[Γα=Ω5]

Data Types & Casting

By Ariel Parra

Primitive (built-in) types in python3

Type	Description	Example
`int`	Integer numbers (positive, negative, zero)	`42`, `-17`, `0`
`float`	Floating-point numbers (decimal)	`3.14`, `-2.5`, `1.0`
`bool`	Boolean values	`True`, `False`
`str`	String (sequence of characters)	`"Hello"`, `'World'`
`bytes`	Sequence of bytes	`b"hello"`, `b'\x00\x01'`
`NoneType`	Represents absence of value	`None`

> **Note**: Python's data types have virtually **infinite** capacity (limited only by available RAM) because Python automatically handles memory allocation and can store arbitrarily large numbers, unlike C++ which has fixed-size primitive types.

Primitive types in c++

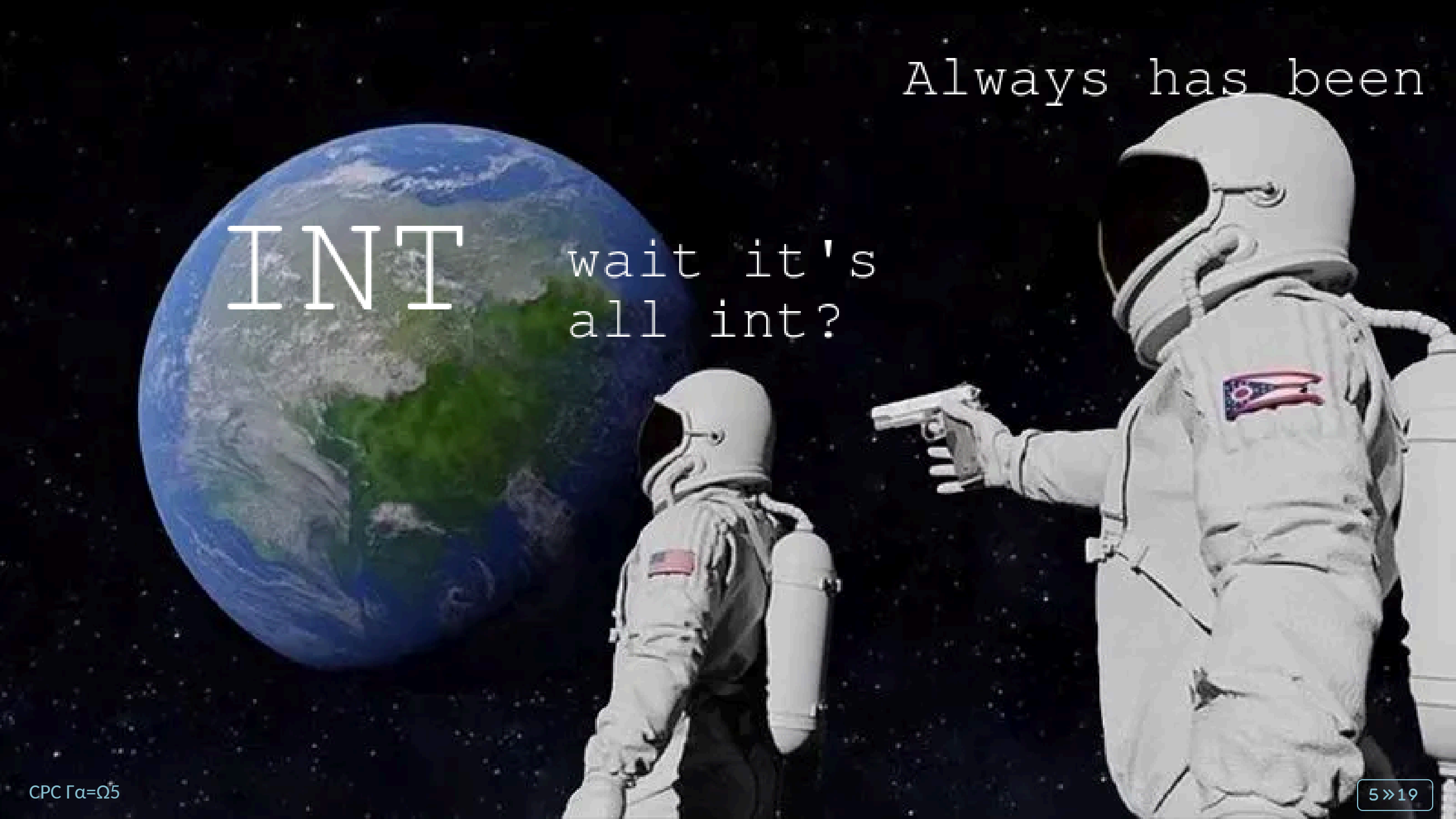
Type	Library	Size	Range (GCC x64)	Description
`char`	`<cstdint>`	8 bits	-128 to 127	Stores a character in single quotes ''
`bool`	`<cstdbool>`	8 bits	true (1) or false (0)	Boolean value (true or false).
`short`	`<cstdint>`	16 bits	-32,768 to 32,767	Short integer.
`int`	`<cstdint>`	32 bits	-2,147,483,648 to 2,147,483,647	Standard integer.
`long long`	`<cstdint>`	64 bits	-9.22e18 to 9.22e18	Extended long integer.
`float`	`<cfloat>`	32 bits	Approx ±3.4e38	Single precision floating-point number.
`double`	`<cfloat>`	64 bits	Approx ±1.7e308	Double precision floating

