C++ Features You Might Not Know

Jonathan Müller — `@foonathan`

[] is commutative

```
int array[SIZE];
```

```
array[17] = 42;
```

[] is commutative

```
int array[SIZE];
```

```
array[17] = 42;
```

```
*(array + 17) = 42;
```

[] is commutative

```
int array[SIZE];
```

```
array[17] = 42;
```

```
*(array + 17) = 42;
```

```
*(17 + array) = 42;
```

[] is commutative

```
int array[SIZE];
```

```
array[17] = 42;
```

```
*(array + 17) = 42;
```

```
*(17 + array) = 42;
```

```
17[array] = 42;
```

[] is commutative

```
int array[SIZE];
```

```
array[17] = 42;
```

```
*(array + 17) = 42;
```

```
*(17 + array) = 42;
```

```
17[array] = 42;
```

```
std::array<int, SIZE> array;
```

```
array[17] = 42;
```

[ ] is commutative

```
int array[SIZE];
```

```
array[17] = 42;
```

```
*(array + 17) = 42;
```

```
*(17 + array) = 42;
```

```
17[array] = 42;
```

```
std::array<int, SIZE> array;
```

```
array[17] = 42;
```

```
array.operator[](17) = 42;
```

[] is commutative

```
int array[SIZE];
```

```
array[17] = 42;
```

```
*(array + 17) = 42;
```

```
*(17 + array) = 42;
```

```
17[array] = 42;
```

```
std::array<int, SIZE> array;
```

```
array[17] = 42;
```

```
array.operator[](17) = 42;
```

```
17[array] = 42; // compiler error :(
```

think-cell

Unary +

```
int a = 1;
int b = -1;
```

Unary +

```cpp
int a = +1;
int b = -1;
```

Unary +

[expr.unary.op]/7

> *The operand of the unary + operator shall have arithmetic, unscoped enumeration, or pointer type and the result is the value of the argument.* ***Integral promotion is performed on integral or enumeration operands.*** *The type of the result is the type of the promoted operand.*

Unary +

[expr.unary.op]/7

> *The operand of the unary + operator shall have arithmetic, unscoped enumeration, or pointer type and the result is the value of the argument.* ***Integral promotion is performed on integral or enumeration operands.*** *The type of the result is the type of the promoted operand.*

```
unsigned short s;
+s; // int
```

Unary +

[expr.unary.op]/7

> *The operand of the unary + operator shall have arithmetic, unscoped enumeration, or pointer type and the result is the value of the argument.* ***Integral promotion is performed on integral or enumeration operands.*** *The type of the result is the type of the promoted operand.*

```cpp
unsigned short s;
+s; // int
```

```cpp
enum foo : int { a, b, c };
+a; // int
```

Unary +

[expr.unary.op]/7

> *The operand of the unary + operator shall have arithmetic, unscoped enumeration, or pointer type and the result is the value of the argument.* ***Integral promotion is performed on integral or enumeration operands.*** *The type of the result is the type of the promoted operand.*

```cpp
unsigned short s;
+s; // int
```

```cpp
enum foo : int { a, b, c };
+a; // int
```

```cpp
int array[17];
+array; // int*
```

Unary +

[expr.unary.op]/7

> *The operand of the unary + operator shall have arithmetic, unscoped enumeration, or pointer type and the result is the value of the argument.* ***Integral promotion is performed on integral or enumeration operands.*** *The type of the result is the type of the promoted operand.*

```cpp
unsigned short s;
+s; // int
```

```cpp
enum foo : int { a, b, c };
+a; // int
```

```cpp
int array[17];
+array; // int*
```

```cpp
+[]{}; // void(*)(void)
```

Unary +

Use cases for unary '+':

- Convert an unscoped `enum` to its underlying type ... but there's `std::to_underlying` for that

Unary +

Use cases for unary ‘+‘:

- Convert an unscoped `enum` to its underlying type … but there’s `std::to_underlying` for that
- Convert an array to pointer … but that’s implicit

Unary +

Use cases for unary ‘+‘:

- Convert an unscoped `enum` to its underlying type … but there’s `std::to_underlying` for that
- Convert an array to pointer … but that’s implicit
- Convert a lambda to function pointer

think-cell

Unary +

Use cases for unary ‘+‘:

- Convert an unscoped `enum` to its underlying type … but there’s `std::to_underlying` for that
- Convert an array to pointer … but that’s implicit
- Convert a lambda to function pointer

```
template <int (*Fn)(int)>
struct foo {};

foo<+[](int i) { return 2 * i; }> f;
```

C++11.

Unary +

Use cases for unary ‘+‘:

- Convert an unscoped `enum` to its underlying type … but there’s `std::to_underlying` for that
- Convert an array to pointer … but that’s implicit
- Convert a lambda to function pointer

```
template <auto Fn>
struct foo {};

foo<[](int i) { return 2 * i; }> f;
```

C++20.

, Operator

Obligatory example:

```
template <typename I>
void reverse(I begin, I end)
{
    for (auto left = begin, right = std::prev(end);
         left < right; ++left, --right)
        std::iter_swap(left, right);
}
```

, Operator

[expr.comma]/1

> *A pair of expressions separated by a comma is evaluated left-to-right; **the left expression is a discarded-value expression**. The left expression is sequenced before the right expression ([intro.execution]). The type and value of the result are the type and **value of the right operand**; the result is of the same value category as its right operand, and is a bit-field if its right operand is a bit-field.*

, Operator

```cpp
template <typename Fn, typename ... Ts>
void for_each_pack(Fn fn, const Ts&... ts)
{
    (fn(ts), ....);
}
```

More fold expression tricks: foonathan.net/2020/05/fold-tricks/

Normal operator overloading

```
operator=
```

Normal operator overloading

```
operator=
```

```
operator==
operator!= // not required in C++20!
```

Normal operator overloading

```
operator=
```

```
operator==
operator!= // not required in C++20!
```

```
operator<=>
```

Normal operator overloading

```
operator=
```

```
operator==
operator!= // not required in C++20!
```

```
operator<=>
```

```
operator*
operator->
```

Normal operator overloading

```
operator=
```

```
operator==
operator!= // not required in C++20!
```

```
operator<=>
```

```
operator*
operator->
```

```
operator+
operator-
operator*
operator/
```

Unusual operator overloading

```
struct my_bool { … };

my_bool operator&&(my_bool lhs, my_bool rhs);
my_bool operator||(my_bool lhs, my_bool rhs);
```

Unusual operator overloading

```
struct my_bool { … };

my_bool operator&&(my_bool lhs, my_bool rhs);
my_bool operator||(my_bool lhs, my_bool rhs);
```

Warning: No short-circuit!

