C++20 in Practice

Nicolai M. Josuttis
josuttis.com
@NicoJosuttis

07/24

Table of Contents

Nicolai M. Josuttis

- **Independent consultant**
  - Continuously learning since 1962
- **C++:**
  - since 1990
  - ISO Standard Committee since 1997
- **Other Topics:**
  - Systems Architect
  - Technical Manager
  - SOA
  - X and OSF/Motif

General Disclaimer

- **C++ covers a big variety of platforms**
  - Some aspects are common but not guaranteed
    - e.g. a character has 8 bits
  - Conventions vs. formal rules
  - Implementations may vary
  - Compilers might have special behavior
- **C++ evolves**
  - Rules change
  - Problem of backward compatibility
- **C++ is complex**
  - We skip a lot of details
  - Keep it simple (maintainable)

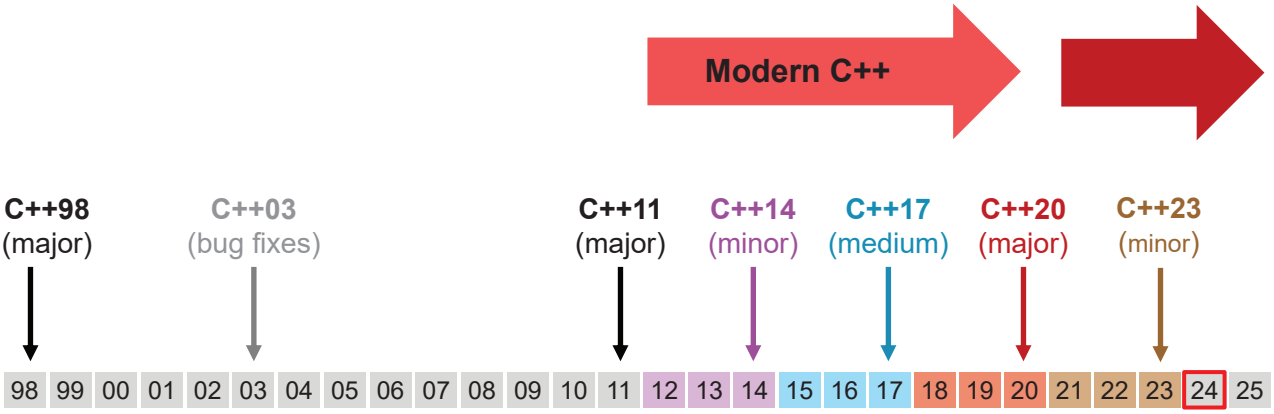
Hints and patterns for programming in practice

- **Sometimes I teach a feature the way you shouldn't use it**

C++ Timeline

C++ version(s) of a slide

http://isocpp.org/std/status:

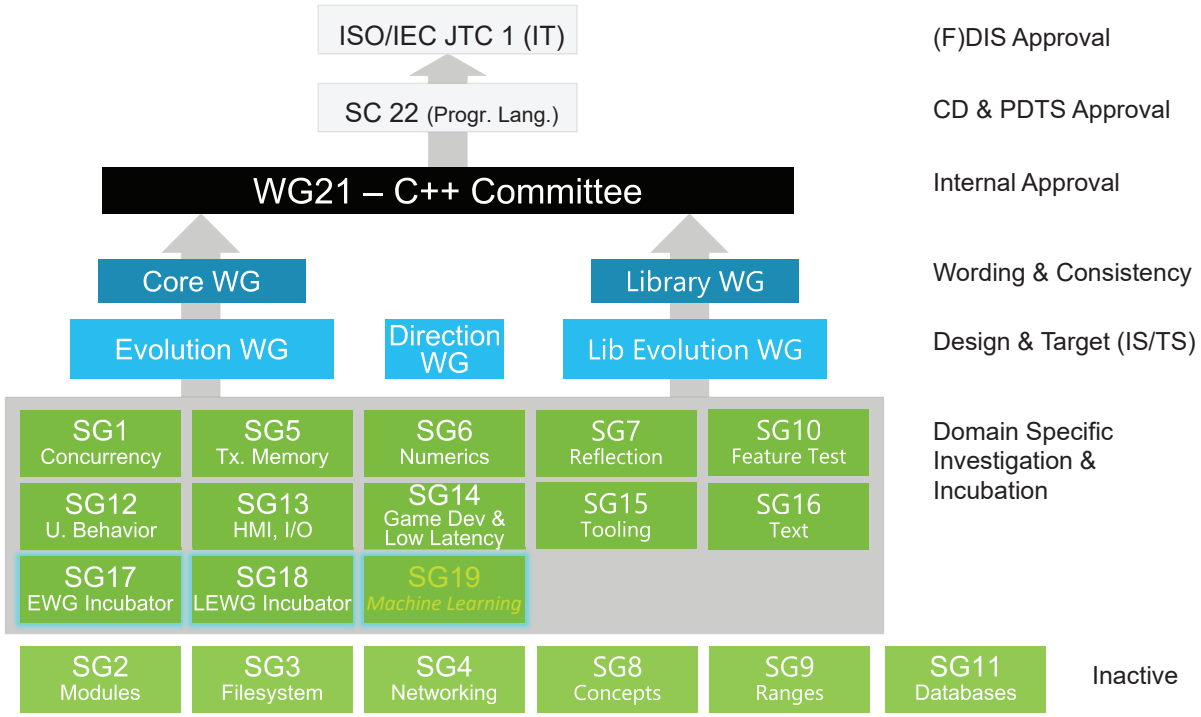


C++ Standardization

- **by ISO**
  - formal votes by national bodies, e.g.:
    - ANSI for USA
    - DIN for Germany
- **Email reflectors for different working and study groups**
  - core, library, concurrency, ...
- **2 or 3 joined meetings of ANSI and ISO per year**
  - open to the public
- **Everybody is welcome to**
  - join meetings
  - propose new features
  - discuss

- **Informal web site: http://isocpp.org/std**
- **Formal web site: http://www.open-std.org/jtc1/sc22/wg21/**

C++ Working and Study Groups



Availability of the C++ Standards

- **Document numbers:**
  - **C++11**: ISO/IEC **14882:2011** or **14882-2012**
  - **C++14**: ISO/IEC **14882:2014**
  - **C++17**: ISO/IEC **14882:2017**
  - **C++20**: ISO/IEC **14882:2020**
- **Available at national standard bodies**
  - ANSI, DIN, BSI, and INCITS
  - for different prices (always same contents)
- **Latest draft for free here:**
  - **C++11**: http://wg21.link/**n3242**
  - **C++14**: http://wg21.link/**n4296**
  - **C++17**: http://wg21.link/**n4659**
  - **C++20**: http://wg21.link/**n4861**
  - **C++23**: http://wg21.link/**n4944**

