Data types

Types

Types	Category	Meaning	Example
float double long double	Floating Point	a number with a fractional part	3.14159
bool	Integral (Boolean)	true or false	true
char wchar_t char8_t (C++20) char16_t (C++11) char32_t (C++11)	Integral (Character)	a single character of text	'c'
short int int long int long long int (C++11)	Integral (Integer)	positive and negative whole numbers, including 0	64
std::nullptr_t (C++11)	Null Pointer	a null pointer	nullptr
void	Void	no type	n/a

Category	Type	Minimum Size	Typical Size	Note
Boolean	bool	1 byte	1 byte	
character	char	1 byte	1 byte	always exactly 1 byte
	wchar_t	1 byte	2 or 4 bytes	
	char8_t	1 byte	1 byte	
	char16_t	2 bytes	2 bytes	
	char32_t	4 bytes	4 bytes	
integer	short	2 bytes	2 bytes	
	int	2 bytes	4 bytes	
	long	4 bytes	4 or 8 bytes	
	long long	8 bytes	8 bytes	
floating point	float	4 bytes	4 bytes	
	double	8 bytes	8 bytes	
	long double	8 bytes	8, 12, or 16 bytes	
pointer	std::nullptr_t	4 bytes	4 or 8 bytes	

```cpp
#include <iomanip> // for std::setw (which sets the width of the subsequent output)
#include <iostream>

int main()
{
    std::cout << std::left; // left justify output
    std::cout << std::setw(16) << "bool:" << sizeof(bool) << " bytes\n";
    std::cout << std::setw(16) << "char:" << sizeof(char) << " bytes\n";
    std::cout << std::setw(16) << "short:" << sizeof(short) << " bytes\n";
    std::cout << std::setw(16) << "int:" << sizeof(int) << " bytes\n";
    std::cout << std::setw(16) << "long:" << sizeof(long) << " bytes\n";
    std::cout << std::setw(16) << "long long:" << sizeof(long long) << " bytes\n";
    std::cout << std::setw(16) << "float:" << sizeof(float) << " bytes\n";
    std::cout << std::setw(16) << "double:" << sizeof(double) << " bytes\n";
    std::cout << std::setw(16) << "long double:" << sizeof(long double) << " bytes\n";

    return 0;
}
```

Output:

```
bool:           1 bytes
char:           1 bytes
short:          2 bytes
int:            4 bytes
long:           4 bytes
long long:      8 bytes
float:          4 bytes
double:         8 bytes
long double:    8 bytes
```

```cpp
short s;      // prefer "short" instead of "short int"
int i;
long l;       // prefer "long" instead of "long int"
long long ll; // prefer "long long" instead of "long long int"
```

Best practise: Prefer the shorthand types that do not use the int suffix or signed prefix.

an 8-bit integer contains 8 bits. 28 is 256, so an 8-bit integer can hold 256 possible values. There are 256 possible values between -128 to 127, inclusive.

7 bits are used to hold the magnitude of the number, and 1 bit is used to hold the sign.

avoid unsigned ints

```
#include <iostream>

// assume int is 4 bytes
int main()
{
    unsigned int x{ 2 };
    unsigned int y{ 3 };

    std::cout << x - y << '\n'; // prints 4294967295 (incorrect!)

    return 0;
}
```

prints 4294967295 (incorrect!)

So when should you use unsigned numbers?

There are still a few cases in C++ where it's okay / necessary to use unsigned numbers.

First, unsigned numbers are preferred when dealing with **bit manipulation** (covered in chapter O -- that's a capital 'o', not a '0'). They are also useful when well-defined wrap-around behavior is required (useful in some algorithms like encryption and random number generation).

Second, use of unsigned numbers is still unavoidable in some cases, mainly those having to do with **array indexing**. We'll talk more about this in the lessons on arrays and array indexing.

Also note that if you're developing for an **embedded system** (e.g. an Arduino) or some other **processor/memory limited context**, use of unsigned numbers is more common and accepted (and in some cases, unavoidable) for **performance reasons**.