Unusual operator overloading

```
struct my_bool { … };

my_bool operator&&(my_bool lhs, my_bool rhs);
my_bool operator||(my_bool lhs, my_bool rhs);
```

Warning: No short-circuit!

… **before C++17!**

Unusual operator overloading

```cpp
struct my_iterator { … };

my_iterator operator,(const auto&, my_iterator iter)
{
    std::puts("Hello from comma!");
    return iter;
}

const auto& operator,(my_iterator, const auto& left)
{
    std::puts("Hello from comma!");
    return left;
}
```

Unusual operator overloading

```
A operator->*(B, C);
```

Unusual operator overloading

```
A operator->*(B, C);
```

What is ->*?

```
auto mem_ptr = &Class::member;
std::cout << (object.*mem_ptr) << '\n';
std::cout << (ptr->*mem_ptr) << '\n';
```

Unusual operator overloading

```
A operator->*(B, C);
```

What is ->*?

```
auto mem_ptr = &Class::member;
std::cout << (object.*mem_ptr) << '\n';
std::cout << (ptr->*mem_ptr) << '\n';
```

```
auto smart_ptr = std::make_unique<Class>(object);
std::cout << (smart_ptr->*mem_ptr) << '\n'; // error, no overloaded operator->*
```

Unusual operator overloading

```
template <typename Fn>
struct scope_exit_impl : Fn {
    ~scope_exit_impl() {
        (*this)();
    }
};

#define tc_scope_exit(...) \
    auto TC_UNIQUE_IDENTIFIER = tc::scope_exit([&]{ __VA_ARGS__ })
```

Unusual operator overloading

```
template <typename Fn>
struct scope_exit_impl : Fn {
    ~scope_exit_impl() {
        (*this)();
    }
};

#define tc_scope_exit(...) \
    auto TC_UNIQUE_IDENTIFIER = tc::scope_exit([&]{ __VA_ARGS__ })
```

```
auto hfile = …;
tc_scope_exit(CloseHandle(hfile););
```

Unusual operator overloading

Ideally:

```
auto hfile = …;
tc_scope_exit { CloseHandle(hfile); };
```

Unusual operator overloading

Ideally:

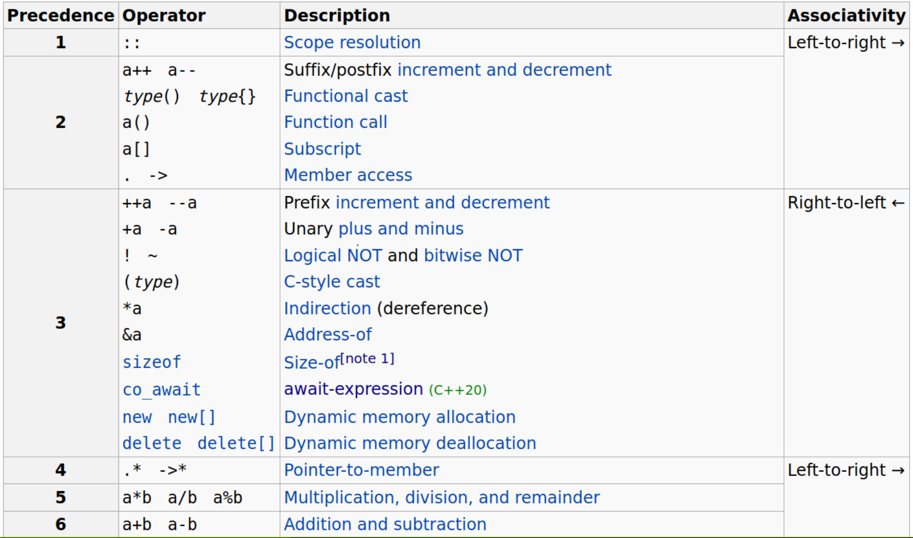
```
auto hfile = …;
tc_scope_exit { CloseHandle(hfile); };
```

```
auto hfile = …;
  auto guard = tc::make_scope_exit_impl{} ??? [&] { CloseHandle(hfile); }
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

`???` must be an overloadable binary operator with high precedence.

Unusual operator overloading

https://en.cppreference.com/w/cpp/language/operator_precedence



Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right →

2	`a++ a--`
	type`()` *type*`{}`
	`a()`
	`a[]`
	`. ->`
		Suffix/postfix increment and decrement
		Functional cast
		Function call
		Subscript
		Member access	

3	`++a --a`
	`+a -a`
	`! ~`
	`(`*type*`)`
	`*a`
	`&a`
	`sizeof`
	`co_await`
	`new new[]`
	`delete delete[]`
		Prefix increment and decrement
		Unary plus and minus
		Logical NOT and bitwise NOT
		C-style cast
		Indirection (dereference)
		Address-of
		Size-of[note 1]
		await-expression (C++20)
		Dynamic memory allocation
		Dynamic memory deallocation
			Right-to-left ←

4	`.* ->*`	Pointer-to-member	Left-to-right →
5	`a*b a/b a%b`	Multiplication, division, and remainder	
6	`a+b a-b`	Addition and subtraction	

Unusual operator overloading

```
template <typename Fn>
struct scope_exit_impl { … };

struct make_scope_exit_impl {
    template <typename Fn>
    auto operator->*(Fn const& fn) const {
        return scope_exit_impl(fn);
    }
};

#define tc_scope_exit \
    auto TC_UNIQUE_IDENTIFIER = tc::make_scope_exit_impl{} ->* [&]
```

`else if` doesn’t exist

`else if` doesn’t exist

`else if` doesn't exist

[stmt.select.general]/1

```
if constexpr? ( init-statement? condition ) statement
if constexpr? ( init-statement? condition ) statement else statement
```

else if doesn't exist

```
if (a) {
    …
} else if (b) {
    …
} else {
    …
}
```

`else if` doesn't exist

```
if (a) {
    …
} else if (b) {
    …
} else {
    …
}
```

```
if (a) {
    …
} else { if (b) {
  …
} else {
  …
} }
```

else if doesn't exist

```
bool is_beautiful(std::optional<color> color)
{
    if (!color)
        return false; // lack of color is not beautiful
    else switch (*color) {
        case red:
        case blue:
        case yellow:
            return true;
        default:
            return false;
    }
}
```

else if doesn't exist