Step-by-Step Compiler Support

C++

- http://en.cppreference.com/w/cpp/compiler_support

C++11 feature	Paper(s)	GCC	Clang	MSVC	Apple Clang	EDG eccp	Intel C++	IBM XLC++	Sun/Oracle C++	Embarcadero C++ Builder	Nvidia HPC C++ (ex Portland Group/PGI) Cray	Nvidia nvcc	HP aCC	Digital Mars C++
C99 preprocessor	N1653	4.3	Yes	19.0 (2015)* (partial)* 19.26*	Yes	4.1	11.1	10.1	5.9	Yes	8.4 2015	7.0	A.06.25	Yes
`static_assert`	N1720	4.3	2.9	16.0*	Yes	4.1	11.0	11.1	5.13	Yes	8.4 2015	7.0	A.06.25	8.52
Right angle brackets	N1757	4.3	Yes	14.0*	Yes	4.1	11.0	12.1	5.13	Yes	8.4 2015	7.0		
Extended `friend` declarations	N1791	4.7	2.9	16.0* (partial) 18.0*	Yes	4.1	11.1* 12.0	11.1	5.13	Yes	8.4 2015	7.0	A.06.25	
`long long`	N1811	Yes	Yes	14.0*	Yes	Yes	Yes	Yes	Yes	Yes	8.4 2015	7.0	Yes	Yes
Compiler support for type traits	N1836 N2518* N2984 N3142	4.3* 4.8* 5	3.0	14.0* (partial)* 19.0 (2015)*	Yes	4.0	10.0	13.1.3	5.13	Yes	8.4 2015		6.16	
`auto`	N1984	4.4	Yes	16.0*	Yes	3.9	11.0 (v0.9) 12.0	11.1	5.13	Yes	8.4 2015	7.0	A.06.25	
Delegating constructors	N1986	4.7	3.0	18.0*	Yes	4.7	14.0	11.1	5.13	Yes	8.4 2015	7.0	A.06.28	

Backward Compatibility of Modern C++

C++

- **Backward compatibility is a major goal**
  - Code should still compile
  - If code still compiles, it must not change its behavior
  - Compiler vendors also force binary compatibility
    - Enables to link together code of different C++ version
- **Backward compatibility might be broken by:**
  - New keywords:

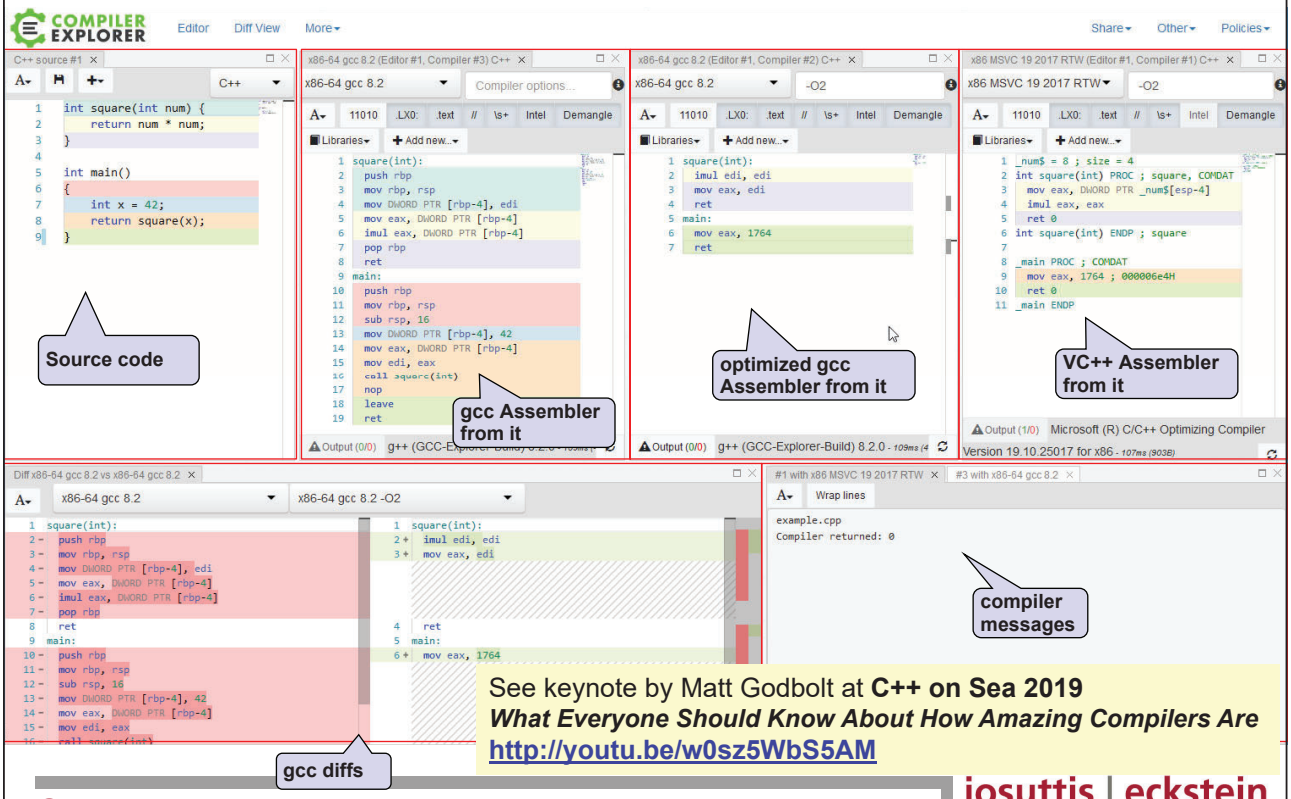
    C++11:
    `alignas`, `alignof`, `char16_t`, `char32_t`, `constexpr`, `decltype`, `noexcept`, `nullptr`, `static_assert`, `thread_local`

    C++20:
    `char8_t`, `co_await`, `co_return`, `co_yield`, `concept`, `requires`, `consteval`, `constinit`

    Context-sensitive keywords: `final`, `override` and `import`, `module`
  - Deprecated and removed features
    - C++11 deprecates e.g. `throw` specifications, keyword `register`
    - C++14 removes e.g. C function `gets()`

Online Compiler Explorer godbolt.org

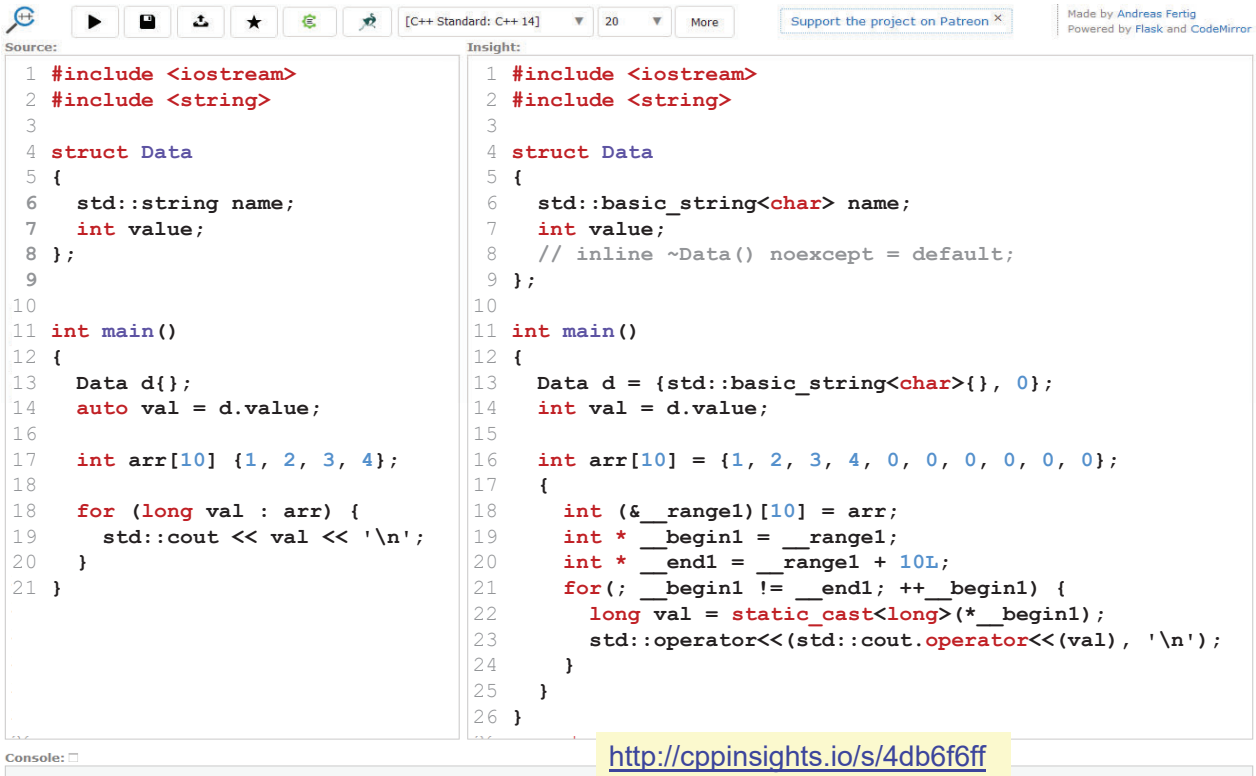
C++



See keynote by Matt Godbolt at **C++ on Sea 2019**
What Everyone Should Know About How Amazing Compilers Are
http://youtu.be/w0sz5WbS5AM

