Fundamental Types Fundamental Types

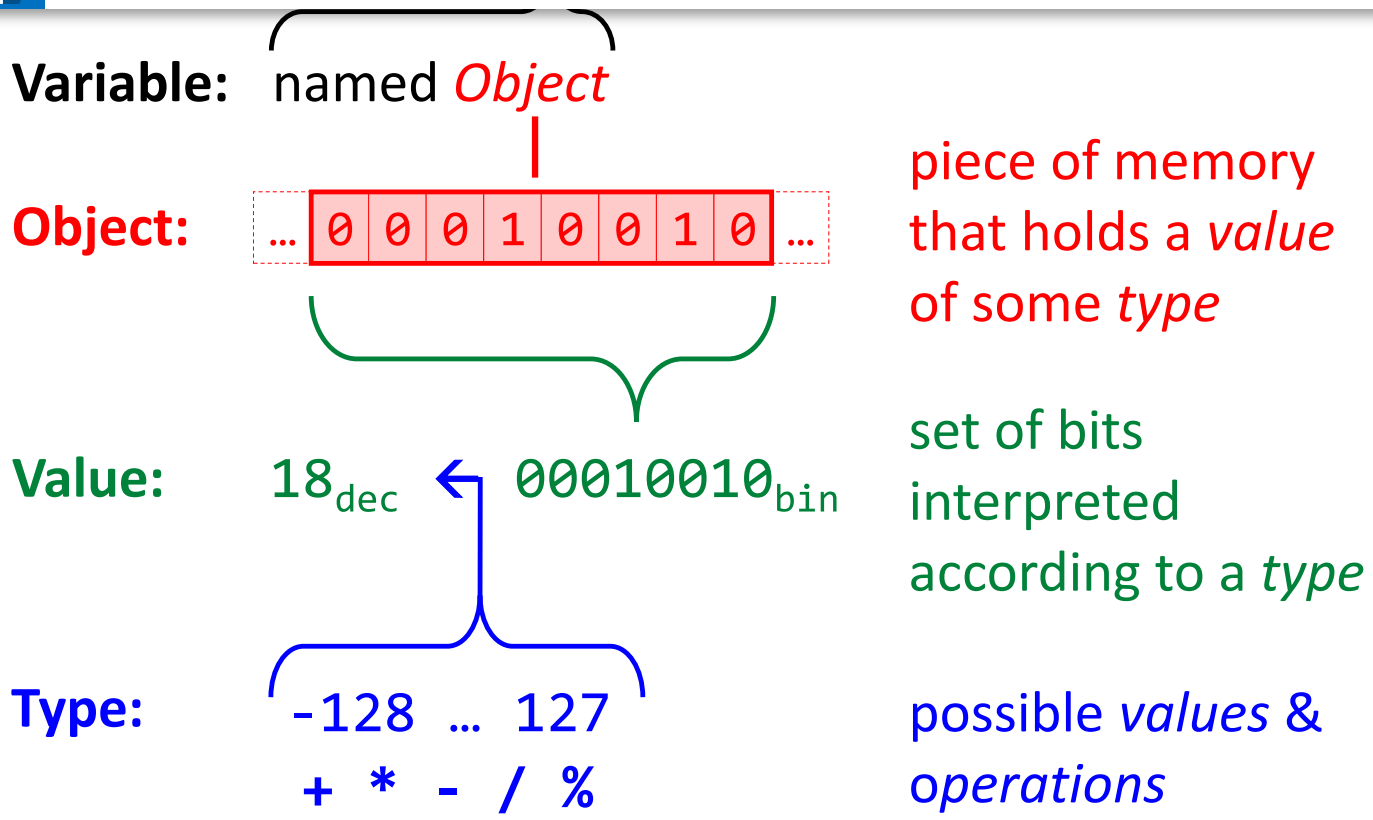
Variables Overview Arithmetic Increment Comparisons Boolean Logic
Memory Sizes numeric_limits Narrowing Bitwise Ops Conversions

Fundamental Types are the basic building blocks for all complex types / data structures like lists, hash maps, trees, graphs, ...

