline
<code>\r</code>	carriage return
<code>\t</code>	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` }.

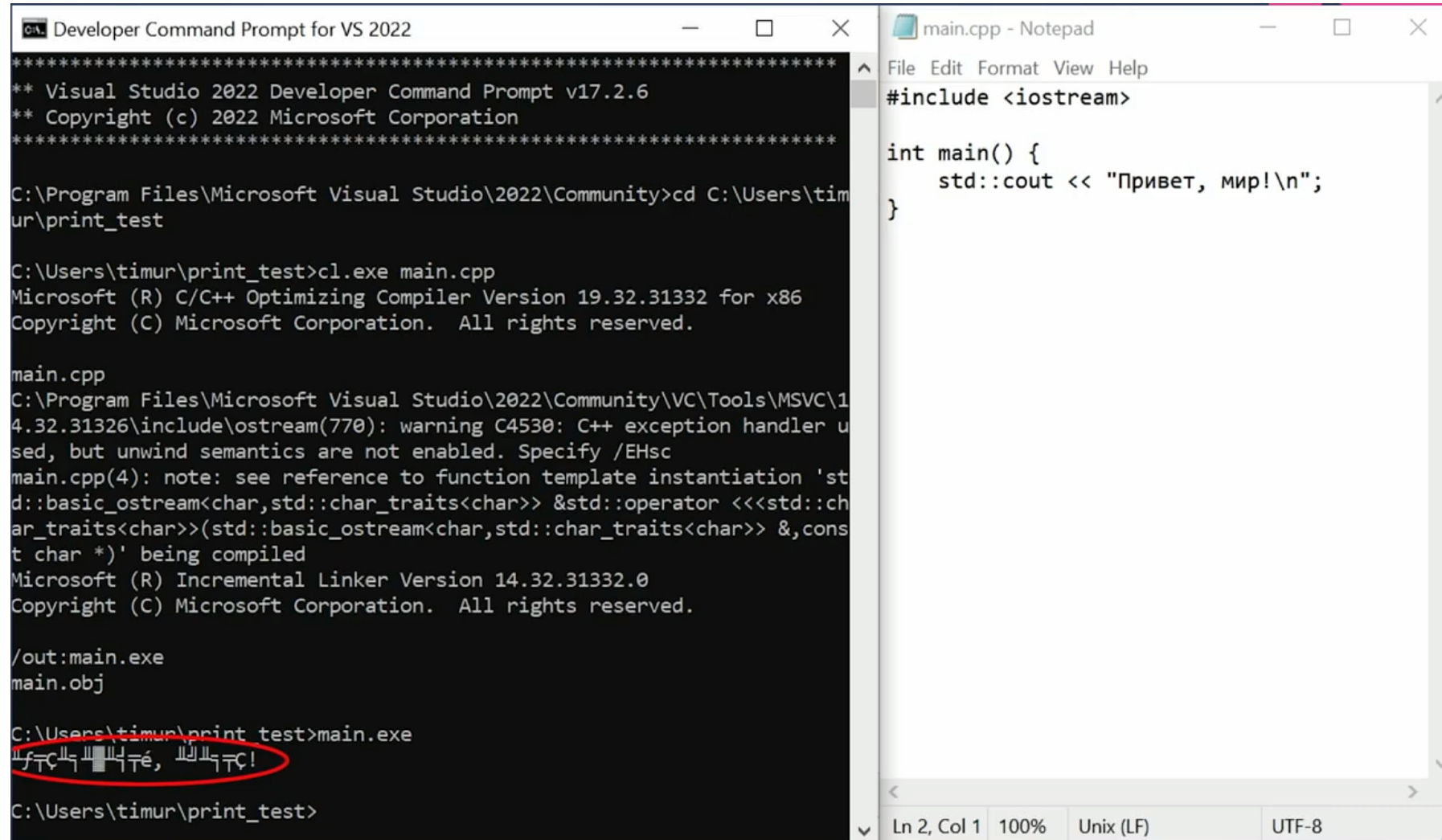
Character types

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 letters? ...



The image shows a Windows development environment with two windows. The left window is the 'Developer Command Prompt for VS 2022', and the right window is 'main.cpp - Notepad'.