cppinsights.io

C++




```
#include <iostream>
#include <string>

struct Data
{
  std::string name;
  int value;
};


int main()
{
  Data d{};
  auto val = d.value;

  int arr[10] {1, 2, 3, 4};

  for (long val : arr) {
    std::cout << val << '\n';
  }
}
```

```
#include <iostream>
#include <string>

struct Data
{
  std::basic_string<char> name;
  int value;
  // inline ~Data() noexcept = default;
};

int main()
{
  Data d = {std::basic_string<char>{}, 0};
  int val = d.value;

  int arr[10] = {1, 2, 3, 4, 0, 0, 0, 0, 0, 0};
  {
    int (&__range1)[10] = arr;
    int * __begin1 = __range1;
    int * __end1 = __range1 + 10L;
    for(; __begin1 != __end1; ++__begin1) {
      long val = static_cast<long>(*__begin1);
      std::operator<<(std::cout.operator<<(val), '\n');
    }
  }
}
```

http://cppinsights.io/s/4db6f6ff

Initialization

C++11

- **Uniform initialization** (since C++11): **{...} everywhere**

```
int i0;                                  // undefined value
int i1 = 42;                             // initializes with 42
int i2(42);                              // initializes with 42
int i3 = int();                          // initializes with 0
int i4{42};                              // initializes with 42
int i5{};                                // initializes with 0
int i6 = {42};                           // initializes with 42
int i7 = {};                             // initializes with 0

int vals1[] = { 1, 2, 3 };               // initialization of struct/array
int vals2[] { 1, 2, 3 };                 // initialization of struct/array

std::complex<double> c1(4.0, 3.0);       // initialization of classes
std::complex<double> c2{4.0, 3.0};       // initialization of classes
std::complex<double> c3 = {4.0, 3.0};    // initialization of classes

std::vector<std::string> cities{"Berlin", "Rome"};
std::vector<int> vals3{0, 8, 15, i1 + i2};
std::vector<int> vals4 = {i1, i2, 42};
```


Ways of Initialization

C++98/C++11

```
int val = 42;

int arr[10] = {0, val, val+10};

for (int i = 0; i < 32; ++i) {
  std::cout << i << '\n';
}

std::string s1 = "hello";
std::string s2("hello");
std::string s3(64, ' ');   // 64 spaces

class MyType {
 private:
  std::string name;
  int id;
 public:
  MyType(std::string n)
   : name(n), id(nextId()) {
  }
};
```

```
int val{42};

int arr[10] {0, val, val+10};

for (int i{0}; i < 32; ++i) {
  std::cout << i << '\n';
}

std::string s1{"hello"};

std::string s4{64, ' '};   // '@' and space

class MyType {
 private:
  std::string name;
  int id{nextId()};   // no (...) allowed here
 public:
  MyType(std::string n)
   : name{n} {
  }
};
```

Prefer uniform direct initialization
- with {...}, but without =

and = for trivial cases

C++20

Comparisons and Operator <=>

Implicit Type Conversions and Operators (before C++20)

C++98/C++11

```
class Person
{
 private:
  std::string name;
  int value;
 public:
  // constructor:
  Person (const std::string& n, int v = 0)
   : name{n}, value{v} {
  }
  std::string getName() const {
    return name;
  }
  ...
  bool operator== (const Person& rhs) const {
    return name == rhs.name &&
           value == rhs.value;
  }
};
```

Each **constructor** defines an **implicit type conversion**

- Implicit conversion possible only for **parameters** (second operand)
- Usual fix: declare free-standing or as hidden friend

```
void print(const Person& p)
{
  std::cout << p.getName() << ": "
            << p.getValue() << '\n';
}

Person p1{"Ann", 42};
Person p2{"Jim"};
print(p1);
if (p1 == p2) {       // OK: p1.operator==(p2)
  ...
}
if (p1 != p2) {       // ERROR: != not defined
  ...
}

std::string str = "Tim";
print(str);           // OK: implicit conv.
print({"Kim", 8});    // OK: implicit conv.

if (p1 == str) {      // OK: p1.operator==(str)
  ...
}
if (str == p1) {      // ERROR: str no parameter
  ...
}
```

Dealing with User-Defined Operators

C++98

Expression:	Compilers tries:
obj1 + *obj2*	*obj1* . operator+ (*obj2*)
	and
	operator+ (*obj1* , *obj2*)
! *obj*	*obj* . operator! ( )
	and
	operator! (*obj*)

Hidden Friends

C++11

```
class Person
{
 private:
  std::string name;
  int value;
 public:
  // constructor:
  Person (const std::string& n, int v = 0)
   : name{n}, value{v} {
  }
  std::string getName() const {
    return name;
  }
  ...
  friend bool operator== (const Person& lhs,
                          const Person& rhs) {
    return lhs.name == rhs.name &&
           lhs.value == rhs.value;
  }
  friend bool operator!= (const Person& lhs,
                          const Person& rhs) {
    return !(lhs == rhs); // or: !operator==(lhs,rhs)
  }
};
```

"hidden friend" `friend` definition inside the class

- **Use hidden friends for operations that clearly belong to a class**
  - Access to private members
  - Not callable without Person involved

(since C++20, not necessary for comparisons)

```
void print(const Person& p)
{
  std::cout << p.getName() << ": "
            << p.getValue() << '\n';
}

Person p1{"Ann", 42};
Person p2{"Jim"};
print(p1);
if (p1 == p2) {       // OK: operator==(p1,p2)
  ...
}
if (p1 != p2) {       // OK: operator!=(p1,p2)
  ...
}

std::string str = "Tim";
print(str);           // OK: implicit conversion
if (p1 == str) {      // OK: operator==(p1,str)
  ...
}
if (str == p1) {      // OK: operator==(str,p1)
  ...
}
if (str != p1) {      // OK: operator!=(str,p1)
  ...
}
```

Implicit Type Conversions and Operators (since C++20)

C++20

```
class Person
{
 private:
  std::string name;
  int value;
 public:
  // constructor:
  Person (const std::string& n, int v = 0)
   : name{n}, value{v} {
  }
  std::string getName() const {
    return name;
  }
  …
  bool operator== (const Person& rhs) const {
    return name == rhs.name &&
           value == rhs.value;
  }
};
```

Expressions *rewritten* to enable compilation

```
void print(const Person& p)
{
  std::cout << p.getName() << ": "
            << p.getValue() << '\n';
}

Person p1{"Ann", 42};
Person p2{"Jim"};
print(p1);
if (p1 == p2) {       // OK: p1.operator==(p2)
  …
}
if (p1 != p2) {       // OK: as  ! (p1 == p2)
  …
}

std::string str = "Tim";
print(str);           // OK: implicit conversion
if (p1 == str) {      // OK: p1.operator==(str)
  …
}
if (str == p1) {      // OK: as  p1 == str
  …
}
if (str != p1) {      // OK: as  ! (p1 == str)
  …
}
```