Fixed-width integers

Name	Type	Range	Notes
std::int8_t	1 byte signed	-128 to 127	Treated like a signed char on many systems. See note below.
std::uint8_t	1 byte unsigned	0 to 255	Treated like an unsigned char on many systems. See note below.
std::int16_t	2 byte signed	-32,768 to 32,767	
std::uint16_t	2 byte unsigned	0 to 65,535	
std::int32_t	4 byte signed	-2,147,483,648 to 2,147,483,647	
std::uint32_t	4 byte unsigned	0 to 4,294,967,295	
std::int64_t	8 byte signed	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	
std::uint64_t	8 byte unsigned	0 to 18,446,744,073,709,551,615	

Fast and least integers

#include // for fast and least types
#include

```
int main()
{
    std::cout << "least 8:  " << sizeof(std::int_least8_t) * 8 << " bits\n";
    std::cout << "least 16: " << sizeof(std::int_least16_t) * 8 << "
bits\n";
    std::cout << "least 32: " << sizeof(std::int_least32_t) * 8 << "
bits\n";
    std::cout << '\n';
    std::cout << "fast 8:  " << sizeof(std::int_fast8_t) * 8 << " bits\n";
    std::cout << "fast 16: " << sizeof(std::int_fast16_t) * 8 << " bits\n";
    std::cout << "fast 32: " << sizeof(std::int_fast32_t) * 8 << " bits\n";

    return 0;
}
```

You can see that std::int_least16_t is 16 bits, whereas std::int_fast16_t is actually 32 bits. This is because on the author's machine, 32-bit integers are faster to process than 16-bit integers.

However, these fast and least integers have their own downsides: First, **not many programmers actually use them**, and a lack of familiarity can lead to errors. Second, the fast types **can lead to memory wastage**, as their actual **size may be larger** than indicated by their name. More seriously, because the size of the fast/least integers can vary, it's possible that your program may exhibit different behaviors on architectures where they resolve to different sizes

Warning

The 8-bit fixed-width integer types are often treated like chars instead of integer values (and this may vary per system). Prefer the 16-bit fixed integral types for most cases.

Best practices for integral types

- Prefer int when the size of the integer doesn't matter (e.g. the number will always fit within the range of a 2-byte signed integer) and the variable is short-lived (e.g. destroyed at the end of the function). For example, if you're asking the user to enter their age, or counting from 1 to 10, it doesn't matter whether int is 16 or 32 bits (the numbers will fit either way). This will cover the vast majority of the cases you're likely to run across.
- Prefer std::int#_t when storing a quantity that needs a guaranteed range.
- Prefer std::uint#_t when doing bit manipulation or where well-defined wrap-around behavior is required.

Avoid the following when possible:

- **short** and **long** integers -- use a fixed-width type instead.
- Unsigned types for holding quantities.
- The 8-bit fixed-width integer types.
- The fast and least fixed-width types.
- Any compiler-specific fixed-width integers -- for example, Visual Studio defines __int8, __int16, etc…

Best practice for std::size_t

If you use **std::size_t** explicitly in your code, #include one of the headers that defines std::size_t (we recommend **< cstddef >**).

Using sizeof does not require a header (even though it return a value whose type is std::size_t).

bool values

```
#include <iostream>

int main()
{
    std::cout << true << '\n';
    std::cout << false << '\n';

    std::cout << std::boolalpha; // print bools as true or false

    std::cout << true << '\n';
    std::cout << false << '\n';
    return 0;
}
```

This prints:

1
0
true
false

You can use std::noboolalpha to turn it back off.

chars

Name	Symbol	Meaning
Alert	\a	Makes an alert, such as a beep
Backspace	\b	Moves the cursor back one space
Formfeed	\f	Moves the cursor to next logical page
Newline	\n	Moves cursor to next line
Carriage return	\r	Moves cursor to beginning of line
Horizontal tab	\t	Prints a horizontal tab
Vertical tab	\v	Prints a vertical tab
Single quote	\'	Prints a single quote
Double quote	\"	Prints a double quote
Backslash	\\	Prints a backslash.
Question mark	\?	Prints a question mark. No longer relevant. You can use question marks unescaped.
Octal number	\(number)	Translates into char represented by octal
Hex number	\x(number)	Translates into char represented by hex number

static cast

Even though it is called a conversion, a type conversion does not actually change the value or type of the value being converted. Instead, the value to be converted is used as input, and the conversion results in a new value of the target type (via direct initialization).

Explicit type conversion allow us (the programmer) to explicitly tell the compiler to convert a value from one type to another type, and that we take full responsibility for the result of that conversion. If such a conversion results in the loss of value, the compiler will not warn us.

`static_cast<new_type>(expression)`

The **static_cast** operator doesn't do any range checking, so if you cast a value to a type whose range doesn't contain that value, undefined behavior will result. Therefore, the above cast from **unsigned int** to **int** will yield unpredictable results if the value of the unsigned int is greater than the maximum value a **signed int** can hold.

std::int8_t and std::uint8_t likely behave like chars instead of integers