```
bool is_beautiful(std::optional<color> color)
{
    if (!color)
        return false; // lack of color is not beautiful
    else switch (*color) {
        case red:
        case blue:
        case yellow:
            return true;
        default:
            return false;
    }
}
```

Who needs pattern matching?!

`else if` doesn't exist

```cpp
bool is_beautiful(std::optional<color> color)
{
    if (!color)
        return false; // lack of color is not beautiful
    else switch (*color) {
        case red:
        case blue:
        case yellow:
            return true;
        default:
            return false;
    }
}
```

Who needs pattern matching?! (We all do. Desperately.)

switch

```
switch (i)
{
case 1:
case 2:
case 3:
    std::puts("i was 1, 2, or 3");
    break;

default:
    std::puts("i was something else");
    break;
}
```

switch

jMGrKbMKh

```
switch (i)
{
default:
    std::puts("i was something else");
    break;

case 1:
case 2:
case 3:
    std::puts("i was 1, 2, or 3");
    break;
}
```

think-cell

switch

zb7cGGfze

```
switch (i)
{
    std::puts("I'm never executed");

case 1:
case 2:
case 3:
    std::puts("i was 1, 2, or 3");
    break;

default:
    std::puts("i was something else");
    break;
}
```

switch

sK3rKq1s6

```
switch (i)
    case 1:
    case 2:
    case 3:
        std::puts("i was 1, 2, or 3");

std::puts("after the switch");
```

switch

Prd8bT5Gd

```
switch (i)
    if (i == 0)
    {
        std::puts("I'm never executed");
    }
    else
    {
case 0:
        std::puts("i is zero");
    }
```

Duff’s Device

raz65nTva

```
auto n = (count + 7) % 8;
switch (count % 8)
  do
  {
case 0: *to = *from++;
case 7: *to = *from++;
case 6: *to = *from++;
case 5: *to = *from++;
case 4: *to = *from++;
case 3: *to = *from++;
case 2: *to = *from++;
case 1: *to = *from++;
  } while (--n > 0);
```

switch_no_default

4caKMf78x

```
#define switch_no_default(...) \
    switch( __VA_ARGS__ ) \
    default: \
        if (true) assert(!"missing switch case"); \
        else
```

```
switch_no_default (i)
{
case 1:
case 2:
case 3:
    std::puts("i was 1, 2, or 3");
    break;
}
```

Floating point environment

Integer:

`std::int32_t`, `std::int_least32_t`, `std::int_fast32_t`

Floating point environment

Integer:

`std::int32_t`, `std::int_least32_t`, `std::int_fast32_t`

Floats:

`std::float_t`, `std::double_t`

Floating point environment

Rounding:

- **`FE_DOWNWARD`:** towards negative infinity (`2.3` → `2`, `-2.3` → `-3`)

Floating point environment

Rounding:

- **`FE_DOWNWARD`:** towards negative infinity (`2.3` → `2`, `-2.3` → `-3`)
- **`FE_UPWARD`:** towards positive infinity (`2.3` → `3`, `-2.3` → `-2`)

Floating point environment

Rounding:

- **`FE_DOWNWARD`:** towards negative infinity (`2.3` → 2, `-2.3` → `-3`)
- **`FE_UPWARD`:** towards positive infinity (`2.3` → `3`, `-2.3` → `-2`)
- **`FE_TOWARDZERO`:** towards zero (`2.3` → `2`, `-2.3` → `-2`)

think-cell

Floating point environment

Rounding:

- **`FE_DOWNWARD`:** towards negative infinity (`2.3` → 2, `-2.3` → `-3`)
- **`FE_UPWARD`:** towards positive infinity (`2.3` → 3, `-2.3` → `-2`)
- **`FE_TOWARDZERO`:** towards zero (`2.3` → 2, `-2.3` → `-2`)
- **`FE_TONEAREST`:** to nearest value (`2.3` → 2, `2.7` → 3)

think-cell

Floating point environment

Rounding:

- **`FE_DOWNWARD`:** towards negative infinity (`2.3` → 2, `-2.3` → `-3`)
- **`FE_UPWARD`:** towards positive infinity (`2.3` → 3, `-2.3` → `-2`)
- **`FE_TOWARDZERO`:** towards zero (`2.3` → 2, `-2.3` → `-2`)
- **`FE_TONEAREST`:** to nearest value (`2.3` → 2, `2.7` → 3)

`std::fesetround` set current rounding mode

Floating point environment

Integer rounding functions:

- **`std::floor`:** towards negative infinity
- **`std::ceil`:** towards positive infinity
- **`std::trunc`:** towards zero
- **`std::round`:** to nearest integer

Floating point environment

Integer rounding functions:

- **`std::floor`:** towards negative infinity
- **`std::ceil`:** towards positive infinity
- **`std::trunc`:** towards zero
- **`std::round`:** to nearest integer

`std::nearbyint` use current rounding mode

Floating point environment

sodsTd7Wd

```
std::printf("%f\n", std::round(2.5));

std::fesetround(FE_TONEAREST);
std::printf("%f\n", std::nearbyint(2.5));
```

Floating point environment

sodsTd7Wd

```
std::printf("%f\n", std::round(2.5));

std::fesetround(FE_TONEAREST);
std::printf("%f\n", std::nearbyint(2.5));
```

```
3.000000
2.000000
```

Floating point environment

Floating point exceptions:

- **`FE_DIVBYZERO`:** division by zero
- **`FE_INEXACT`:** result needed to be rounded
- **`FE_INVALID`:** domain error (`sqrt(-1)`)
- **`FE_OVERFLOW`:** too large
- **`FE_UNDERFLOw`:** too close to zero

think-cell

Floating point environment

Floating point exceptions:

- **`FE_DIVBYZERO`:** division by zero
- **`FE_INEXACT`:** result needed to be rounded
- **`FE_INVALID`:** domain error (`sqrt(-1)`)
- **`FE_OVERFLOW`:** too large
- **`FE_UNDERFLOw`:** too close to zero

`std::feraiseexcept` raise floating point exception manually

Floating point environment

Floating point exceptions:

- **`FE_DIVBYZERO`:** division by zero
- **`FE_INEXACT`:** result needed to be rounded
- **`FE_INVALID`:** domain error (`sqrt(-1)`)
- **`FE_OVERFLOW`:** too large
- **`FE_UNDERFLOW`:** too close to zero

`std::feraiseexcept` raise floating point exception manually

`std::fetestexcept` test whether exception was raised

Floating point environment

Gq8PPE985