Implicit Type Conversions and Operators (since C++20)

C++20

```
class Person
{
 private:
  std::string name;
  int value;
 public:
  // constructor:
  Person (const std::string& n, int v = 0)
   : name{n}, value{v} {
  }
  std::string getName() const {
    return name;
  }
  …
  bool operator== (const Person&) const = default;
  auto operator<=> (const Person&) const = default;
};
```

- `operator==` used for rewritten calls of `==, !=`
- `operator<=>` used for rewritten calls of `<, <=, >, >=`

```
void print(const Person& p)
{
  std::cout << p.getName() << ": "
            << p.getValue() << '\n';
}

Person p1{"Ann", 42};
Person p2{"Jim"};
print(p1);
if (p1 == p2) {       // OK: p1.operator==(p2)
  …
}
if (p1 != p2) {       // OK: as  ! (p1 == p2)
  …
}

std::string str = "Tim";
print(str);           // OK: implicit conversion
if (p1 == str) {      // OK: p1.operator==(str)
  …
}
if (str == p1) {      // OK: as  p1 == str
  …
}
if (str != p1) {      // OK: as  ! (p1 == str)
  …
}
if (str < p1) {       // OK: as  (p1 <=> str) > 0
  …
}
```

Implicit Type Conversions and Operators (since C++20)

C++20

```
class Person
{
 private:
  std::string name;
  int value;
 public:
  // constructor:
  Person (const std::string& n, int v = 0)
   : name{n}, value{v} {
  }
  std::string getName() const {
    return name;
  }
  …
  bool operator== (const Person&) const = default;
  auto operator<=> (const Person&) const = default;
};
```

Implicitly also declares:
`bool operator== (const Person&) const = default;`

- `operator==` used for rewritten calls of `==, !=`
- `operator<=>` used for rewritten calls of `<, <=, >, >=`

```
void print(const Person& p)
{
  std::cout << p.getName() << ": "
            << p.getValue() << '\n';
}

Person p1{"Ann", 42};
Person p2{"Jim"};
print(p1);
if (p1 == p2) {        // OK: p1.operator==(p2)
  …
}
if (p1 != p2) {        // OK: as ! (p1 == p2)
  …
}

std::string str = "Tim";
print(str);            // OK: implicit conversion
if (p1 == str) {       // OK: p1.operator==(str)
  …
}
if (str == p1) {       // OK: as p1 == str
  …
}
if (str != p1) {       // OK: as ! (p1 == str)
  …
}
if (str < p1) {        // OK: as (p1 <=> str) > 0
  …
}
```


C++20: Automatic Generation of Comparison Operators

C++20

```
class MyType {
 private:
  MyType val;
 public:
  …
  bool operator==(const MyType& rhs) const { return val == rhs.val; }
  bool operator!=(const MyType& rhs) const { return !(*this == rhs); }
  bool operator<(const MyType& rhs)  const { return val < rhs.val; }
  bool operator<=(const MyType& rhs) const { return !(rhs < *this); }
  bool operator>(const MyType& rhs)  const { return rhs < *this; }
  bool operator>=(const MyType& rhs) const { return !(*this < rhs); }
};
```

Should ideally be:
- Hidden friend
- `noexcept`
- `constexpr`
- `[[nodiscard]]`

C++20

```
#include <compare>
class MyType {
 private:
  MyType val;
 public:
  …
  bool operator==(const MyType&) const = default;  // implicitly also declared
  auto operator<=>(const MyType&) const = default;
};
```

Effect:
- like hidden friend
- with `noexcept`
- with `constexpr`

- Delegates comparisons to members
- Order of member matters for `<, <=, >, >=`

Using Default Generated Comparisons

C++20

```
#include <compare>    // necessary to define operator <=>

struct Coord {
  double x, y, z;

  // enable all comparison operators (comparing according to x, then y, then z):
  auto operator<=> (const Coord&) const = default;

  friend std::ostream& operator<< (std::ostream& strm, const Coord& c) {
    return strm << '[' << c.x << ',' << c.y << ',' << c.z << ']';
  }
};
```

Generated operators are
- constexpr
- noexcept

Due to rewriting we also get
- conversions for both operands

```
std::vector<Coord> coll{{1,2,3}, {3,2,1}, {1,-1,1}, {0,8,15}, {9,7,7}};
std::sort(coll.begin(), coll.end());    // OK: uses rewriting of < to <=>
print(coll);

constexpr Coord c{0, 0, 0};
if constexpr (c < Coord{0.5, -0.5, -0.5}) ...  // OK
```

Output:

```
[0,8,15]
[1,-1,1]
[1,2,3]
[3,2,1]
[9,7,7]
```


Using Defaulted Operator <=>

C++20

```
class Person {
  std::string first;  // firstname
  std::string last;   // lastname
 public:
  Person(std::string f, std::string l)
   : first{std::move(f)}, last{std::move(l)} {
  }
  Person(const char* l)
   : first{}, last{l} {
  }
  ...
  auto operator<=> (const Person&) const = default;

  friend std::ostream& operator<< (std::ostream& os, const Person& p);
};
```

Order of members matters

Enables
==, !=, <, <=, >, >=
Comparison order:
first
last

```
std::vector<Person> coll{{"Aretha", "Franklin"}, {"Muddy", "Waters"},
                         {"Bob", "Dylan"}, {"Ray", "Charles"}};
std::sort(coll.begin(), coll.end());    // OK
print(coll);

Person p{"Tina", "Turner"};
if (p <= coll[0]) ...       // OK
if (p == coll[0]) ...       // OK
```

Output:

```
- elem: [Aretha Franklin]
- elem: [Bob Dylan]
- elem: [Muddy Waters]
- elem: [Ray Charles]
```

Implementing Operator <=>

C++20

```
class Person {
  std::string first;  // firstname
  std::string last;   // lastname
 public:
  Person(std::string f, std::string l)
   : first{std::move(f)}, last{std::move(l)} {
  }
  Person(const char* l)
   : first{}, last{l} {
  }
  ...
  auto operator<=> (const Person& p2) const {
    return last <=> p2.last;  // ordering uses lastname only
  }
};
```

Used by <, <=, >, >= (operator== *not* defined)

```
std::vector<Person> coll{{"Aretha", "Franklin"}, {"Muddy", "Waters"},
                         {"Bob", "Dylan"}, {"Ray", "Charles"}};
std::sort(coll.begin(), coll.end());    // OK
print(coll);

Person p{"Tina", "Turner"};
if (p <= coll[0]) ...       // OK
if (p == coll[0]) ...       // ERROR
```

== and != *not* callable

Output:

```
- elem: [Ray Charles]
- elem: [Bob Dylan]
- elem: [Aretha Franklin]
- elem: [Muddy Waters]
```


Implementing Operators == and <=>

C++20

```
class Person {
  std::string first;  // firstname
  std::string last;   // lastname
 public:
  Person(std::string f, std::string l)
   : first{std::move(f)}, last{std::move(l)} {
  }
  ...
  bool operator== (const Person& p2) const {
    return last == p2.last;   // equality uses lastname only
  }
  auto operator<=> (const Person& p2) const {
    return last <=> p2.last;  // ordering uses lastname only
  }
};
```

Used by ==, !=

Used by <, <=, >, >=

```
std::vector<Person> coll{{"Aretha", "Franklin"}, {"Muddy", "Waters"},
                         {"Bob", "Dylan"}, {"Ray", "Charles"}};
std::sort(coll.begin(), coll.end());    // OK
print(coll);

Person p{"Tina", "Turner"};
if (p <= coll[0]) ...       // OK
if (p == coll[0]) ...       // OK
```

