Modern C++ Programming

4. Basic Concepts II

Integral Data Types

Federico Busato

2026-01-06

Table of Contents

Table of Contents

Integral Types

Integral Types

Type	Bytes	Range	Fixed width types `<cstdint>`
`bool`	1	`true`, `false`	
`char` †$	1	implementation defined	
`signed char` $	1	-128 to 127	int8_t
`unsigned char`	1	0 to 255	uint8_t
`short` $	2	-2^{15} to $2^{15}-1$	int16_t
`unsigned short`	2	0 to $2^{16}-1$	uint16_t
`int` $	4	-2^{31} to $2^{31}-1$	int32_t
`unsigned`	4	0 to $2^{32}-1$	uint32_t
`long` $	4*/8		int32_t*/int64_t
`long unsigned`	4*/8		uint32_t*/uint64_t
`long long` $	8	-2^{63} to $2^{63}-1$	int64_t*
`long long unsigned`	8	0 to $2^{64}-1$	uint64_t*

*on Windows 64-bit, †`signed`/`unsigned` from C++11, $two-complement from C++20

Integral Type Names

Signed Type	Short Name
`signed char`	/
`signed short int`	`short`
`signed int`	`int`
`signed long int`	`long`
`signed long long int`	`long long`

Unsigned Type	Short Name
`unsigned char`	/
`unsigned short int`	`unsigned short`
`unsigned int`	`unsigned`
`unsigned long int`	`unsigned long`
`unsigned long long int`	`unsigned long long`

Integral Types - Literal Suffixes

Type	SUFFIX	Example
`int`	/	2
`unsigned`	u, U	3u
`long`	l, L	8L
`long unsigned`	ul, UL	2ul
`long long`	ll, LL	4ll
`long long unsigned`	ull, ULL	7ULL

Integral Types Literals

A **literal** is a fixed value written directly into the source code

Representation	PREFIX	Example
`Decimal`	/	5
`Binary`	0b	0b010101
`Octal`	0	0307
`Hexadecimal`	0x or 0X	0xFFA010

C++14 also allows *digit separators* to improve the readability `1'000'000`

Integral Types Literals

Note: The type of a `literal` depends on its value, starting from `int`. This property doesn't apply to `literal` operations

```
auto v1 = 1;                          // int
auto v2 = 0b10101;                    // int
auto v3 = 0x8;                        // int

auto v3 = 2'000'000'000;              // int
auto v4 = 3'000'000'000;              // unsigned
auto v6 = 32'000'000'000;             // long int
auto v7 = 32 * 1000 * 1000 * 1000;    // int!!

auto v7 = 0x80000000;                 // unsigned
```

Good practice: to always use the `uniform initialization` syntax to prevent errors (see "`Basic Concepts V`" lecture): `int{0x80000000}` → error.

The `<climits >` header is derived from `C` and provides macros that define the *minimum* and *maximum* values for integral types as macros.

Type	Signed Min	Signed Max	Unsigned Max
`char`	`SCHAR_MIN`	`SCHAR_MAX`	`UCHAR_MAX`
`short`	`SHRT_MIN`	`SHRT_MAX`	`USHRT_MAX`
`int`	`INT_MIN`	`INT_MAX`	`UINT_MAX`
`long`	`LONG_MIN`	`LONG_MAX`	`ULONG_MAX`
`long long`	`LLONG_MIN`	`LLONG_MAX`	`ULLONG_MAX`

The C++11 `<limits >` header introduces `std::numeric_limits` for both *integral* and *floating-point* types to provide properties such as minimum, maximum, epsilon, infinity, and others.

```cpp
#include <limits>

std::numeric_limits<int>::max();            // 2^31 − 1
std::numeric_limits<unsigned short>::max(); // 65,535

std::numeric_limits<int>::min();            // −2^31
std::numeric_limits<unsigned short>::min(); // 0
```

* this syntax will be explained in the next lectures

C++ Data Model

C++ Data Model	OS	Number of Bits				
		`short`	`int`	`long`	`long long`	`pointer`
ILP32	Windows/Unix 32-b	16	32	32	64	32
LLP64	Windows 64-bit	16	32	**32**	64	64
LP64	Linux 64-bit	16	32	**64**	64	64

`char` is always 1 byte

LP32: Windows 16-bit APIs (no more used)

`int*_t/uint*_t <cstdint>`

C++11 provides fixed width integer types:

`int8_t`, `int16_t`, `int32_t`,`int64_t`

`uint8_t`, `uint16_t`, `uint32_t`,`uint64_t`

They have the same number of bits on all architecture.