Variable Declarations

```
type variable = value;

type variable {value}; C++11
```

```
// declare & initialize 'i':
int i = 1;
// print i's value:
cout << i << '\n';
int j {5};
cout << j << '\n';
```



⚠ Variables of fundamental types are not initialized by default!

```
int k;  // k not initialized!
cout << k << '\n';  // value might be anything
```

Because in C++ "you only pay for what you use" (initilization of large memory blocks can be quite expensive)

👉 **But:** You should almost always initialize variables when declaring them to prevent bugs!

Quick Overview

Booleans

```
bool b1 = true;
bool b2 = false;
```

Characters

```
char c = 'A';  // character literal
char a = 65;   // same as above
```

Signed Integers

n bits ⇒ values ∈ $[-2^{(n-1)}, 2^{(n-1)}-1]$

```
short s = 7;
int   i = 12347;
long  l1 = -7856974990L;
long long  l2 = 895656569749900LL;

// ' digit separator  C++14
long l3 = 512'232'697'499;
```

Unsigned Integers

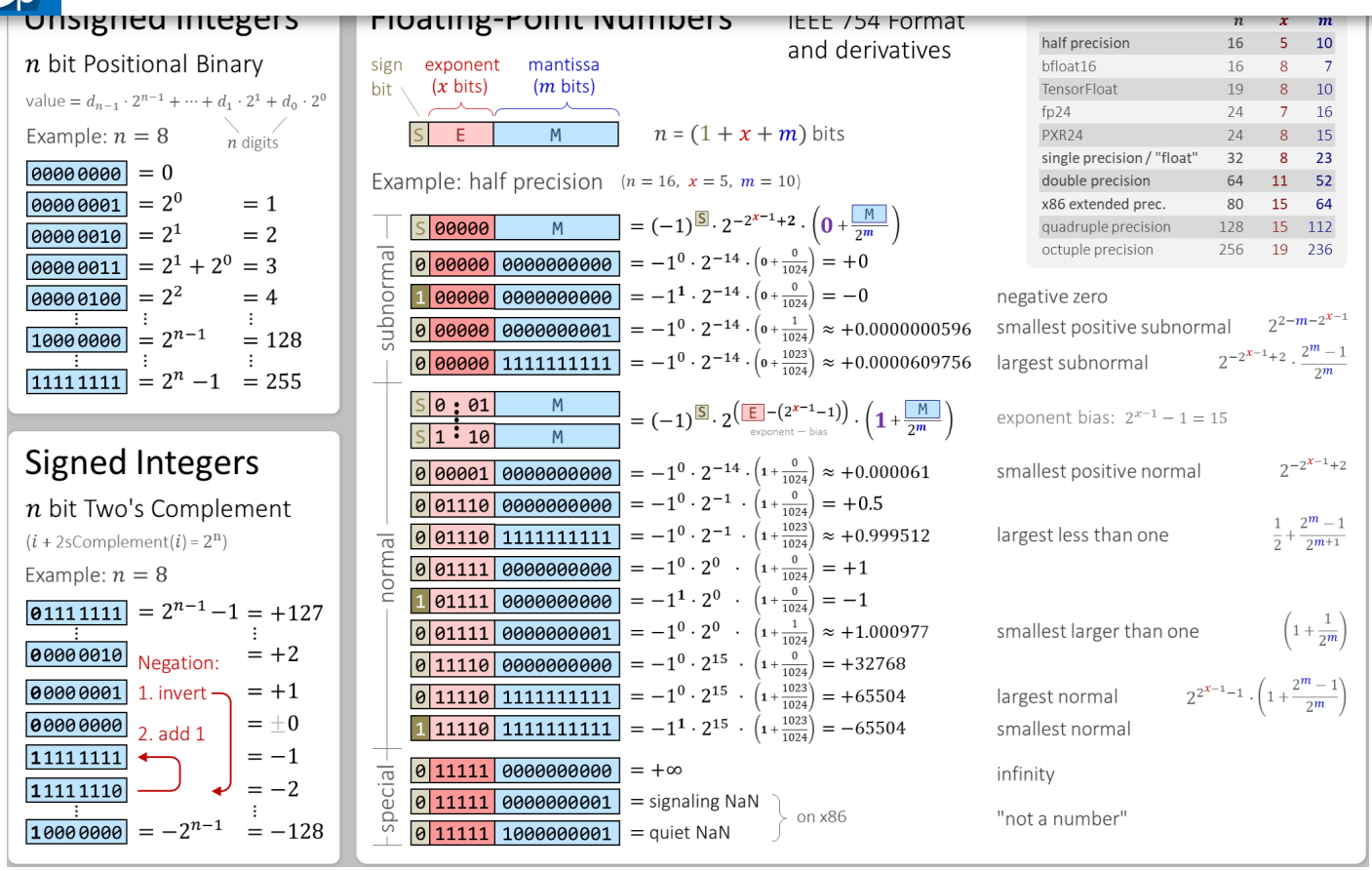
n bits ⇒ values ∈ $[0, 2^{n}-1]$

```
unsigned u1 = 12347U;
unsigned long u2 = 123478912345UL;
unsigned long long u3 = 123478912345ULL;

// non-decimal literals
unsigned x = 0x4A;        // hexadecimal
unsigned b = 0b10110101;  // binary  C++14
```