Output:

```
- elem: [Ray Charles]
- elem: [Bob Dylan]
- elem: [Aretha Franklin]
- elem: [Muddy Waters]
```

Ordering Types

C++

- **Strong Ordering**
  - **Total order**: Each value with a different value is less or greater
    - Examples: integral types, strings, ...
  - Comparison results: less, equal, greater
- **Weak Ordering**
  - Equivalent values might not be equal
    - Example: Case-insensitive strings:
      - `"HowGee"` and `"howgee"` are equivalent but not equal
  - Comparison results: less, equivalent, greater
- **Partial Ordering**
  - There are values you can't compare (each comparison is false):
    - Example: Floating-point types:
      - 1.0 < NaN and 1.0 == NaN and 1.0 > NaN are false, so 1.0 <= NaN ≠ !(NaN < 1.0)
  - Comparison results: less, equivalent, greater, unordered

User-Defined Comparison Operators (since C++20)

C++20

```
class Cust {
  std::string first;                  // firstname
  std::string last;                   // lastname
  int         val;                    // some value
  std::vector<std::string> data;      // some data
 public:
  …
  bool operator== (const Cust& c) const noexcept {
    return first == c.first && last == c.last && val == c.val;
  }

  auto operator<=> (const Cust& c) const noexcept {
    auto cmp = last <=> c.last;       // 1st ordering criterion: lastname
    if (cmp != 0) return cmp;         //  if not equal, return result of comparison
    cmp = first <=> c.first;          // 2nd ordering criterion: firstname
    if (cmp != 0) return cmp;         //  if not equal, return result of comparison
    return val <=> c.val;             // 3rd ordering criterion: value
  }
  …
};
```

```
Cust c1{…};
Cust c2{…};

c1 < c2
// is rewritten as:
c1 <=> c2 < 0
```

Return type of <=> is ordering type:
- `std::strong_ordering` here

Comparisons with 0 are supported by all ordering types:
- < 0 : less
- > 0 : greater
- == 0 : neither less nor greater

Specific ordering types support enums like
`std::strong_ordering::less`

User-Defined Comparison Operators (since C++20)

C++20

```
class Cust {
  std::string first;                  // firstname
  std::string last;                   // lastname
  double      val;                    // some value
  std::vector<std::string> data;      // some data
 public:
  …
  bool operator== (const Cust& c) const noexcept {
    return first == c.first && last == c.last && val == c.val;
  }

  auto operator<=> (const Cust& c) const noexcept {
    auto cmp = last <=> c.last;       // 1st ordering criterion: lastname
    if (cmp != 0) return cmp;         //  if not equal, return result of comparison
    cmp = first <=> c.first;          // 2nd ordering criterion: firstname
    if (cmp != 0) return cmp;         //  if not equal, return result of comparison
    return val <=> c.val;             // 3rd ordering criterion: value
  }
  …
};
```

```
Cust c1{…};
Cust c2{…};

c1 < c2
// is rewritten as:
c1 <=> c2 < 0
```

ERROR: Can't deduce return type
- `std::strong_ordering` for <=> on strings
- `std::partial_ordering` for <=> on `double`

User-Defined Comparison Operators (since C++20)

C++20

```
class Cust {
  std::string first;                  // firstname
  std::string last;                   // lastname
  double      val;                    // some value
  std::vector<std::string> data;      // some data
 public:
  …
  bool operator== (const Cust& c) const noexcept {
    return first == c.first && last == c.last && val == c.val;
  }

  std::partial_ordering operator<=> (const Cust& c) const noexcept {
    auto cmp = last <=> c.last;       // 1st ordering criterion: lastname
    if (cmp != 0) return cmp;         //  if not equal, return result of comparison
    cmp = first <=> c.first;          // 2nd ordering criterion: firstname
    if (cmp != 0) return cmp;         //  if not equal, return result of comparison
    return val <=> c.val;             // 3rd ordering criterion: value
  }
  …
};
```

```
Cust c1{…};
Cust c2{…};

c1 < c2
// is rewritten as:
c1 <=> c2 < 0
```

OK:
- `std::strong_ordering` converts to `std::partial_ordering`

User-Defined Comparison Operators (since C++20)

C++20

```
class Cust {
  std::string first;                  // firstname
  std::string last;                   // lastname
  double      val;                    // some value
  std::vector<std::string> data;      // some data
 public:
  ...
  bool operator== (const Cust& c) const noexcept {
    return first == c.first && last == c.last && val == c.val;
  }

  std::strong_ordering operator<=> (const Cust& c) const noexcept {
    auto cmp = last <=> c.last;       // 1st ordering criterion: lastname
    if (cmp != 0) return cmp;         //  if not equal, return result of comparison
    cmp = first <=> c.first;          // 2nd ordering criterion: firstname
    if (cmp != 0) return cmp;         //  if not equal, return result of comparison
    return val <=> c.val;             // 3rd ordering criterion: value
  }
  ...
};
```

```
Cust c1{...};
Cust c2{...};

c1 < c2
// is rewritten as:
c1 <=> c2 < 0
```

ERROR:
- `std::partial_ordering` does *not* convert to `std::strong_ordering`

User-Defined Comparison Operators (since C++20)

C++20

```
class Cust {
  std::string first;                  // firstname
  std::string last;                   // lastname
  double      val;                    // some value
  std::vector<std::string> data;      // some data
 public:
  ...
  bool operator== (const Cust& c) const noexcept {
    return first == c.first && last == c.last && val == c.val;
  }

  std::strong_ordering operator<=> (const Cust& c) const noexcept {
    auto cmp = last <=> c.last;       // 1st ordering criterion: lastname
    if (cmp != 0) return cmp;         //  if not equal, return result of comparison
    cmp = first <=> c.first;          // 2nd ordering criterion: firstname
    if (cmp != 0) return cmp;         //  if not equal, return result of comparison
    auto v = val <=> c.val;           // 3rd ordering criterion: value
    if (v == std::partial_ordering::less) return std::strong_ordering::less;
    if (v == std::partial_ordering::equivalent) return std::strong_ordering::equal;
    if (v == std::partial_ordering::greater) return std::strong_ordering::greater;
    ...  // handle std::partial_ordering::unordered
  }
  ...
};
```

```
Cust c1{...};
Cust c2{...};

c1 < c2
// is rewritten as:
c1 <=> c2 < 0
```

map `std::partial_ordering` to `std::strong_ordering`

User-Defined Comparison Operators (since C++20)

C++20

```
class Cust {
  std::string first;                  // firstname
  std::string last;                   // lastname
  double      val;                    // some value
  std::vector<std::string> data;      // some data
 public:
  …
  bool operator== (const Cust& c) const noexcept {
    return first == c.first && last == c.last && val == c.val;
  }

  std::strong_ordering operator<=> (const Cust& c) const noexcept {
    auto cmp = last <=> c.last;       // 1st ordering criterion: lastname
    if (cmp != 0) return cmp;         //  if not equal, return result of comparison
    cmp = first <=> c.first;          // 2nd ordering criterion: firstname
    if (cmp != 0) return cmp;         //  if not equal, return result of comparison
    // 3rd ordering criterion: value
    // - use helper to map partial_ordering to strong_ordering:
    return std::strong_order(val, c.val);          // for types with <=>
    // - or if only < and == provided for the type:
    return std::compare_strong_order_fallback(val, c.val);   // works with < and ==
  }
  …
};
```

```
Cust c1{…};
Cust c2{…};

c1 < c2
// is rewritten as:
c1 <=> c2 < 0
```

honors rules for `double`/`float` (acc. to ISO/IEC/IEEE 60559)