Good practice: Prefer fixed-width integers instead of native types. `int` and `unsigned` can be directly used as they are widely accepted by C++ data models.

`int*_t` types are not "real" types, they are merely *typedefs* to appropriate fundamental types

C++ standard does not ensure a one-to-one mapping:

- There are **five** distinct *fundamental types* (`char` , `short` , `int` , `long` , `long long`)
- There are **four** `int*_t` *overloads* (`int8_t` , `int16_t` , `int32_t` , and `int64_t`)

ithare.com/c-on-using-int_t-as-overload-and-template-parameters

Warning: I/O Stream interprets `uint8_t` and `int8_t` as `char` and not as integer values

```
int8_t var;
cin >> var;   // read '2'
cout << var;  // print '2'
int var2 = var;
cout << var2; // print '100' !!
```

size_t

size_t <cstddef>

`size_t` is an *alias* data type capable of storing the biggest representable value on the current architecture

- `size_t` is an unsigned integer type (of at least 16-bit)
- `size_t` is the return type of `sizeof()` and commonly used to represent size measures
- `size_t` is 4 bytes on 32-bit architectures, and 8 bytes on 64-bit architectures
- C++23 adds `uz` / `UZ` literals for `size_t`, e.g. `5uz`
- C++23 adds `z` / `Z` for the signed version of `size_t`, e.g. `5z`

ptrdiff_t ★

ptrdiff_t <cstddef>

`ptrdiff_t` is an *alias* data type used to store the results of the difference between pointers or iterators

- `ptrdiff_t` is the signed version of `size_t` commonly used for computing pointer differences
- `ptrdiff_t` are 4 bytes on 32-bit architectures, and 8 bytes on 64-bit architectures

`uintptr_t` ★

`uintptr_t <cstdint>`

`uintptr_t` (C++11) is an integer type that can be converted from and to a `void` pointer

- `uintptr_t` is an unsigned type
- `sizeof(uintptr_t) == sizeof(void*)`
- `uintptr_t` is an *optional* type of the standard and compilers may not provide it

Arithmetic Operation Semantics

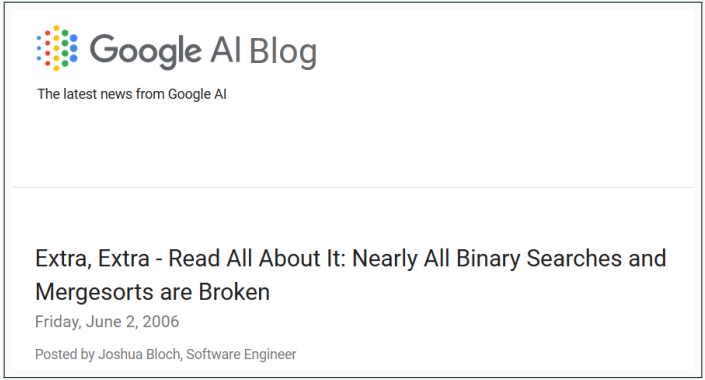
A Firmware Bug

> *"Certain SSDs have a firmware bug causing them to irrecoverably fail after exactly 32,768 hours of operation. SSDs that were put into service at the same time will fail simultaneously, so RAID won't help"*

`HPE SAS Solid State Drives - Critical Firmware Upgrade`

Via `twitter.com/martinkl/status/1202235877520482306?s=19`

Overflow Implementations



Note: Computing the average in the right way is not trivial, see `On finding the average of two unsigned integers without overflow`

related operations: ceiling division, rounding division

`ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html`

Potentially Catastrophic Failure

$$51\ \textit{days} = 51 \cdot 24 \cdot 60 \cdot 60 \cdot 1000 = 4\,406\,400\,000\ \textit{ms}$$