Floating Point Types

- `float` usually IEEE 754 32 bit
- `double` usually IEEE 754 64 bit
- `long double` usually 80-bit on x86/x86-64

```
float       f  = 1.88f;
double      d1 = 3.5e38;
long double d2 = 3.5e38L;  C++11

// ' digit separator  C++14
double d3 = 512'232'697'499.052;
```



(click for fullscreen view)

Float Toy: interactive visualization of bit → value mapping

Arithmetic Operations

expression `a ⊕ b` returns result of operation ⊕ applied to the values of a and b

expression `a ⊕= b` stores result of operation ⊕ in a

```
int a = 4;
int b = 3;

a = a + b;
a += b;
```

```
variable a is set to value 4
variable b is set to value 3

a: 7    add
a: 10
```

```
a *= b;

a = a / b;
a /= b;

a = a % b;
```

```
a: 36

a: 12   divide
a: 4

a: 1    remainder of division
(modulo)modulo
```

Increment/Decrement

- changes value by +/- 1
- prefix expression `++x` / `--x` returns new (incremented/decremented) value
- postfix expression `x++` / `x--` increments/decrements value, but *returns old value*

```
int a = 4;
int b = 3;

b = a++;
b = ++a;

b = --a;
b = a--;
```

```
a: 4
        b: 3

a: 5    b: 4
a: 6    b: 6

a: 5    b: 5
a: 4    b: 5
```

Comparisons

2-way Comparisons

result of comparison is either ***true*** or ***false***

```
int x = 10;
int y = 5;

bool b1 =  x == 5;
bool b2 = (x != 6);
```

```
result  operator

false   equals
true    not equal
```

3-Way Comparisons With `<=>` C++20

determines the relative ordering of 2 objects:

`(a <=> b) < 0`	if a < b
`(a <=> b) > 0`	if a > b
`(a <=> b) == 0`	if a and b are equal/equivalent

3-way comparisons return a comparison category value that can be compared to literal `0`.

The returned value comes from one of three possible categories: `std::strong_ordering`, `std::weak_ordering` or `std::partial_ordering`.

```
4 <=> 6        →  std::strong_ordering::less
5 <=> 5        →  std::strong_ordering::equal
8 <=> 1        →  std::strong_ordering::greater
```

Boolean Logic

Operators

```
bool a = true;
bool b = false;

bool c = a && b;   // false    logical AND
bool d = a || b;   // true     logical OR
bool e = !a;       // false    logical NOT

Alternative Spellings:
bool x = a and b;  // false
bool y = a or b;   // true
bool z = not a;    // false
```

Conversion to bool

- `0` is always `false`;

```
bool g = 0;    // false  (int → bool)
bool h = 1.2;  // true   (double → bool)
```

Short-circuit Evaluation

The second operand of a boolean comparison is not evaluated if the result is already known after evaluating the first operand.