```
std::feclearexcept(FE_ALL_EXCEPT);
std::printf("%f\n", 1 / x);
std::printf("%s\n",
    std::fetestexcept(FE_DIVBYZERO) ? "division by zero" : "okay");
```

think-cell

Floating point environment

Gq8PPE985

```
std::feclearexcept(FE_ALL_EXCEPT);
std::printf("%f\n", 1 / x);
std::printf("%s\n",
    std::fetestexcept(FE_DIVBYZERO) ? "division by zero" : "okay");
```

```
inf
division by zero
```

Floating point environment

Gd3PqYKWG

```
std::feclearexcept(FE_ALL_EXCEPT);
std::printf("%f\n", 0 / x);
std::printf("%s\n",
    std::fetestexcept(FE_DIVBYZERO) ? "division by zero" : "okay");
```

Floating point environment

Gd3PqYKWG

```cpp
std::feclearexcept(FE_ALL_EXCEPT);
std::printf("%f\n", 0 / x);
std::printf("%s\n",
    std::fetestexcept(FE_DIVBYZERO) ? "division by zero" : "okay");
```

```
-nan
division by zero
```

Floating point environment

`NaN`, `-NaN`

Floating point environment

`NaN`, `-NaN`

```
s111 1111 1xxx xxxx xxxx xxxx xxxx xxxx
```

Floating point environment

`NaN`, `-NaN`

```
s111 1111 1xxx xxxx xxxx xxxx xxxx xxxx
```

`16'777'216` different NaN values of a `float`!

Floating point environment

`NaN`, `-NaN`

`s111 1111 1xxx xxxx xxxx xxxx xxxx xxxx`

`16'777'216` different NaN values of a `float`!

```
namespace std
{
    double nan(const char* payload);
}
```

Floating point environment

`NaN`, `-NaN`

`s111 1111 1xxx xxxx xxxx xxxx xxxx xxxx`

`16'777'216` different NaN values of a `float`!

```
namespace std
{
    double nan(const char* payload);
}
```

NaN boxing: piotrduperas.com/posts/nan-boxing

think-cell

Declaration specifier ordering

```
const int a;
```

```
int const a;
```

Declaration specifier ordering

```
const int a;
```

```
constexpr explicit b(…);
```

```
int const a;
```

```
explicit constexpr b(…);
```

Declaration specifier ordering

```
const int a;
```

```
constexpr explicit b(…);
```

```
unsigned int c;
```

```
int const a;
```

```
explicit constexpr b(…);
```

```
int unsigned c;
```

Declaration specifier ordering

cs4bKcz3x

```
decl-specifier-seq:
    decl-specifier
    decl-specifier decl-specifier-seq
```

Declaration specifier ordering

cs4bKcz3x

```
decl-specifier-seq:
    decl-specifier
    decl-specifier decl-specifier-seq
```

```cpp
int typedef a;
```

Declaration specifier ordering

cs4bKcz3x

```
decl-specifier-seq:
    decl-specifier
    decl-specifier decl-specifier-seq
```

```cpp
int typedef a;
```

```cpp
volatile inline float static b;
```

Declaration specifier ordering

cs4bKcz3x

```
decl-specifier-seq:
    decl-specifier
    decl-specifier decl-specifier-seq
```

```
int typedef a;
```

```
volatile inline float static b;
```

```
int constexpr c;
```

Declaration specifier ordering

cs4bKcz3x

```
decl-specifier-seq:
    decl-specifier
    decl-specifier decl-specifier-seq
```

```cpp
int typedef a;
```

```cpp
volatile inline float static b;
```

```cpp
int constexpr c;
```

```cpp
long thread_local unsigned extern long d;
```

Declaration specifier ordering

Guideline?

Sort declaration specifiers alphabetically.

Multiple declarations in single line

```cpp
int a, b;
```

Multiple declarations in single line

```
int a, b, *c;
```

Multiple declarations in single line

```
int a, b, *c;
```

```
int* a, b;
```

Multiple declarations in single line

```cpp
int a, b, *c, d = 42;
```

Multiple declarations in single line

```cpp
int a, b, *c, d = 42, e();
```

Multiple declarations in single line

```cpp
int a, b, *c, d = 42, e(), f(int arg);
```

Multiple declarations in single line

```cpp
int a, b, *c, d = 42, e(), f(int arg), (*g(float arg))(int* arg);
```

Function pointer syntax

Variable:

```
int (*ptr)(int);
```

Function pointer syntax

Variable:

```
int (*ptr)(int);
```

Function return type:

```
int (*f(int))(int);
```

Function pointer syntax

Variable:

```
int (*ptr)(int);
```

Function return type:

```
int (*f(int))(int);
```

Array:

```
int (*array[10])(int);
```

Function pointer syntax

Conversion operator:

```
struct lambda
{
    operator int(*)(int) ();
};
```

Function pointer syntax

Conversion operator:

```
struct lambda
{
    int (*operator())(int);
};
```

Function pointer syntax

Conversion operator:

```
struct lambda
{
    operator int(*())(int);
};
```

Function pointer syntax

Conversion operator:

```
struct lambda
{
    (*operator int())(int);
};
```

Function pointer syntax

Conversion operator:

```
struct lambda
{
    operator auto();
};
```

Local global declarations

ea1jzq3Pv

```
extern int global;
void g();

void f()
{
    ++global;
    g();
}
```

Local global declarations

ea1jzq3Pv

```
void f()
{
    extern int global;
    void g();

    ++global;
    g();
}
```

static in C has only a single meaning

```c
static int file_local = 42;

void f()
{
    ++file_local;
}
```

```c
void f()
{
    static int file_local = 42;
    ++file_local;
}
```

Only difference: visibility of `file_local`.

Function try blocks

```
int main()
{
    try
    {
        …
    }
    catch (std::exception& ex)
    {
        std::cerr << "Error: " << ex.what() << '\n';
        return 1;
    }
}
```

Function try blocks

```
int main() try
{
    …
}
catch (std::exception& ex)
{
    std::cerr << "Error: " << ex.what() << '\n';
    return 1;
}
```

Function try blocks

```cpp
class foo
{
public:
    foo()
    : member(make_member()) // may throw
    {}
};
```

Function try blocks

```
class foo
{
public:
    foo() try
    : member(make_member())
    {}
    catch (...)
    {
        // Handle exception.
    }
};
```

struct vs. class

```cpp
struct foo {};
```

- Member `public` by default
- Base classes `public` by default