Developer Command Prompt for VS 2022:

```
*****
** Visual Studio 2022 Developer Command Prompt v17.2.6
** Copyright (c) 2022 Microsoft Corporation
*****

C:\Program Files\Microsoft Visual Studio\2022\Community>cd C:\Users\timur\print_test

C:\Users\timur\print_test>cl.exe main.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.32.31332 for x86
Copyright (C) Microsoft Corporation. All rights reserved.

main.cpp
C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.32.31326\include\ostream(770): warning C4530: C++ exception handler used, but unwind semantics are not enabled. Specify /EHsc
main.cpp(4): note: see reference to function template instantiation 'std::basic_ostream<char,std::char_traits<char>> &std::operator <<<std::char_traits<char>>(std::basic_ostream<char,std::char_traits<char>> &,const char *)' being compiled
Microsoft (R) Incremental Linker Version 14.32.31332.0
Copyright (C) Microsoft Corporation. All rights reserved.

/out:main.exe
main.obj

C:\Users\timur\print_test>main.exe
Привет, мир!
C:\Users\timur\print_test>
```

main.cpp - Notepad:

```
File Edit Format View Help

#include <iostream>

int main() {
    std::cout << "Привет, мир!\n";
}
```

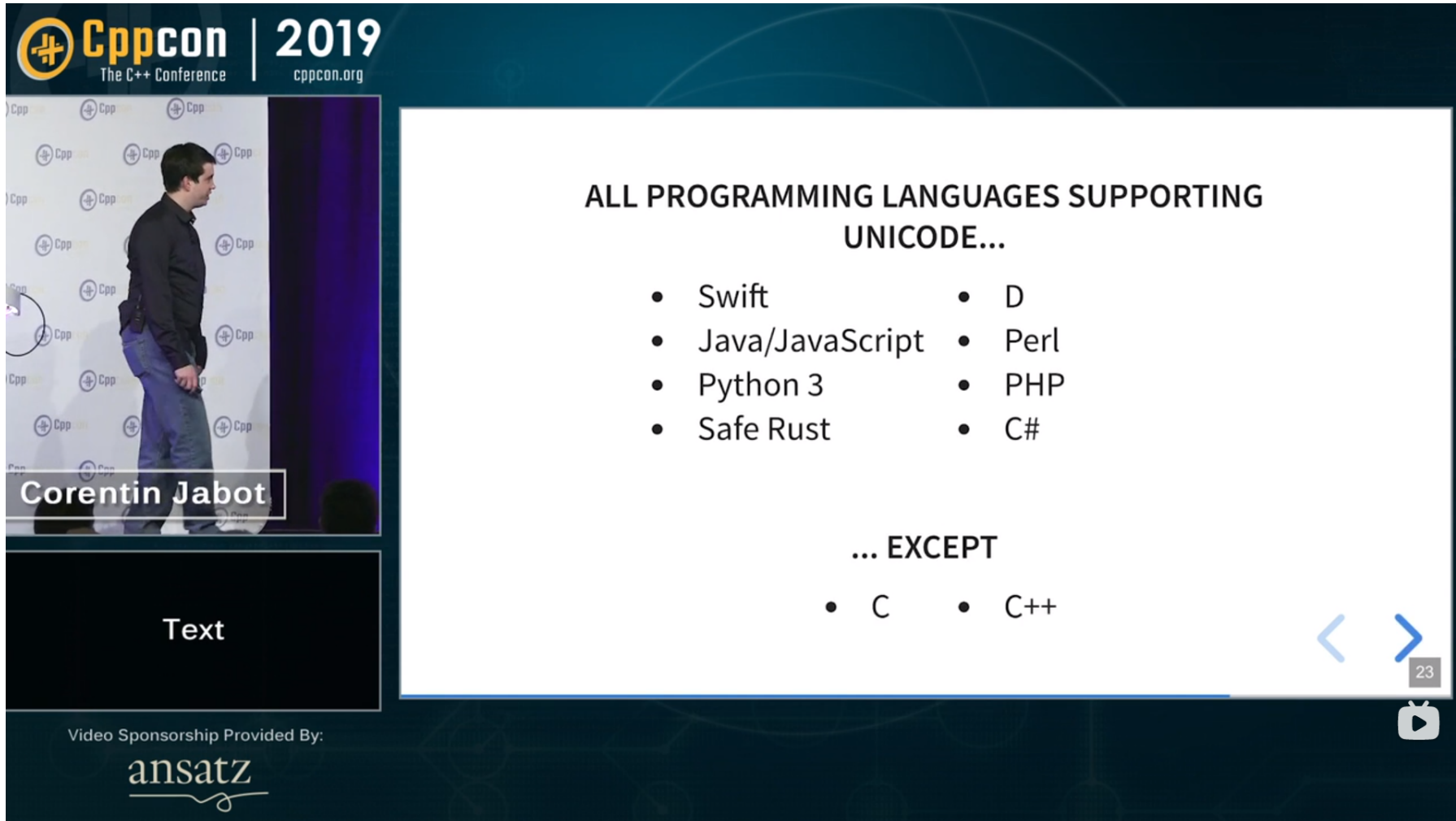
The output of the program in the command prompt is "Привет, мир!". The text "Привет, мир!" is circled in red in the original image.