User-Defined Comparison Operators (since C++20)

C++20

```
class Cust {
  std::string first;                  // firstname
  std::string last;                   // lastname
  double      val;                    // some value
  std::vector<std::string> data;      // some data
 public:
  …
  bool operator== (const Cust& c) const noexcept {
    return operator<=>(c) == 0;       // map == to <=>
  }

  std::strong_ordering operator<=> (const Cust& c) const noexcept {
    auto cmp = last <=> c.last;       // 1st ordering criterion: lastname
    if (cmp != 0) return cmp;         //  if not equal, return result of comparison
    cmp = first <=> c.first;          // 2nd ordering criterion: firstname
    if (cmp != 0) return cmp;         //  if not equal, return result of comparison
    // 3rd ordering criterion: value
    // - use helper to map partial_ordering to strong_ordering:
    return std::strong_order(val, c.val);          // for types with <=>
    // - or if only < and == provided for the type:
    return std::compare_strong_order_fallback(val, c.val);   // works with < and ==
  }
  …
};
```

```
Cust c1{…};
Cust c2{…};

c1 < c2
// is rewritten as:
c1 <=> c2 < 0
```

honors rules for `double`/`float` (acc. to ISO/IEC/IEEE 60559)

Basic Operators (since C++17 / C++20)

C++17 / C++20

lower prio

Operator	Meaning	Remark
`.`, `->`, `[]` `()`	member/element access function call	evaluate args left-to-right funcexpr before args (still no order between args)
`++`, `--`	increment, decrement	`++x` differs from `x++`
`-`, `!`, `~`	negate, NOT, bitwise NOT	`+x` also supported
`.*`, `->*`	pointer-member access	evaluate args left-to-right
`*`, `/`, `%`	multiply, divide, modulo	with integral operands the result is integral
`+`, `-`	add, subtract	
`<<`, `>>`	shift and I/O	evaluate args left-to-right
C++20 `<=>`	less or equal or greater	needs `<compare>`
`<`, `<=`, `>=`, `>`	less and greater (or equal)	
`==`, `!=`	equal, not equal	
`&`	bitwise AND	
`^`	bitwise XOR	
`|`	bitwise OR	
`&&`	logical AND	evaluate args left-to-right until first `false`
`||`	logical OR	evaluate args left-to-right until first `true`
`=` `+=`, `-=`, `*=`, ...	assign	evaluate args right-to-left `x+=a` is equivalent to `x = x+a`
`?:`	conditional evaluation	(only op with 3 operands)
`,`	sequence of expressions	evaluate args left-to-right

C++20: Compatibility Issue with Operator ==

C++20

```
class MyType {
 private:
  int value;
 public:
  MyType(int i)      // implicit constructor from int
   : value{i} {
  }
  bool operator==(const MyType& rhs) const {   // enables MyType == int
    return value == rhs.value;
  }
};

bool operator==(int i, const MyType& t) {      // enables int == MyType
  return t == i;    // calls member function until C++17, since C++20: endless recursion
}

MyType x = 42;
if (x == 0) ...       // OK
if (0 == x) ...       // OK until C++17, since C++20: endless recursion
```

Remove or use `return t == MyType{i};`

Prefers rewritten `i == t` (better match than member function due to necessary type conversion)

http://stackoverflow.com/questions/65648897/c20-behaviour-breaking-existing-code-with-equality-operator

C++20: Using Feature Test Macros

C++20

```
class MyType {
 private:
  int value;
 public:
  MyType(int i)      // implicit constructor from int
   : value{i} {
  }
  bool operator==(const MyType& rhs) const {   // enables MyType == int
    return value == rhs.value;
  }
};

#ifndef __cpp_impl_three_way_comparison
bool operator==(int i, const MyType& t) {
  return t == i;    // calls member function
}
#endif

MyType x = 42;
if (x == 0) ...       // OK
if (0 == x) ...       // OK until C++17, since C++20: endless recursion
```

Feature test macros:

- Macros defined for each language and library feature
- May have different versions

http://stackoverflow.com/questions/65648897/c20-behaviour-breaking-existing-code-with-equality-operator

C++20: Compatibility Issue with Operator ==

C++20

- **Defaulted operator == needs operator == in the base class:**

```
class B {
};

class D : public B {
  int i;
 public:
  bool operator==(const D&) const = default;   // ERROR
};
```

- **Defaulted operator == does not work with protected operator == in the base class:**

```
class B {
 protected:
  bool operator==(const B&) const = default;
};

class D : public B {
  int i;
 public:
  bool operator==(const D&) const = default;   // ERROR
};
```

Possible workarounds:

Implement operator== in D
or:

```
class BFixed : public B {
 public:
  bool operator==(const BFixed& rhs) const {
    return Base::operator==(rhs);
  }
};

class D : public BFixed {
  int i;
 public:
  bool operator==(const D&) const = default;
};
```

C++20

Concepts, Constraints, and Requirements

Generic Function to Insert a Value

C++98

```
template<typename CollT, typename T>
void add(CollT& coll, const T& val)
{
  coll.push_back(val);
}
```

```
std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);     // OK
add(coll2, 42);     // ERROR: no push_back()
```

Overloading Function Templates

C++98

```
template<typename CollT, typename T>
void add(CollT& coll, const T& val)
{
  coll.push_back(val);
}

template<typename CollT, typename T>
void add(CollT& coll, const T& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);     // ERROR: two definitions of add()
add(coll2, 42);     // ERROR: two definitions of add()
```

Overload resolution cares only for declarations (ignoring return types)

Constraints with Concepts

C++20

```
template<typename CollT>
concept HasPushBack = requires (CollT c, CollT::value_type v) {
                        c.push_back(v);
                      };

template<typename CollT, typename T>
requires HasPushBack<CollT>
void add(CollT& coll, const T& val)
{
  coll.push_back(val);
}

template<typename CollT, typename T>
void add(CollT& coll, const T& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);     // OK, uses 1st  add() calling push_back()
add(coll2, 42);     // OK, uses 2nd add() calling insert()
```

Concept (named requirements)

Requirements with *requires expression*

Constraints formulated by a *requires clause*

Overload resolution prefers more specialized function

Concepts as Type Constraints

C++20

Concept (named requirements)

```
template<typename CollT>
concept HasPushBack = requires (CollT c, CollT::value_type v) {
                        c.push_back(v);
                      };
```

Requirements with ***requires expression***

```
template<HasPushBack CollT, typename T>
void add(CollT& coll, const T& val)
{
  coll.push_back(val);
}

template<typename CollT, typename T>
void add(CollT& coll, const T& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);     // OK, uses 1st  add() calling push_back()
add(coll2, 42);     // OK, uses 2nd add() calling insert()
```

Type Constraints with concepts applied to types

Overload resolution prefers more specialized function

Invalid Concepts

C++20

```
template<typename CollT>
concept HasPushBack = requires (CollT c, CollT::value_type v) {
                        c.pushback(v);   // OOPS: spelling error
                      };
```

```
template<HasPushBack CollT, typename T>
void add(CollT& coll, const T& val)
{
  coll.push_back(val);
}

template<typename CollT, typename T>
void add(CollT& coll, const T& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);     // ERROR: "can't call insert()"
add(coll2, 42);     // OK, uses 2nd add() calling insert()
```

Requirements not met
=> Concept not satisfied
=> 1st `add()` ignored

2nd `add()` is used, because concept for 1st `add()` not satisfied

Testing Concepts

C++20

```
template<typename CollT>
concept HasPushBack = requires (CollT c, CollT::value_type v) {
                        c.pushback(v);   // OOPS: spelling error
                      };
```