Synonyms / Alias for `int` values

- **1 byte/8 bits:** `uint8_t`, `unsigned char`, `int8_t`, `signed char`, `int_least8_t`, `uint_least8_t`, `int_fast8_t`, `uint_fast8_t`
- **2 byte/16 bits:** `int16_t`, `short`, `short int`, `signed short int`, `uint16_t`, `unsigned short`, `unsigned short int`, `int_least16_t`, `uint_least16_t`
- **4 bytes/32 bits:** `int32_t`, `int`, `long int` `signed int`, `uint32_t`, `unsigned int`, `unsigned long int`, `long`, `int_least32_t`, `uint_least32_t`, `int_fast16_t`, `uint_fast16_t`, `int_fast32_t`, `uint_fast32_t`
- **8 bytes/64 bits:** `int64_t`, `long long`, `signed long long`, `uint64_t`, `unsigned long long`, `int_least64_t`, `uint_least64_t`, `int_fast64_t`, `uint_fast64_t`, `intmax_t`, `uintmax_t`

Primitive data type conversions depend on the compiler implementation (GCC, Clang, MSVC, etc.) as well as the processor architecture (x86, x64, ARM, etc.).

For example, in x86 processors from 2002, an `int` was worth 16 bits and `long long` was worth 32 bits, and in 2025 a `long long` is 64 bits but an `int32_t` will always have 32 bits.


Always has been
INT
wait it's
all int?

Type qualifiers in c++

Type qualifiers in C/C++ modify the behavior of variables.

- **const**: Defines that a variable is constant and cannot be modified after its initialization, so it will be the one we use in competitions. Example:

```
const int constant = 100;
```

- **static**: Has different uses depending on the context:
  - For **global variables and functions**, it restricts access to the file in which it is defined, limiting its scope to that file.
  - For **local variables**, it means that the variable retains its value between function calls, and is initialized only once. That is, the variable has static duration. Example:

```
void counter() {
    static int count = 0;
    count++;
}
```

- **unsigned**: Indicates that a variable can only store non-negative values. Example:

```
unsigned long long num = 10e9;
```

- **signed**: Indicates that a variable can store both negative and positive values (redundant as it is the default qualifier).
- **long**: long is a data type in itself, but when placed before another type it becomes a type qualifier, commonly used in `long long` and `long double`.
- **inline**: Only for functions, suggests to the compiler to expand the function at the place where it is called, instead of making a normal function call. This can improve performance in some cases.
- * The asterisk operator (*) is used to declare pointers when placed after a primitive type. For example `int *ptr;`
- **volatile**: Indicates that the value of a variable can change at any time, without the program explicitly modifying it (we won't use