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

Even though the standard provides `wchar_t`, `char8_t` (since C23), `char16_t` and `char32_t` to handle wide/UTF-8 characters, there are still a lot of problems.

C++23 has some improvement.

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



The image is a screenshot of a video recording from the Cppcon 2019 conference. On the left, a man named Corentin Jabot is standing on a stage, facing right. He is wearing a dark long-sleeved shirt and blue jeans. Behind him is a backdrop with the Cppcon logo and the year 2019. The main part of the image is a large white rectangular area containing text and a list of programming languages. The text reads 'ALL PROGRAMMING LANGUAGES SUPPORTING UNICODE...' followed by a bulleted list of languages: Swift, Java/JavaScript, Python 3, Safe Rust, D, Perl, PHP, and C#. Below this list, it says '... EXCEPT' followed by a bulleted list of C and C++. At the bottom of the slide, there are navigation arrows and a small number '23'. In the bottom left corner of the video frame, there is a logo for 'ansatz' with the text 'Video Sponsorship Provided By:' above it.

Cppcon | 2019
The C++ Conference
cppcon.org

Corentin Jabot

Text

Video Sponsorship Provided By:
ansatz

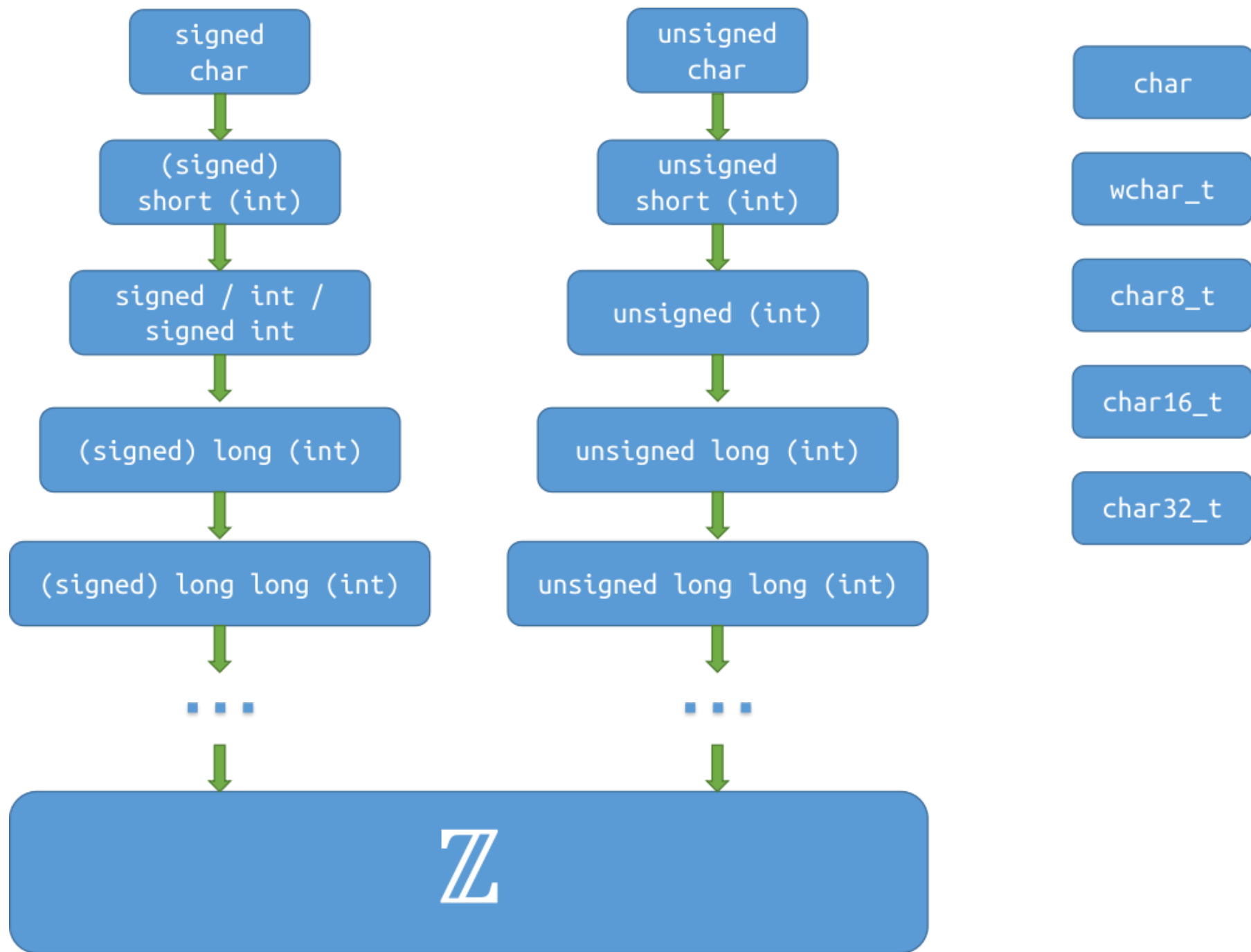
ALL PROGRAMMING LANGUAGES SUPPORTING
UNICODE...