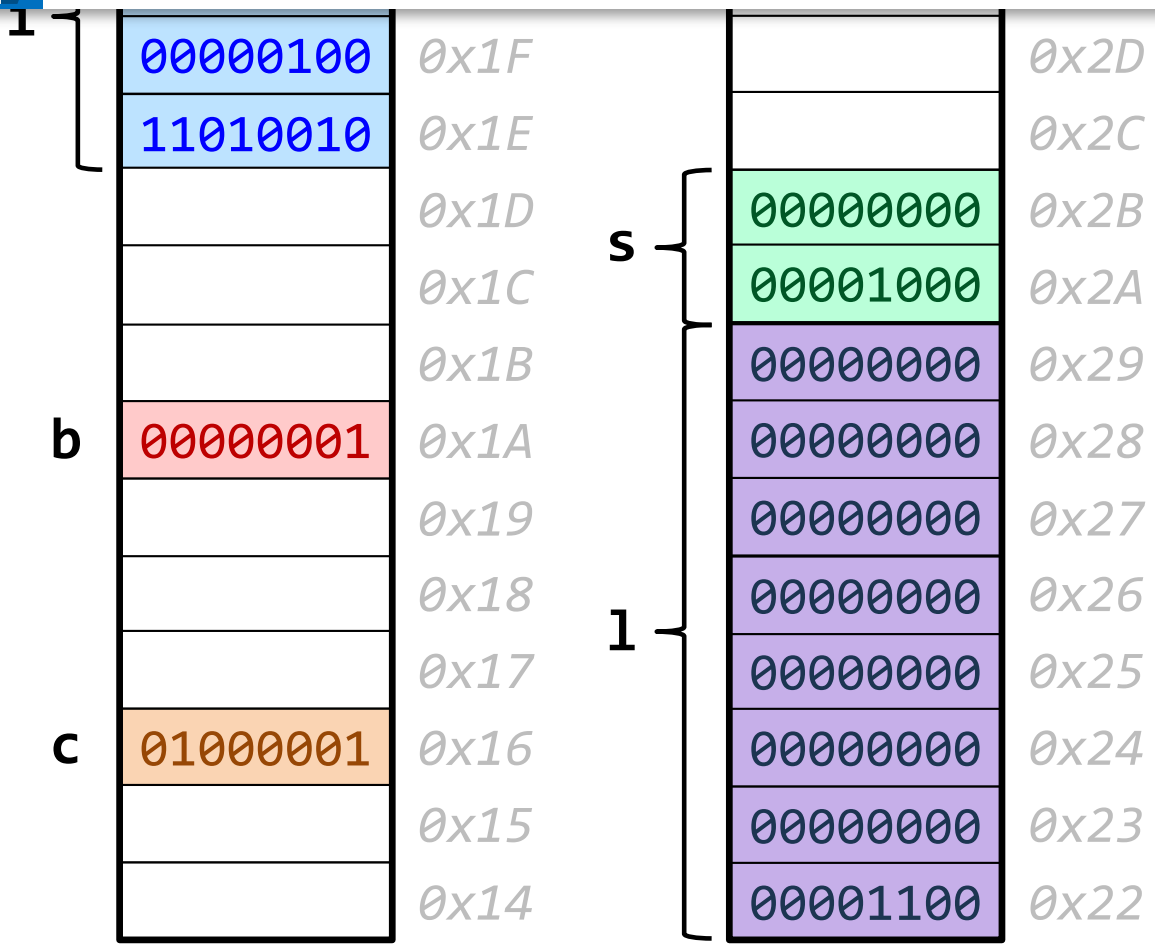
```
int i = 2;
int k = 8;
bool b1 = (i > 0) || (k < 3);
```