Boeing 787s must be turned off and on every 51 days to prevent ‘misleading data’ being shown to pilots

Arithmetic Operation Semantics

Overflow The result of an arithmetic operation exceeds the word length, namely the largest positive/negative values. It indicates an error in the program logic

Wraparound The result of an arithmetic operation is reduced modulo 2^N where N is the number of bits of the word

Saturation The result of an arithmetic operation is constrained within a fixed range defined by a minimum and maximum value. If the result of an operation exceeds this range, it is *"clamped"* to the boundary value

Signed/Unsigned Integer Characteristics

Without undefined behavior, ***signed*** and ***unsigned*** integers use exactly the same hardware, and they are equivalent at the binary level thanks to the *two-complement* representation.

```
#include <cstdint>
int      a1 = -6;                 // 11111111111111111111111111111010
unsigned b1 = -6;                 // ?

unsigned b2 = UINT_MAX + 1;       // 0
unsigned b3 = (UINT_MAX + 1) - 6; //
unsigned b4 = UINT_MAX -5;        // 11111111111111111111111111111010 (2^32 - 5)
```

However, ***signed*** and ***unsigned*** integers have different semantics in C++. The compiler can exploit undefined behavior to optimize the code, even if such operations are well defined at the hardware level.

Signed Integer

- Represent positive, negative, and zero values ($\mathbb{Z}$)

- ✅ Represent the human intuition of numbers

- *Properties*:
  - Reflexive
  - Commutative
  - Not associative (overflow/underflow) for addition and multiplication, e.g. `(x + y) + z != x + (y + z)`
  - Multiplication, subtraction, division, modulo, and negation can lead to subtle overflow

- Two-complement since C++20
  Before, *legacy platforms* can define them as one's complement or sign-and-magnitude

- ⚠ All bitwise operations are well-defined since C++20, *except shift*

  If signed integers are not two-complement, bitwise operations with negative values have a different behavior depending on the platform, e.g. `-1 != ~0`, `-1 & 1 == 0` with one's complement

- ⚠ *Overflow/underflow semantic* → undefined behavior

  overflow: `INT_MAX + 1`
  underflow: `INT_MIN - 1`

⚠ More negative values ($2^{31} - 1$) than positive ($2^{31} - 2$)

undefined behavior:

- subtraction: `a - b` with `b=INT_MIN`
- multiplication: `a * b` with `a=INT_MIN, b=-1`
- division: `a / b` with `a=INT_MIN, b=-1`
- modulo: `a % b` with `a=INT_MIN, b=-1`
- negation: `-x` with `x=INT_MIN`

⚠ Shift could lead to undefined behavior `x` $\ll$ `y`

- undefined behavior if `y` is larger than the number of bits of `x`
- undefined behavior if `y` is negative
- implementation-defined if `x` is negative (until C++20)

Unsigned Integer

- Represent only *non-negative* values ($\mathbb{N}$)
- *Properties*: commutative, reflexive, associative for all operations
- Discontinuity in 0, $2^{32} - 1$
- ✅ Wraparound semantic → well-defined (modulo 2^{32})
- ✅ All bitwise operations are well-defined, *except shift*
- ⚠️ Shift could lead to undefined behavior `x` $\ll$ `y`
  - undefined behavior if `y` is larger than the number of bits of `x`

Google C++ Style Guide

> Because of historical accident, the C++ standard also uses unsigned integers to represent the size of containers - many members of the standards body believe this to be a mistake, but it is effectively impossible to fix at this point

- C++ core guidelines: Don't use unsigned for subscripts, prefer gsl::index, *Bjarne Stroustrup*, *Herb Sutter*
- Subscripts and sizes should be signed, *Bjarne Stroustrup*
- Don't add to the signed/unsigned mess, *Bjarne Stroustrup*
- Integer Type Selection in C++: in Safe, Secure and Correct Code, *Robert C. Seacord*

Good solution: use `int64_t`

Hard to overflow!