- Swift
- Java/JavaScript
- Python 3
- Safe Rust
- D
- Perl
- PHP
- C#

... EXCEPT

- C
- C++

23



Boolean type: `bool` (since C99)

A type that represents true/false, 0/1, 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, use `int`, `0` and `1`:

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

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

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

Boolean type: `bool` (since C99)

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

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

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

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

Both return values can be used as follows:

```
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.

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

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

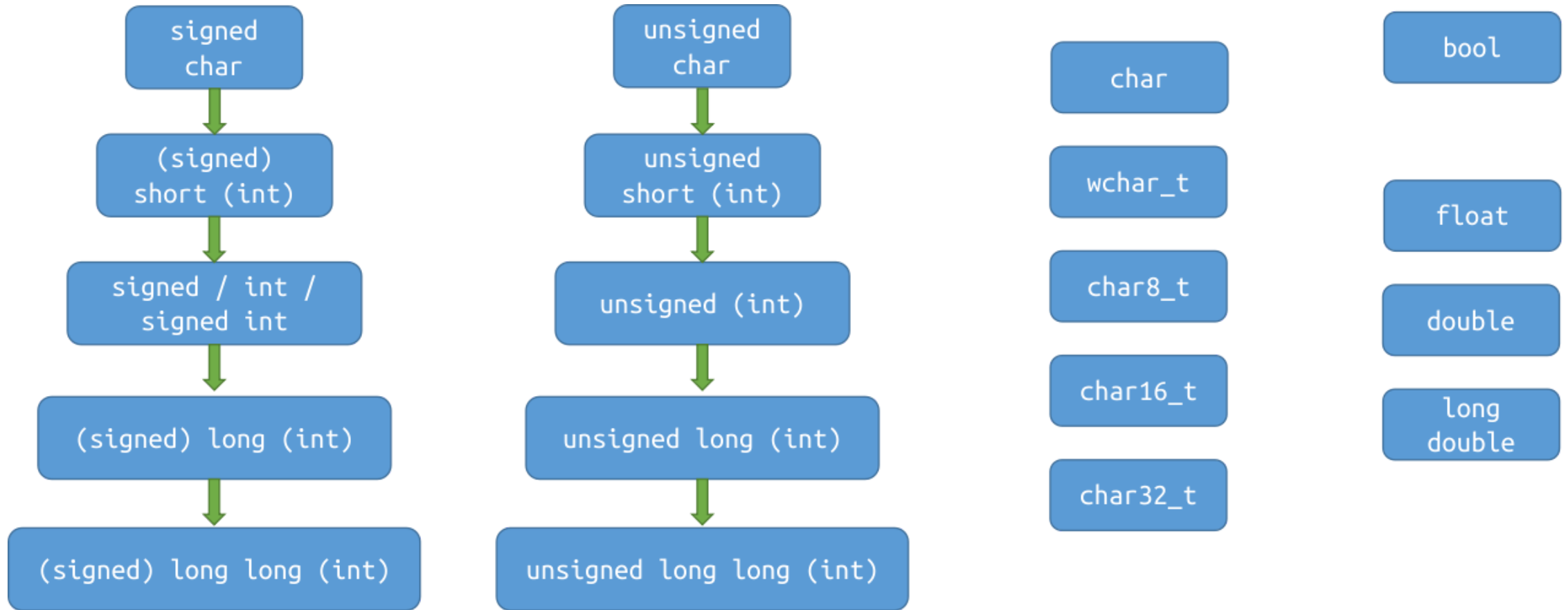
We will introduce the operators (`&&` , `<=` , `>=`) involved here in later lectures.

Summary

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

Summary

- Arithmetic types



Summary

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

Notes

- ¹ A byte is 8 bits on most platforms, but we do have exceptions: [36-bit computing](#).
- ² 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 $[-2^{n-1} + 1, 2^{n-1} - 1]$, 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 $[-2^{n-1}, 2^{n-1} - 1]$. In CS100 we still assume that two's complement is used, even though we are based on C17/C++17.