Special types in c++

- **auto**: Introduced in C++11, `auto` allows the compiler to automatically deduce the type of a variable from its initialization value. This simplifies code writing and avoids type errors. Example:

```
auto number = 10; // 'number' will be int
auto pi = 3.14;   // 'pi' will be double
```

- **size_t**: is a type guaranteed by the compiler to contain any index (in this case: unsigned int).
- **std::byte**: Introduced in C++17 and defined in `<cstddef>` it is an 8-bit data type that **does not allow** arithmetic operations.
- **__int128**: It is a special data type that can hold 128 bits. It is available in GCC and Clang compilers and can handle very large values, but it is slower and requires buffers for practical use. Its signed range is from -1.70e38 to 1.70e38 and unsigned from 0 to approximately 3.40e38.
- **void**: When used in a function declaration it indicates that the function does not return any value. It is also used to declare generic pointers (`void*`), which can point to any data type.
- **NULL**: Represents a null pointer constant in C.
- **nullptr**: Introduced in C++11, represents a null pointer constant safely in C++.

Literal declarations in c++

Infix

- **Exponential**: Scientific notation, uses the letter `e` after the base number and before the exponent.

```
double value = 1.0e7; // 1 * 10^7
```

Prefixes

- **Hexadecimal**: Prefix `0x`.

```
int value = 0x10; // 16 in decimal
```

- **Binary**: Prefix `0b` (From C++14 onwards).

```
int value = 0b1011; // 11 in decimal
```

- **Octal**: Prefix with `0`.

```
int value = 010; // 8 in decimal
```

Postfixes

Postfixes are used to specify the type of the literal, allowing the compiler to understand the exact type. Although this often ends up being redundant.

- **l** or **L**: Indicates that the literal is of type long or long long.

```
long value = 10L; // Literal long
long long value_ll = 10LL; // Literal long long
```

- **f** and **d**: Indicates that the literal is of type float or double.

```
float value = 10.5f; // Literal float
double value = 10.5; // Literal double
```

- **u**: Indicates that the literal is of type unsigned and can be combined with other postfixes for more specific types.

```
unsigned int value = 10u; // Literal unsigned int
unsigned long long value = 10ULL; // Literal unsigned long long
```

Literal declarations in Python

Infix

- **Exponential**: Scientific notation, uses the letter `e` after the base number and before the exponent.

```
value = 1.0e7  # 1 * 10^7
```

Prefixes

- **Hexadecimal**: Prefix `0x`.

```
value = 0x10  # 16 in decimal
```

- **Binary**: Prefix `0b`.

```
value = 0b1011  # 11 in decimal
```

- **Octal**: Prefix `0o`.

```
value = 0o10  # 8 in decimal
```

String literals in Python

Python offers various ways to declare string literals with prefixes that modify their behavior:

- **Raw strings**: Prefix `r` - backslashes are treated literally.

```
path = r"C:\Users\name\Documents"  # No need to escape backslashes
```

- **Byte strings**: Prefix `b` - creates a bytes object instead of str.

```
data = b"Hello"  # bytes object
```

- **f-strings**: Prefix `f` - formatted string literals (Python 3.6+).

```
name = "World"
message = f"Hello {name}!"  # "Hello World!"
```

> **Note**: Unlike C++, Python doesn't use postfixes for numeric types since Python automatically handles type inference and has dynamic typing.

Casting in Python

Casting or type conversion is the process of converting one data type to another. In Python, there are two types:

- **Implicit Casting**
  Python automatically performs some conversions between data types, known as implicit casting.

```
i = 10        # int
j = 3.14      # float
result = i + j  # i is automatically converted to float: 13.14
```

- **Explicit Casting**
  To convert a data type to another in a more controlled way, explicit casting is used. This is done using built-in functions:

```
a = 15, b = 7
result = float(a) / float(b)  # Convert to float before division
```

Casting in C lang

Casting or type conversion is the process of converting one data type to another and there are two types:

- **Implicit Casting**

  C automatically performs some conversions between data types, known as implicit casting.

```c
int i = 10;
long long j = i; // i went from int to long long
```