Max value: $2^{63} - 1 = 9{,}223{,}372{,}036{,}854{,}775{,}807$ or
9 quintillion (9 billion of billion),
about 292 years in nanoseconds,
9 million terabytes

When use `signed` integer?

- If the quantity can be mixed with negative values, e.g. subtracting byte sizes
- Prefer expressing strictly positive values with signed integers and assertions
- Optimization purposes, e.g. exploit undefined behavior for overflow or in loops

When use `unsigned` integer?

- If the quantity can never be mixed with negative values (?)
- Bitmask values
- Optimization purposes, e.g. division, modulo
- Safety-critical system, where *undefined behavior* is not acceptable

Detecting Overflow / Underflow

Detecting wraparound for unsigned integral types is **not trivial**

```
// some examples
bool is_add_overflow(unsigned a, unsigned b) {
    return (a + b) < a || (a + b) < b;
}

bool is_mul_overflow(unsigned a, unsigned b) {
    unsigned x = a * b;
    return a != 0 && (x / a) != b;
}
```

Detecting overflow/underflow for signed integral types is even harder and must be checked before performing the operation

Saturation Arithmetic ★

C++26 adds four main functions to perform **saturation arithmetic** with integer types in the `<numeric>` library. In other words, the undefined behavior or the wrap-around behavior for overflow/underflow is replaced by **saturation** values, namely the *minimum* or *maximum* values of the operands

- `T add_sat(T x, T y)`
- `T sub_sat(T x, T y)`
- `T mul_sat(T x, T y)`
- `T div_sat(T x, T y)`
- `R saturate_cast<R>(T x)`

Integer Undefined Behavior

The C++ standard does not prescribe any specific behavior (undefined behavior) for several integer/unsigned arithmetic operations

- *Classical signed integer overflow/underflow*

```
int x = std::numeric_limits<int>::max() + 20; // x=?
if (y < y + 5) // y: int -> can be always true (compiler optmizes it away)
```

- *More negative values than positive*

```
int x = std::numeric_limits<int>::max() * -1; //  (2^31 -1) * -1
cout << x;                                    //  -2^31 +1 ok

int y = std::numeric_limits<int>::min() * -1; // -2^31 * -1
cout << y; // hard to see in complex examples // 2^31 overflow!!
```

```
#include <climits>
#include <cstdio>

void f(int* ptr, int pos) {
    pos++;
    if (pos < 0)    // <-- the compiler could assume that signed overflow never
        return;     //     happen and "simplify" the condition to check
    ptr[pos] = 0;
}

int main() {                // the code compiled with optimizations, e.g. -O3
    int* tmp = new int[10]; // leads to segmentation faults with clang, while
    f(tmp, INT_MAX);        // it terminates correctly with gcc
    printf("%d\n", tmp[0]);
}
```

s/open.c of the Linux kernel

```
int do_fallocate(..., loff_t offset, loff_t len) {
    inode *inode = ...;
    if (offset < 0 || len <= 0)
        return -EINVAL;
    /* Check for wrap through zero too */
    if ((offset + len > inode->i_sb->s_maxbytes) || (offset + len < 0))
        return -EFBIG;      // the compiler is able to infer that both 'offset' and
    ...                     // 'len' are non-negative and can eliminate this check,
}                           // without verify integer overflow
```

Even worse example:

```cpp
#include <iostream>

int main() {
    for (int i = 0; i < 4; ++i)
        std::cout << i * 1000000000 << std::endl;
}
// with optimizations, it is an infinite loop
// --> 1000000000 * i > INT_MAX
// undefined behavior!!

// the compiler translates the multiplication constant into an addition
```

Why does this loop produce undefined behavior?

Is the following loop safe?

```
void f(int size) {
    for (int i = 0; i < size; i += 2)
        ...
}
```

- What happens if `size` is equal to `INT_MAX`?
- How to make the previous loop safe?
- `i >= 0 && i < size` is not the solution because of *undefined behavior* of signed overflow
- Can we generalize the solution when the increment is `i += step`?