Concepts are compile-time Boolean values

```
// test code:
static_assert(HasPushBack<std::vector<int>>);

static_assert(!HasPushBack<std::set<int>>);

std::vector<int> coll1;
static_assert(HasPushBack<decltype(coll1)>);
```


Generic Function to Insert a Value

C++98

```
template<typename CollT, typename T>
void add(CollT& coll, const T& val)
{
  coll.push_back(val);
}
```

```
std::vector<int> coll;

add(coll, 42);     // OK
```

auto as Function Parameters

C++20

```
void add(auto& coll, const auto& val)
{
  coll.push_back(val);
}
```

"*Abbreviated function template*"

- Generic code
- Equivalent to:

```
template<typename T1, typename T2>
void add(T1& coll, const T2& val) {
  coll.push_back(val);
}
```

- Definition usually in header files
- No `inline` necessary

```
std::vector<int> coll;

add(coll, 42);     // OK
```


auto Parameter for Ordinary Functions

C++20

- **Generic lambda** (since C++14):

```
auto printColl = [] (const auto& coll) {
                   for (const auto& elem : coll) {
                     std::cout << elem << '\n';
                   }
                 };
```

- ***Abbreviated* function template** (since C++20):

```
void printColl(const auto& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << '\n';
  }
}
```

- Generic code
- Usually in header file

Equivalent to:

```
template <typename T>
void printColl(const T& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << '\n';
  }
}
```

```
std::vector<int> v;
...
printColl(v);          // OK for both
printColl(v);          // OK for both
printColl("hello");    // OK for both
```

Generic Lambda vs. Abbreviated Function Template

C++20

- **Generic lambda** (since C++14):

```
auto printLmbd = [] (const auto& coll) {
                   for (const auto& elem : coll) {
                     std::cout << elem << '\n';
                   }
                 };
```

Function object
Non-generic object
with generic `operator()`

- ***Abbreviated* function template** (since C++20):

```
void printAuto(const auto& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << '\n';
  }
}
```

Function template
Generic before called

Equivalent to:

```
template <typename T>
void printAuto(const T& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << '\n';
  }
}
```

```
std::vector<int> v;
...
printLmbd(v);                          // OK
printAuto(v);                          // OK
printLmbd<std::string>("hi");          // ERROR
printAuto<std::string>("hi");          // OK
call(printLmbd, v);                    // OK
call(printAuto, v);                    // ERROR
call(printAuto<decltype(v)>, v);       // OK
```


`auto` Defers Type Checks

C++

```
struct C2;

struct C1 {
  void foo(const C2& c2) const {
    c2.print();   // ERROR without definition
  }

  void print() const {
    ...
  }
};

struct C2 {
  void foo(const C1& c1) const {
    c1.print();   // OK
  }
  void print() const {
    ...
  }
};

C1 c1;
C2 c2;
c1.foo(c2);
c2.foo(c1);
```

auto Defers Type Checks

C++98 / C++20

C++98

```
struct C2;

struct C1 {
  void foo(const C2& c2) const {
    c2.print();  // ERROR without definition
  }
  void foo(const C2& c2) const;
  void print() const {
    …
  }
};

struct C2 {
  void foo(const C1& c1) const {
    c1.print();  // OK
  }
  void print() const {
    …
  }
};

inline void C1::foo(const C2& c2) const {
  c2.print();    // OK
}

C1 c1;
C2 c2;
c1.foo(c2);
c2.foo(c1);
```

C++20

```
struct C1 {
  void foo(const auto& c2) const {
    c2.print();  // OK
  }

  void print() const {
    …
  }
};

struct C2 {
  void foo(const auto& c1) const {
    c1.print();  // OK
  }
  void print() const {
    …
  }
};

C1 c1;
C2 c2;
c1.foo(c2);  // OK: auto deduced with call
c2.foo(c1);
```

You can constrain foo() with `std::same_as<C2> auto`

C++20: Functions with Variadic auto

C++20

Equivalent to:

```
template<typename... Types>
printArgs(const Types&... args)
```

```
void printArgs(const auto&... args)   // takes any number of arguments of any type
{
  std::cout << "args:\n";
  auto printLn = [] (const auto& arg) {
                   std::cout << "- " << arg << '\n';
                 };
  (... , printLn(args));              // call printLn() for each argument in args
}
```

Fold expression (C++17), expands to:

```
printLn(arg1), printLn(arg2), ..., printLn(argN);
```

```
printArgs(0, '&', "five");
```

Output:

```
args:
- 0
- &
- five
```

auto as Function Parameters

C++20

```
void add(auto& coll, const auto& val)
{
  coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // ERROR: two definitions of add()
add(coll2, 42);    // ERROR: two definitions of add()
```


Concepts as Type Constraints

C++20

Concept (named requirements)

```
template<typename CollT>
concept HasPushBack = requires (CollT c, CollT::value_type v) {
                        c.push_back(v);
                      };
```

Requirements with ***requires expression***

Type Constraints with concepts applied to types

```
void add(HasPushBack auto& coll, const auto& val)
{
  coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // OK, uses 1st add() calling push_back()
add(coll2, 42);    // OK, uses 2nd add() calling insert()
```

Equivalent to:

```
template<HasPushBack T1, typename T2>
void add(T1& coll, const T2& val) {
  coll.push_back(val);
}
```

Overload resolution prefers more specialized function

Concepts as Type Constraints

C++20

```
template<typename CollT>
concept HasPushBack = requires (CollT c, CollT::value_type v) {
  c.push_back(v);
};
```

No need of **typename** for type members of template parameters when it's clearly a type

```
void add(HasPushBack auto& coll, const auto& val)
{
  coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // OK, uses 1st add() calling push_back()
add(coll2, 42);    // OK, uses 2nd add() calling insert()
```

Equivalent to:

```
template<HasPushBack T1, typename T2>
void add(T1& coll, const T2& val) {
  coll.push_back(val);
}
```

Overload resolution prefers more specialized function

Concepts in `requires` Clauses

C++20

```
template<typename CollT>
concept HasPushBack = requires (CollT c, CollT::value_type v) {
  c.push_back(v);
};
```

```
void add(auto& coll, const auto& val)
requires HasPushBack<decltype(coll)>
{
  coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // ERROR: can't call insert()
add(coll2, 42);    // OK, uses 2nd add() calling insert()
```

`std::vector<int>&::value_type` is not valid

Concepts in `requires` Clauses

C++20

```
template<typename CollT>
concept HasPushBack = requires (CollT c, CollT::value_type v) {
  c.push_back(v);
};
```

```
void add(auto& coll, const auto& val)
requires HasPushBack<std::remove_cvref_t<decltype(coll)>>
{
  coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // OK, uses 1st add() calling push_back()
add(coll2, 42);    // OK, uses 2nd add() calling insert()
```


Concepts in `requires` Clauses

C++20

```
template<typename CollT>
concept HasPushBack = requires (CollT c,
                                std::remove_cvref_t<CollT>::value_type v) {
  c.push_back(v);
};
```

```
// test case:
static_assert(HasPushBack<std::vector<int>&>);
```

```
void add(auto& coll, const auto& val)
requires HasPushBack<decltype(coll)>
{
  coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // OK, uses 1st add() calling push_back()
add(coll2, 42);    // OK, uses 2nd add() calling insert()
```

Concepts in `requires` Clauses

C++20

```
template<typename CollT>
concept HasPushBack = requires (CollT c,
                                std::ranges::range_value_t<CollT> v) {
  c.push_back(v);
};
```