- **Explicit Casting**

  To convert a data type to another in a more controlled way, explicit casting is used. This is done by using parentheses with the destination type before the expression to be converted:

```c
int a = 15, b = 7;
float result = (float)(a / b);
```

Casting in C++

- **static_cast**: this is the one we will use in the course because the others are more related to object-oriented programming and/or memory management.

```
int a = 15, b = 7;
double result = static_cast<double>(a) / b;
```

- **dynamic_cast**: Used mainly with pointers and references to polymorphic classes to cast down in the inheritance hierarchy.
- **const_cast**: Allows adding or removing const from a variable.
- **reinterpret_cast**: For low-level conversions, such as converting between pointers of unrelated types.

Limits of each type in c++

<climits>

This C library includes constants for the limits of each type:

`CHAR_BIT`, `SCHAR_MIN`, `SCHAR_MAX`, `UCHAR_MAX`, `CHAR_MIN`, `CHAR_MAX`, `MB_LEN_MAX`, `SHRT_MIN`, `SHRT_MAX`, `USHRT_MAX`, `INT_MIN`, `INT_MAX`, `UINT_MAX`, `LONG_MIN`, `LONG_MAX`, `ULONG_MAX`, `LLONG_MIN`, `LLONG_MAX`, `ULLONG_MAX`

<limits>

Unlike `<climits>`, this C++ library includes functions to handle the limits of each type:

- Minimum value for any primitive type `T`:

```
std::numeric_limits<T>::min()
```

- Maximum value for any primitive type `T`:

```
std::numeric_limits<T>::max()
```

Problems

- Codeforces **677A** Vanya and Fence
- Codeforces **791A** Bear and Big Brother

References

- Agarwal, H. (2023). *C++ Data Types*. GeeksforGeeks. https://www.geeksforgeeks.org/cpp-data-types/
- bug8wdqo. (2024). *Type Qualifiers in C++*. GeeksforGeeks. https://www.geeksforgeeks.org/type-qualifiers-in-cpp/
- cplusplus. (2023). *< climits > (limits.h)*. https://cplusplus.com/reference/climits/
- GeeksforGeeks. (2023). *Data Types in C*. https://www.geeksforgeeks.org/data-types-in-c/
- GeeksforGeeks. (2025). *Type Casting in Python (Implicit and Explicit) with Examples*. https://www.geeksforgeeks.org/python/type-casting-in-python/
- GeeksforGeeks. (2025). *Primitive Data Types vs Non Primitive Data Types in Python*. https://www.geeksforgeeks.org/python/primitive-data-types-vs-non-primitive-data-types-in-python/
- GNU. (2024). *128-bit Integers*. https://gcc.gnu.org/onlinedocs/gcc/_005f_005fint128.html
- kushagra. (2018). *"static const" vs "#define" vs "enum"*. GeeksforGeeks. https://www.geeksforgeeks.org/static-const-vs-define-vs-enum/

- Microsoft. (2023). *Standard types*. Microsoft Learn. https://learn.microsoft.com/en-us/cpp/c-runtime-library/standard-types?view=msvc-170
- Microsoft. (2024). *Data types ranges*. Microsoft Learn. https://learn.microsoft.com/en-us/cpp/cpp/data-type-ranges?view=msvc-170
- Pain--In--The--Brain. (2020). *Why use std::byte over unsigned char?* Reddit. https://www.reddit.com/r/cpp_questions/comments/hlyav9/why_use_stdbyte_over_unsigned_char/
- Python Software Foundation. (2025). *Built-in Types*. Python Documentation. https://docs.python.org/3/library/stdtypes.html
- Stannum. (2016). *Unsized integer types and signedness*. https://stannum.io/blog/0MXgB0
- Trivedi, U. (2024a). *Type Inference in C++ (auto and decltype)*. GeeksforGeeks. https://www.geeksforgeeks.org/type-inference-in-c-auto-and-decltype/
- Trivedi, U. (2024b). *Understanding nullptr in C++*. GeeksforGeeks. https://www.geeksforgeeks.org/understanding-nullptr-c/