```cpp
class foo {};
```

- Member `private` by default
- Base classes `private` by default

struct vs. class

```
enum class foo
{
    a,
    b,
    c
};
```

struct vs. class

```cpp
enum class foo
{
    a,
    b,
    c
};
```

```cpp
enum struct foo
{
    a,
    b,
    c,
};
```

Aside: `using enum`

```
const char* to_string(foo f)
{
    switch (f)
    {

    case foo::a:
        return "a";
    case foo::b:
        return "b";
    case foo::c:
        return "c";
    }
}
```

Aside: `using enum`

```cpp
const char* to_string(foo f)
{
    switch (f)
    {

    case foo::a:
        return "a";
    case foo::b:
        return "b";
    case foo::c:
        return "c";
    }
}
```

```cpp
const char* to_string(foo f)
{
    switch (f)
    {
        using enum foo;
    case a:
        return "a";
    case b:
        return "b";
    case c:
        return "c";
    }
}
```

struct vs. class

```
template <typename T>
struct foo
{};
```

struct vs. class

```
template <typename T>
struct foo
{};
```

```
template <class T>
struct foo
{};
```

struct vs. class

```
template <typename T>
struct foo
{};
```

```
template <class T>
struct foo
{};
```

What about `template <struct T>`?

struct vs. class

```
template <struct T>
struct foo {};
```

struct vs. class

```cpp
template <struct T>
struct foo {};
```

```cpp
struct T { int i; }

foo<T{0}> f;
```

think-cell

dynamic_cast<void*>

Checked downcast.

```
struct base { virtual ~base() = 0; };

struct derived : base {};
```

```
if (auto derived_ptr = dynamic_cast<derived*>(base_ptr))
{
    …
}
```

dynamic_cast<void*>

Checked sidecast.

```
struct base1 { virtual ~base1() = 0; };

struct base2 { virtual ~base2() = 0; };

struct derived : base1, base2 {};
```

```
if (auto base2_ptr = dynamic_cast<base2*>(base1_ptr))
{
    …
}
```

dynamic_cast<void*>

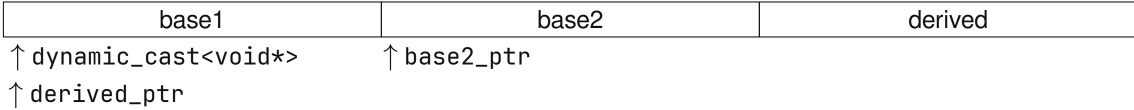
Cast to most-derived type.

```
struct base1 { virtual ~base1() = 0; };

struct base2 { virtual ~base2() = 0; };

struct derived : base1, base2 {};
```

```
auto address_of_derived = dynamic_cast<void*>(base2_ptr);
```



base1	base2	derived

↑ dynamic_cast<void*>

↑ derived_ptr

↑ base2_ptr

dynamic_cast<void*>

```cpp
class any_ref
{
    void* _ptr;
    std::type_info _type;

public:
    template <typename T>
    any_ref(T& obj)
    : _ptr(&obj), _type(typeid(obj))
    {}
    template <typename Base>
    static any_ref from_base(Base& base)
    {
        return {dynamic_cast<void*>(&base), typeid(base)};
    }
};
```

union

```
struct event
{
    event_kind kind; // uint8_t
    union {
        struct keyboard_event {
            bool shift : 1, ctrl : 1, alt : 1, system : 1;
            std::uint32_t keycode;
        } keyboard;
        struct mouse_click_event {
            button_kind button;
            std::uint16_t x, y;
        } mouse_click;
        …
    };
}; // sizeof(event) == 3 * sizeof(std::uint32_t)`
```

`union`

[class.mem.general]/26

> *In a standard-layout union with an active member of struct type `T1`, it is permitted to **read a non-static data member `m` of another union member** of struct type `T2` **provided m is part of the common initial sequence** of `T1` and `T2`; the behavior is as if the corresponding member of `T1` were nominated.*

union

[class.mem.general]/26

*In a standard-layout union with an active member of struct type T1, it is permitted to **read a non-static data member `m` of another union member** of struct type T2 **provided m is part of the common initial sequence** of T1 and T2; the behavior is as if the corresponding member of T1 were nominated.*

[class.mem.general]/25

*The common initial sequence of two standard-layout struct types is the **longest sequence of** non-static data **members** and bit-fields in **declaration order**, starting with the first such entity in each of the structs, such that*

- *corresponding entities have **layout-compatible types**,*
- *corresponding entities have the same alignment requirements,*
- *either both entities are declared with the no_unique_address attribute or neither is, and*
- *either both entities are bit-fields with the same width or neither is a bit-field.*

union

```
union event
{
    event_kind kind; // uint8_t
    struct keyboard_event {
        event_kind kind; // uint8_t
        bool shift : 1, ctrl : 1, alt : 1, system : 1;
        std::uint32_t keycode;
    } keyboard;
    struct mouse_click_event {
        event_kind kind; // uint8_t
        button_kind button;
        std::uint16_t x, y;
    } mouse_click;
    …
}; // sizeof(event) == 2 * sizeof(std::uint32_t)
```

union

```cpp
struct no_default_ctor
{
    no_default_ctor() = delete;
};
static_assert(std::is_empty_v<no_default_ctor>);

no_default_ctor obj = legally_create_object<no_default_ctor>();
```

Louis Dionne - “The Object Upside Down” - C++Now 2018 Lightning Talk

union

```cpp
union event
{
private:
    event_kind kind;
    struct keyboard_event { … } keyboard;
    struct mouse_click_event { … } mouse_click;
    …

public:
    static event make_keyboard(…);
    static event make_mouse_click(…);

    std::uint32_t keycode() const { … }


    …
};
```

`std::valarray`

Dynamically sized sequence containers:

- `std::vector<T>`
- `std::deque<T>`
- `std::list<T>`
- `std::forward_list<T>`

std::valarray

Dynamically sized sequence containers:

- `std::vector<T>`
- `std::deque<T>`
- `std::list<T>`
- `std::forward_list<T>`
- `std::vector<bool>`

std::valarray

Dynamically sized sequence containers:

- `std::vector<T>`
- `std::deque<T>`
- `std::list<T>`
- `std::forward_list<T>`
- `std::vector<bool>`
- `std::valarray<T>`

think-cell

std::valarray

`std::valarray` is an actual `vector`:

```
std::valarray<float> pos(3), velocity(3);
…
pos += dt * velocity;
```

std::valarray

std::valarray is an actual vector:

```
std::valarray<float> pos(3), velocity(3);
…
pos += dt * velocity;
```

```
std::valarray<float> matrix(n * n);
…
auto trace = matrix[std::slice(0, n, n + 1)].sum();
```

std::valarray

- wide range of mathematical operations
- implicitly `restrict`
- use of expression templates for optimized computation

std::valarray

- wide range of mathematical operations
- implicitly `restrict`
- use of expression templates for optimized computation

But: nobody uses it?

<stdexcept> implementation details

```
namespace std {
    class runtime_error : public exception {
    public:
        explicit runtime_error(const string& what_arg);
        explicit runtime_error(const char* what_arg);
        runtime_error(const runtime_error& other) noexcept;
        runtime_error& operator=(const runtime_error& other) noexcept;

        const char* what() const noexcept override;
    };
}
```

```
void fail(const T& arg) {
    throw std::runtime_error(std::format("'{}' went wrong.", arg));
}
```

<stdexcept> implementation details

`std::runtime_error` is a ref-counted string!

<stdexcept> implementation details

```cpp
class refcounted_string {
    std::runtime_error _impl;
public:
    refcounted_string(const std::string& str) : _impl(str) {}

    const char* c_str() const { return _impl.what(); }
    std::size_t length() const { return std::strlen(c_str()); }

    char operator[](std::size_t idx) const { return c_str()[idx]; }

};
```

Integer overflow

Is there UB?

```
int f(int a, int b)
{
    return a + b;
}
```

Integer overflow

Is there UB?

```
int f(int a, int b)
{
    return a * b;
}
```

Integer overflow

Is there UB?

```
int f(int a, int b)
{
    return a * b;
}
```

Sean Parent: overflow on 99.9999993% of all possible inputs.

Integer overflow

Is there UB?

```
int f(int a, int b)
{
    return a / b;
}
```

Integer overflow

Is there UB?

```
int f(int a, int b)
{
    assert(b != 0);
    return a / b;
}
```

Aside: Two's complement

Positive values: `0b0'xxxxxxx`

Negative values: `0b1'xxxxxxx`

Aside: Two's complement

Positive values: `0b0'xxxxxxx`

Negative values: `0b1'xxxxxxx`

What about zero?

Aside: Two's complement

Positive values: `0b0'xxxxxxx`

Negative values: `0b1'xxxxxxx`

What about zero?

```
-128
-127
…
-1
0
1
…
126
127
```

```
0b1'0000000
0b1'0000001
…
0b1'1111111
0b0'0000000
0b0'0000001
…
0b0'1111110
0b0'1111111
```

Integer overflow

Is there UB?

```
int f(int a, int b)
{
    assert(b != 0);
    return a / b;
}
```

```
f(INT_MIN, -1) // integer overflow!
```

Integer overflow

Is there UB?

```
int f(int a, int b)
{
    assert(b != 0);
    return a % b;
}
```

Integer overflow

Is there UB?

```
int f(int a, int b)
{
    assert(b != 0);
    return a % b;
}
```

```
f(INT_MIN, -1) // integer overflow!?
```

Integer overflow

[expr.mul]/4

> *The binary / operator yields the quotient, and the binary % operator yields the remainder from the division of the first expression by the second. If the second operand of / or % is zero the behavior is undefined. For integral operands the / operator yields the algebraic quotient with any fractional part discarded; if the quotient `a/b` is representable in the type of the result, `(a/b)*b + a%b` is equal to `a`;* ***otherwise, the behavior of both a/b and a%b is undefined.***

Integer overflow

```cpp
int f(int a, int b)
{
    assert(b != 0);
    return a % b;
}
```

```asm
mov     eax, DWORD PTR [rbp-4]
cdq
idiv    DWORD PTR [rbp-8]
mov     eax, edx
```

`idiv` computes quotient in `eax` and remainder in `edx`.

Integer overflow

```
int f(int a, int b)
{
    assert(b != 0);
    return a % b;
}
```

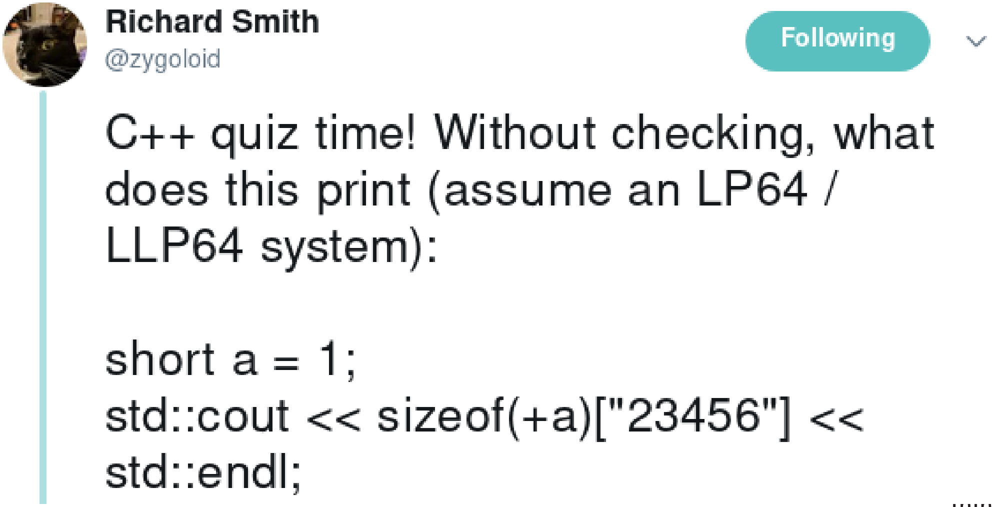
```
ldr     w8, [sp, #12]
ldr     w10, [sp, #8]
sdiv    w9, w8, w10
mul     w9, w9, w10
subs    w0, w8, w9
```

```
return a - (a / b) * b;
```

Integer overflow

```
$ lldb ./a.out
(lldb) target create "./a.out"
Current executable set to '/home/foonathan/Downloads/a.out' (x86_64).
(lldb) r
Process 645117 launched: '/home/foonathan/Downloads/a.out' (x86_64)
Process 645117 stopped
* thread #1, name = 'a.out',
    stop reason = signal SIGFPE: integer divide by zero
    frame #0: 0x0000555555555180 a.out`f(int, int) + 64
a.out`f:
->  0x555555555180 <+64>: idivl  -0x8(%rbp)
    0x555555555183 <+67>: movl   %edx, %eax
    0x555555555185 <+69>: addq   $0x10, %rsp
    0x555555555189 <+73>: popq   %rbp
```

My favorite C++ question



Richard Smith
@zygoloid

C++ quiz time! Without checking, what does this print (assume an LP64 / LLP64 system):