`i > 0` is already *true* ⇒ `k < 3` is not evaluated, because the result of logical OR is already *true*

Memory Sizes of Fundamental Types

All type sizes are a multiple of **sizeof**(char)

```
cout << sizeof(char);   // 1
cout << sizeof(bool);   // 1
cout << sizeof(short);  // 2
cout << sizeof(int);    // 4
cout << sizeof(long);   // 8

// number of bits in a char
cout << CHAR_BIT;   // 8

char   c = 'A';
bool   b = true;
int    i = 1234;
long   l = 12;
short  s = 8;
```



⚠ Sizes Are *Platform Dependent*

only basic guarantees

- `sizeof(short)` ≥ `sizeof(char)`
- `sizeof(int)` ≥ `sizeof(short)`
- `sizeof(long)` ≥ `sizeof(int)`

e.g., on some 32-bit platforms: `int` = `long`

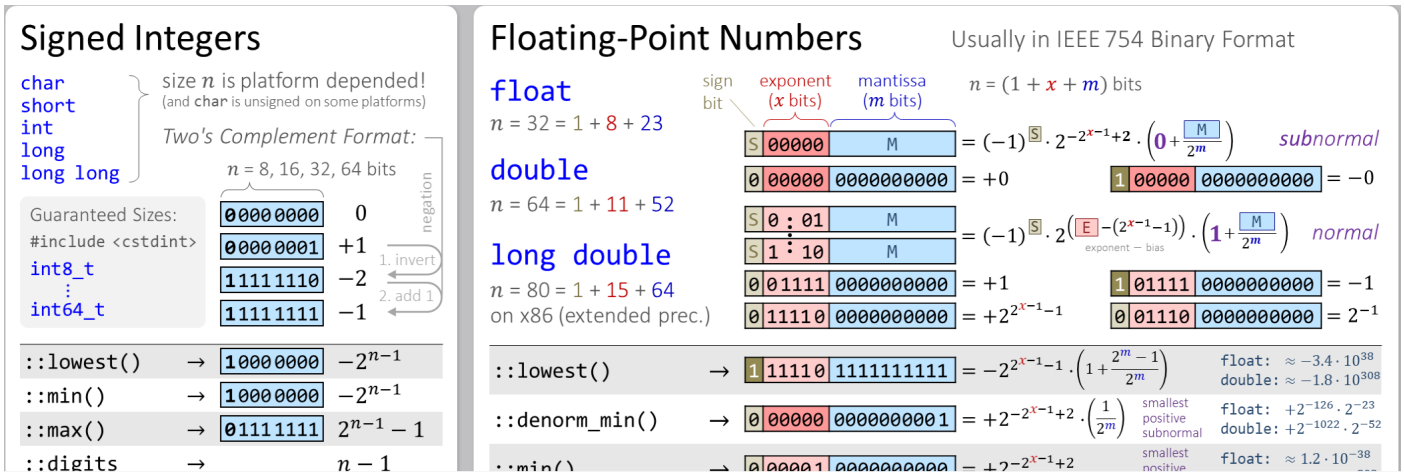
Integer Size Guarantees C++11

```
#include <cstdint>
```

- exact size (not available on some platforms)
  `int8_t`, `int16_t`, `int32_t`, `int64_t`, `uint8_t`, …
- guaranteed minimum size
  `int_least8_t`, `uint_least8_t`, …

std::numeric_limits<type>

```
#include <limits>

// smallest negative value:
cout << std::numeric_limits<double>::lowest();

// float/double: smallest value > 0
// integers: smallest value
cout << std::numeric_limits<double>::min();

// largest positive value:
cout << std::numeric_limits<double>::max();

// smallest difference btw. 1 and next value:
cout << std::numeric_limits<double>::epsilon();
…
```



(click for fullscreen view)