Undefined Behavior - Other Cases

- *Bitwise operations* on signed integer types with negative value is undefined behavior

```
int x = -1 << 12; // undefined behavior!!, until C++20
int y = 1 << -12; // undefined behavior!!
int z = 1 &  ~-1; // undefined behavior!!
```

- *Shift* larger than #bits of the data type is undefined behavior even for **unsigned**

```
unsigned x = 1u << 32u; // undefined behavior!!
```

- Division by zero

```
unsigned y = 3 / 0; // undefined behavior!!
```

Undefined Behavior - Division by Zero Example

src/backend/utils/adt/int8.c of PostgreSQL

```
if (arg2 == 0) {
    ereport(ERROR, (errcode(ERRCODE_DIVISION_BY_ZERO), // the compiler is not aware
            errmsg("division by zero")));              // that this function
}                                                      // doesn't return
/* No overflow is possible */
PG_RETURN_INT32((int32) arg1 / arg2);  // the compiler assumes that the divisor is
                                       // non-zero and can move this statement on
                                       // the top  (always executed)
```

Integer Conversion Rules

Conversion Basics

Conversion between *arithmetic types* can occur **implicitly** or **explicitly**, depending on the context

```
void f(unsigned v) {}

int      x  = 5;
float    y  = 2.0;
unsigned u1 = (unsigned) x; // explicit
unsigned u2 = f;            // implicit
f(x);                       // implicit
```

- Conversion to a *smaller integral type* is called **truncation**.
- Conversion to a *larger type* is called **promotion**.

Conversion to an integer value follows *modulo arithmetic*:

$$\texttt{(largest destination value + original value + 1)} \ \% \ 2^N$$

- Conversion between integers of same size *doesn't not affect the binary representation** but their interpretation (no operations).
- Conversion of values in the valid range of the destination type is always well defined, preserving the original value. Promotion preserves the sign.
- Conversion of values outside the destination range still follows modulo arithmetic*.

* unsigned → signed is implementation-defined before C++20 (non-two-complement platforms and negative values)

```
int      a2 = -1;     // 0xFFFFFFFF
unsigned b2 = x;      // UINT_MAX + (-1) + 1 = UINT_MAX (0xFFFFFFFF)

int      a3 = 65537; // 2^16 + 1  (0x10001)
int16_t  b3 = a3;    // x % 2^16 = 1, truncation

int      a4 = 32768; // 2^15
int16_t  b4 = a4;    // (int16_t) (x % 2^16 = 32768 = 0b1000000000000000)
                     //  = -32768 (smallest int16_t)
```

Signed integer promotion could be very dangerous (with negative values):

```
unsigned a5 = 0xFFFFFFFF;
int      b5 = a5;              // b5=0xFFFFFFFF in memory
a5 == b5;                      // true
(int64_t) a5 == (int64_t) b5; // false!!, b5 is subject to sign extend
```

***Implicit* conversion rules**, applied in order, before any operation:

$\otimes$: any arithmetic (`*, +, /, -, %`), comparison (`==, !=, >, <, >=, <=`), and bitwise operation (`%, &, |, ∧`), except shift `«`

(A) Floating point promotion

`floating_type` $\otimes$ `integer_type` $\rightarrow$ `floating_type`

(B) Size promotion

`small_type` $\otimes$ `large_type` $\rightarrow$ `large_type`

(C) Sign promotion

`signed_type` $\otimes$ `unsigned_type` $\rightarrow$ `unsigned_type`

`small_integral_type`: any integral type smaller than `int`

$\odot$: unary `-`, `+`, $\sim$, `++`, `--`

(D) Implicit integer promotion

- `small_integral_type` $\otimes$ `small_integral_type` $\rightarrow$ `int`
- $\odot$ `small_integral_type` $\rightarrow$ `int`

Examples

```
float    f = 1.0f;
double   d = 3.0;
unsigned u = 2;
int      i = 3;
short    s = 4;
uint8_t  c = 5;

f * d; // float × double → double: 3.0f
f * u; // float × unsigned → float: 2.0f
s * c; // short × unsigned char → int: 20
u * i; // unsigned × int → unsigned: 6u
+c;    // unsigned char → int: 5

uint8_t a1 = 255;
uint8_t b1 = 255;
a1 + b1;  // '510'→int (no overflow)
```