```
short a = 1;
std::cout << sizeof(+a)["23456"] << std::endl;
```

My favorite C++ question

```
   short a = 1;
//
   std::cout << sizeof(+a)["23456"] << std::endl;
//                         ^^^^^^^
```

- "23456" is a string literal

My favorite C++ question

```
   short a = 1;
//
   std::cout << sizeof(+a)["23456"] << std::endl;
//                         ^^^^^^^
```

- `"23456"` is a string literal
- string literals have type `const char[N]`

My favorite C++ question

```
   short a = 1;
//
   std::cout << sizeof(+a)["23456"] << std::endl;
//                     ^^
```

- a is a short

My favorite C++ question

```
   short a = 1;
//
   std::cout << sizeof(+a)["23456"] << std::endl;
//                     ^^
```

- a is a `short`
- unary plus does integer promotion

My favorite C++ question

```
   short a = 1;
//
   std::cout << sizeof(+a)["23456"] << std::endl;
//                     ^^
```

- a is a `short`
- unary plus does integer promotion
- the result is of type `int`

My favorite C++ question

```
   short a = 1;
//
   std::cout << sizeof(+a)["23456"] << std::endl;
//              ^^^^^^^^^^
```

- sizeof returns a std::size_t

My favorite C++ question

```
   short a = 1;
//
   std::cout << sizeof(+a)["23456"] << std::endl;
//              ^^^^^^^^^^
```

- `sizeof` returns a `std::size_t`
- `sizeof` of `char` is 1, `sizeof` otherwise implementation-defined

My favorite C++ question

```
   short a = 1;
//
   std::cout << sizeof(+a)["23456"] << std::endl;
//              ^^^^^^^^^^
```

- sizeof returns a std::size_t
- sizeof of char is 1, sizeof otherwise implementation-defined
- LP64/LLP64: sizeof(int) == 4

My favorite C++ question

```
   short a = 1;
//
   std::cout << sizeof(+a)["23456"] << std::endl;
//              ^^^^^^^^^^^^^^^^^^^
```

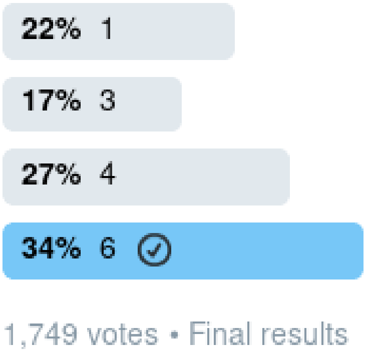
- builtin index operator is commutative

My favorite C++ question

```
   short a = 1;
//
   std::cout << sizeof(+a)["23456"] << std::endl;
//              ^^^^^^^^^^^^^^^^^^^
```

- builtin index operator is commutative
- `4["23456"] == "23456"[4] == 6`

My favorite C++ question



22% 1

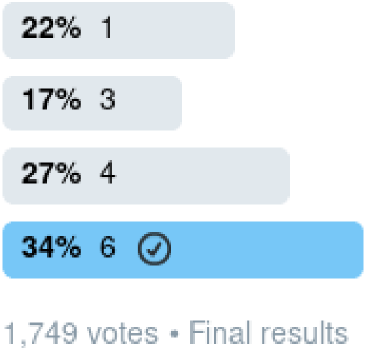
17% 3

27% 4

34% 6 ✓

1,749 votes • Final results

My favorite C++ question



22% 1

17% 3

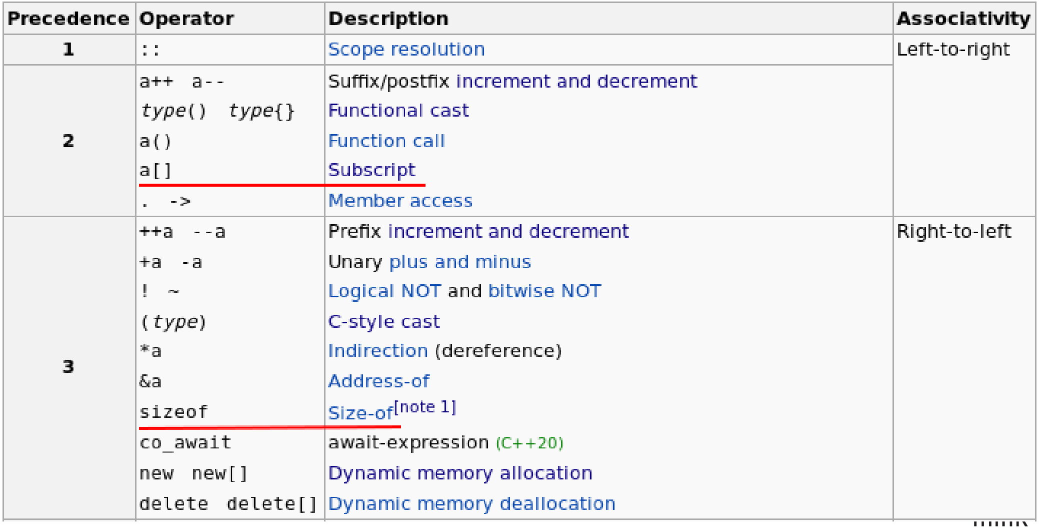
27% 4

34% 6 ✓

1,749 votes • Final results

1

My favorite C++ question



Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right
2	`a++ a--`	Suffix/postfix increment and decrement	
	type`()` *type*`{}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
	`. ->`	Member access	
3	`++a --a`	Prefix increment and decrement	Right-to-left
	`+a -a`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	`(`*type*`)`	C-style cast	
	`*a`	Indirection (dereference)	
	`&a`	Address-of	
	`sizeof`	Size-of[note 1]	
	`co_await`	await-expression (C++20)	
	`new new[]`	Dynamic memory allocation	
	`delete delete[]`	Dynamic memory deallocation	

My favorite C++ question

```
short a = 1;
std::cout << sizeof(+a)["23456"] << std::endl;
```

My favorite C++ question

```cpp
short a = 1;
std::cout << sizeof(+a)["23456"] << std::endl;
```

```cpp
short a = 1;
std::cout << sizeof (+a)["23456"] << std::endl;
```

My favorite C++ question