Type Narrowing

- conversion from type that can represent more values to one that can represent less
- may result in loss of information
- in general no compiler warning – happens silently
- potential source of subtle runtime bugs

```
double   d = 1.23456;
float    f = 2.53f;
unsigned u = 120u;

double e = f;  // OK  float → double

int i = 2.5;   // NARROWING double → int
int j = u;     // NARROWING unsigned int → int
int k = f;     // NARROWING float → int
```

Braced Initialization C++11

type variable { *value* };

- works for all fundamental types
- narrowing conversion ⇒ **compiler warning**

```
double   d {1.23456};  // OK
float    f {2.53f};    // OK
unsigned u {120u};     // OK

double e {f};  // OK float → double

int i {2.5};   // ⚠ COMPILER WARNING: double       → int
int j {u};     // ⚠ COMPILER WARNING: unsigned int → int
int k {f};     // ⚠ COMPILER WARNING: float        → int
```

👉 Make sure to prevent silent type conversions, especially narrowing unsigned to signed integer conversions — they can cause hard-to-find runtime bugs!

Bitwise Logic

`a & b`	bitwise AND
`a | b`	bitwise OR
`a ^ b`	bitwise XOR
`~a`	bitwise NOT (one's complement)

```
#include <cstdint>

std::uint8_t a = 6;
std::uint8_t b = 0b00001011;

std::uint8_t c1 = (a & b);  // 2
std::uint8_t c2 = (a | b);  // 15
std::uint8_t c3 = (a ^ b);  // 13

std::uint8_t c4 = ~a;       // 249
std::uint8_t c5 = ~b;       // 244

// test if int is even/odd:
bool a_odd  = a & 1;
bool a_even = !(a & 1);
```

```
memory bits:

0000 0110
0000 1011

0000 0010
0000 1111
0000 1101

1111 1001
1111 0100

result:
0 ⇒ false
1 ⇒ true
```

Bitwise Shifts

Arithmetic Conversions & Promotions

Unfortunately, there's a lot of rules (that go back to C) whose purpose is to determine a common type for both operands and the result of a binary operation

`Operand A` ⊕ `Operand B`

A Simplified Summary

Operations Involving At Least 1 Floating-Point Type

- `long double` ⊕ any other type → `long double`
- `double` ⊕ `float` → `double`

Operations On 2 Integer Types

1. **integer promotion** is first applied to *both* operands:
   basically everything smaller than **`int`** gets promoted to either **`int`** or **`unsiged int`**
   (depending on which can represent all values of the unpromoted type)
2. **integer conversion** is then applied if both operand types are different
   - both signed: smaller type converted to larger
   - both unsigned: smaller type converted to larger
   - signed ⊕ unsigned:
     1. signed converted to unsigned if both have same width
     2. otherwise unsigned converted to signed if that can represent all values
     3. otherwise both converted to unsigned

Cheat Sheet With All Rules

← I/O Basics Beginner's Guide / First Steps vector Intro →

Related

- cppreference: Fundamental Types
- cppreference: Fixed Width Integer Types (`cstdint`)
- cppreference: Arithmetic Operators
- cppreference: Operator Precedence
- cppreference: Numeric Limits

- The Usual Arithmetic Confusions (by Shafik Yaghmour)

Last updated: 2022-02-20

Comments

Tags

algorithms allocators arrays article beginner-level blogs books build-systems C++ C++-standardization C++11 C++14 C++17 C++20 C++98 C-style C-vs-C++ casts classes code-editors code-formatters code-trans command-line community comparisons compilers concepts conferences const constexpr containers control-flow CUDA custom-types data-structures debugging design diagnostics exceptions file-io find find_if functional-prog functions gallery generic-prog groups guide guidelines hash-map hash-set hashing header-files heap ides idiom initialization input io iostreams iterators lambda language-mechanism language-references learning libraries library linker list low-level map memory modern-C++ move-semantics news oop organizations output package-manager paradigm pattern people performance podcasts pointers preprocessor profiling Python randomness ranges recipe references set social-media stack standardization std-algorithms std-containers std-library std-macros std-vector STL strings style taste templates testing toolchain tools traversal types user version-control views VIM VIM-plugins warnings websites