(A) Common Integer/Floating Point Misconceptions

Integers are not floating points!

```
int   b = 7;
float a = b / 2;   // a = 3 not 3.5!!
int   c = b / 2.0; // again c = 3 not 3.5!!
```

```
int f(int a, unsigned b, int* array) {  // array is small
    if (a > b)
        return array[a - b]; // ?
    return 0;
}
```

☠ Segmentation fault for `a < 0` !

```
// v.size() returns an unsigned integer
for (size_t i = 0; i < v.size() - 1; i++)
    array[i] = 3; // ?
```

☠ Segmentation fault for `v.size() == 0` !

A real-world case:

```
// --> Allocate a rtvec vector of N elements

rtvec rtvec_alloca(int n) {
    rtvec rt;
    rt = (rtvec) obstack_alloc(
        rtl_obstack,
        sizeof(struct rtvec_def) + ((n - 1) * sizeof(rtunion)));
//  ...
    return rt;
}
```

- Garbage In, Garbage Out: Arguing about Undefined Behavior with Nasal Daemons, Chandler Carruth, CppCon 2016
- MSVC Warning C26839: Array new allocation size is the result of a signed to unsigned narrowing conversion, January 2025!

(D) Implicit Integer Promotion Errors


```
uint8_t  u  = 0;
uint8_t  u1 = ~u;  // 0b11111111 = 255
unsigned u2 = ~u;  // 0b11...111 = 4'294'967'295
```

```
int16_t  a = -1;    // 0xFFFF
uint16_t b = 65535; // 0xFFFF
a * b;              // '-131071', ok
b * b;              // overflow: 4'294'836'225 is not representable with 'int'
                    // undefined behavior!!
```

Again, signed integer promotion could be very dangerous (with negative values):

```
int16_t  c = 0xFFFF; // c=0xFFFF in memory
uint16_t d = c;      // d=0xFFFF in memory
(c == d);            // false!! (implicit)
```

Implicit promotion can also occurs in ternary operator:

```
int16_t  c = 0xFFFF; // c=0xFFFF in memory
uint16_t d = c;      // d=0xFFFF in memory
auto     x = true ? c : d;
sizeof(x) == 4;      // true!!, x is 'int'
```

```
unsigned a = 10;  // array is small
int      b = -1;
array[10ull + a * b] = 0; // ?
```

☠ Segmentation fault!

Considering the following code:

```
unsigned x = 32;      // x can be also a pointer
x         += 2u - 4; // 2u - 4 = 2 + (2^32 - 4)
                     //        = 2^32 - 2
                     // (32 + (2^32 - 2)) % 2^32
cout << x;           // print 30 (as expected)
```

What about the following snippet?

```
uint64_t x = 32;      // x can be also a pointer
x         += 2u - 4;
cout << x;
```

Safe Comparison Functions ★

C++20 introduces a set of functions `<utility>` to safely compare integers of different types (`signed`, `unsigned`)

```
bool cmp_equal(T1 a, T2 b)
bool cmp_not_equal(T1 a, T2 b)
bool cmp_less(T1 a, T2 b)
bool cmp_greater(T1 a, T2 b)
bool cmp_less_equal(T1 a, T2 b)
bool cmp_greater_equal(T1 a, T2 b)
```

example:

```
#include <utility>
unsigned a  = 4;
int      b  = -3;
bool     v1 = (a > b);                // false!!!
bool     v2 = std::cmp_greater(a, b); // true
```


How to compare signed and unsigned integers in C++20?

Arithmetic Operation Special Cases

Logical not `!` converts to boolean

```
int x = 4;
int y = !!x; // y = 1, not 4
```

Bitwise shift `«` result type is the same of the *left operand*

```
uint64_t power_of_two_64bit(uint64_t exponent) {
     return 1 << (exponent % 64);
}

power_of_two_64bit(40); // undefined behavior
```