```cpp
short a = 1;
std::cout << sizeof(+a)["23456"] << std::endl;
```

```cpp
short a = 1;
std::cout << sizeof (+a)["23456"] << std::endl;
```

```cpp
short a = 1;
std::cout << sizeof( (+a)["23456"] ) << std::endl;
```

My favorite C++ question

```cpp
short a = 1;
std::cout << sizeof(+a)["23456"] << std::endl;
```

```cpp
short a = 1;
std::cout << sizeof (+a)["23456"] << std::endl;
```

```cpp
short a = 1;
std::cout << sizeof( (+a)["23456"] ) << std::endl;
```

[expr.sizeof]/1

> *[…] The **result of sizeof applied to any of the narrow character types is 1**. The result of sizeof applied to any other fundamental type ([basic.fundamental]) is implementation-defined.*

Modulo and negative numbers

`10 % 7?`

Modulo and negative numbers

10 % 7? 3

Modulo and negative numbers

`10 % 7?` 3

`10 % -7?`

Modulo and negative numbers

`10 % 7?` 3

`10 % -7?` 3

Modulo and negative numbers

`10 % 7?` 3

`10 % -7?` 3

`-10 % 7?` ???

Modulo and negative numbers

Division of a by b `a = (a/b) * b + (a%b)` and `abs(a%b) < b`.

Modulo and negative numbers

Division of a by b `a = (a/b) * b + (a%b)` and `abs(a%b) < b`.

Algorithm	Rounding of Quotient	Remainder Sign	Remainder Interval

Modulo and negative numbers

Division of a by b `a = (a/b) * b + (a%b)` and `abs(a%b) < b`.

Algorithm	Rounding of Quotient	Remainder Sign	Remainder Interval
Truncation	towards zero	`sgn(a)`	`[0, a)` or `(a, 0]`

Modulo and negative numbers

Division of a by b `a = (a/b) * b + (a%b)` and `abs(a%b) < b`.

Algorithm	Rounding of Quotient	Remainder Sign	Remainder Interval
Truncation	towards zero	`sgn(a)`	`[0, a)` or `(a, 0]`
Floored	towards `INT_MIN`	`sgn(b)`	`[0, a)` or `(-a, 0]`

Modulo and negative numbers

Division of a by b `a = (a/b) * b + (a%b)` and `abs(a%b) < b`.

Algorithm	Rounding of Quotient	Remainder Sign	Remainder Interval
Truncation	towards zero	`sgn(a)`	`[0, a)` or `(a, 0]`
Floored	towards `INT_MIN`	`sgn(b)`	`[0, a)` or `(-a, 0]`
Ceiling	towards `INT_MAX`	`-sgn(b)`	`[0, a)` or `(-a, 0]`

think-cell

Modulo and negative numbers

Division of a by b `a = (a/b) * b + (a%b)` and `abs(a%b) < b`.

Algorithm	Rounding of Quotient	Remainder Sign	Remainder Interval
Truncation	towards zero	`sgn(a)`	`[0, a)` or `(a, 0]`
Floored	towards `INT_MIN`	`sgn(b)`	`[0, a)` or `(-a, 0]`
Ceiling	towards `INT_MAX`	`-sgn(b)`	`[0, a)` or `(-a, 0]`
Rounded	to closer integer	+ or -	`[-b/2, b/2]`

Modulo and negative numbers

Division of a by b `a = (a/b) * b + (a%b)` and `abs(a%b) < b`.

Algorithm	Rounding of Quotient	Remainder Sign	Remainder Interval
Truncation	towards zero	`sgn(a)`	`[0, a)` or `(a, 0]`
Floored	towards `INT_MIN`	`sgn(b)`	`[0, a)` or `(-a, 0]`
Ceiling	towards `INT_MAX`	`-sgn(b)`	`[0, a)` or `(-a, 0]`
Rounded	to closer integer	+ or -	`[-b/2, b/2]`
Euclidean	depending on `sgn(b)`	+	`[0, abs(a))`

Modulo and negative numbers

Truncated (C++):

- `10 / 7 == 1`
- `10 % 7 == 3`

Floored (Lua):

- `10 / 7 == 1`
- `10 % 7 == 3`

Euclidean (Dart):

- `10 / 7 == 1`
- `10 % 7 == 3`

think-cell

Modulo and negative numbers

Truncated (C++):

- `10 / 7 == 1`
- `10 % 7 == 3`

- `10 / -7 == -1`
- `10 % -7 == 3`

Floored (Lua):

- `10 / 7 == 1`
- `10 % 7 == 3`

- `10 / -7 == -2`
- `10 % -7 == -4`

Euclidean (Dart):

- `10 / 7 == 1`
- `10 % 7 == 3`

- `10 / -7 == -1`
- `10 % -7 == 3`

Modulo and negative numbers

Truncated (C++):

- `10 / 7 == 1`
- `10 % 7 == 3`

- `10 / -7 == -1`
- `10 % -7 == 3`

- `-10 / 7 == -1`
- `-10 % 7 == -3`

Floored (Lua):

- `10 / 7 == 1`
- `10 % 7 == 3`

- `10 / -7 == -2`
- `10 % -7 == -4`

- `-10 / 7 == -2`
- `-10 % 7 == 4`

Euclidean (Dart):

- `10 / 7 == 1`
- `10 % 7 == 3`

- `10 / -7 == -1`
- `10 % -7 == 3`

- `-10 / 7 == -2`
- `-10 % 7 == 4`

Modulo and negative numbers

Truncated (C++):

- `10 / 7 == 1`
- `10 % 7 == 3`

- `10 / -7 == -1`
- `10 % -7 == 3`

- `-10 / 7 == -1`
- `-10 % 7 == -3`

- `-10 / -7 == 1`
- `-10 % -7 == -3`

Floored (Lua):

- `10 / 7 == 1`
- `10 % 7 == 3`

- `10 / -7 == -2`
- `10 % -7 == -4`

- `-10 / 7 == -2`
- `-10 % 7 == 4`

- `-10 / -7 == 1`
- `-10 % -7 == -3`

Euclidean (Dart):

- `10 / 7 == 1`
- `10 % 7 == 3`

- `10 / -7 == -1`
- `10 % -7 == 3`

- `-10 / 7 == -2`
- `-10 % 7 == 4`

- `-10 / -7 == 2`
- `-10 % -7 == 4`

Conclusion

We're hiring: think-cell.com/italiancpp

jonathanmueller.dev/talk/cpp-features

@foonathan@fosstodon.org
youtube.com/@foonathan

think-cell