Ranges library utility
- Needs: `#include <ranges>`

```
void add(auto& coll, const auto& val)
requires HasPushBack<decltype(coll)>
{
  coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // OK, uses 1st add() calling push_back()
add(coll2, 42);    // OK, uses 2nd add() calling insert()
```


Concepts for Multiple Parameters

C++20

```
template<typename CollT, typename T>
concept CanPushBack = requires (CollT c, T v) {
  c.push_back(v);
};
```

Concept for multiple parameters:
"Can we can `push_back()` a `T` in a `CollT` ?"

```
template<typename CollT, typename T>
requires CanPushBack<CollT, T>
void add(CollT& coll, const T& val)
{
  coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // OK, uses 1st add() calling push_back()
add(coll2, 42);    // OK, uses 2nd add() calling insert()
```

Constraint for multiple parameters:
"Provided we can `push_back()` a `T` in a `CollT`"

Concepts for Multiple Parameters

C++20

```
template<typename CollT, typename T>
concept CanPushBack = requires (CollT c, T v) {
                        c.push_back(v);
                      };
```

```
void add(auto& coll, const auto& val)
requires CanPushBack<decltype(coll), decltype(val)>
{
  coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // OK, uses 1st add() calling push_back()
add(coll2, 42);    // OK, uses 2nd add() calling insert()
```


Granularity of Concepts

C++20

```
template<typename CollT, typename T>
concept CanPushBack = requires (CollT c, T v) {
                        c.push_back(v);
                      };
```

```
void add(auto& coll, const auto& val)
requires CanPushBack<decltype(coll), decltype(val)>
{
  coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // OK, uses 1st add() calling push_back()
add(coll2, 42);    // OK, uses 2nd add() calling insert()
```

Don't introduce a concept for each statement
(too fine grained)

Granularity of Concepts

C++20

Standard concept for iterating over elements

```
template<typename CollT>
concept SequenceCont = std::ranges::range<CollT> &&
requires (std::remove_cvref_t<CollT> c,
          std::ranges::range_value_t<CollT> v) {
   c.push_back(v);
   c.pop_back();
   c.insert(c.begin(), v);
   c.erase(c.begin());
   c.clear();
   std::remove_cvref_t<CollT>{v, v, v};   // init-list support
   c = {v, v, v};
   {c < c} -> std::convertible_to<bool>;
   ...
};
```

Combine multiple requirements into general-purpose concepts

```
template<typename CollT, typename T>
requires SequenceCont<CollT>
void add(CollT& coll, const T& val)
{
  coll.push_back(val);
}
```


`requires` Expression and `requires` Clause

C++20

```
template<typename CollT, typename T>
concept CanPushBack = requires (CollT c, T v) {
                        c.push_back(v);
                      };
```

- ***Requires expression*** defines ***requirements***

```
void add(auto& coll, const auto& val)
requires CanPushBack<decltype(coll), decltype(val)>
{
  coll.push_back(val);
}
```

- ***Requires clause*** defines ***constraints***

```
void add(auto& coll, const auto& val)
{
  coll.insert(val);
}
```

- Requirements are
- Concepts are
- Constraints process

compile-time Boolean values

```
std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // OK, uses 1st add() calling push_back()
add(coll2, 42);    // OK, uses 2nd add() calling insert()
```

Combining `requires` Expression and `requires` Clause

C++20

```
void add(auto& coll, const auto& val)
requires requires { coll.push_back(val); }
{
  coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // OK, uses 1st add() calling push_back()
add(coll2, 42);    // OK, uses 2nd add() calling insert()
```

- ***Requires expression*** defines ***requirements***
- ***Requires clause*** defines ***constraints***

- Requirements are
- Concepts are
- Constraints process

compile-time Boolean values

`requires` and Compile-Time `if`

C++20

```
void add(auto& coll, const auto& val)
{
  if constexpr (requires { coll.push_back(val); }) {
    coll.push_back(val);
  }
  else {
    coll.insert(val);
  }
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // OK, calls push_back()
add(coll2, 42);    // OK, calls insert()
```

- Requirements are
- Concepts are
- Constraints process

compile-time Boolean values

Concepts and Error Messages

C++20

```
void add(auto& coll, const auto& val)
{
  if constexpr (requires { coll.push_back(val); }) {
    coll.push_back(val);
  }
  else {
    coll.insert(val);
  }
}

std::vector<int> coll1;
std::set<std::string> coll2;

add(coll1, 42);    // OK, calls push_back()
add(coll2, 42);    // ERROR
```


Possible Error Message:

```
prog.cpp:16:10: error: no matching member function for call to 'insert'
    coll.insert(val);
    ~~~~~^~~~~~
prog.cpp:30:1: note: in instantiation of function template specialization
'add<std::set<std::basic_string<char>>, int>' requested here
add(coll2, 42);
^
/include/c++/12.0.1/bits/stl_set.h:509:7: note: candidate function not viable: no
known conversion from 'const int' to 'const
std::set<std::basic_string<char>>::value_type' (aka 'const std::basic_string<char>')
for 1st argument
      insert(const value_type& __x)
      ^
/include/c++/12.0.1/bits/stl_set.h:518:7: note: candidate function not viable: no
known conversion from 'const int' to 'std::set<std::basic_string<char>>::value_type'
(aka 'std::basic_string<char>') for 1st argument
      insert(value_type&& __x)
      ^
...
/include/c++/12.0.1/bits/stl_set.h:603:7: note: candidate function not viable:
requires 2 arguments, but 1 was provided
      insert(const_iterator __hint, node_type&& __nh)
      ^
1 error generated.
```

```
void add(auto& col
{
  if constexpr (re
    coll.push_back
  }
  else {
    coll.insert(va
  }
}

std::vector<int> coll1;
std::set<std::string> coll2;

add(coll1, 42);    // OK, calls push_back()
add(coll2, 42);    // ERROR in the code of std::set<> when calling insert()
```

Concepts and Error Messages

C++20

```
template<std::ranges::range CollT, typename T>
void add(CollT& coll, const T& val)
requires std::convertible_to<T, std::ranges::range_value_t<CollT>>
{
  if constexpr (requires { coll.push_back(val); }) {
    coll.push_back(val);
  }
  else {
    coll.insert(val);
  }
}

std::vector<int> coll1;
std::set<std::string> coll2;

add(coll1, 42);    // OK, calls push_back()
add(coll2, 42);    // ERROR when calling add()
```


Concepts and Error Messages

C++20

Possible Error Message (clang):

```
prog.cpp:30:1: error: no matching function for call to 'add'
add(coll2, 42);
^~~
prog.cpp:9:6: note: candidate template ignored: constraints not satisfied [with CollT
= std::set<std::basic_string<char>>, T = int]
void add(CollT& coll, const T& val)
     ^
prog.cpp:10:15: note: because 'std::convertible_to<int,
std::ranges::range_value_t<set<basic_string<char> > > >' evaluated to false
requires std::convertible_to<T, std::ranges::range_value_t<CollT>>
               ^
/include/c++/12.0.1/concepts:72:30: note: because 'is_convertible_v<int,
std::basic_string<char> >' evaluated to false
    concept convertible_to = is_convertible_v<_From, _To>
                             ^
1 error generated.
```

```
template<std::rang
void add(CollT& co
requires std::conv
{
  if constexpr (re
    coll.push_back
  }
  else {
    coll.insert(va
  }
}

std::vector<int> coll1;
std::set<std::string> coll2;

add(coll1, 42);    // OK, calls push_back()
add(coll2, 42);    // ERROR when calling add()
```

C++20

The Ranges Library

Ranges

- **Ranges**
  - Objects that have elements to iterate over
- **Ranges Library**
  - Content:
    - **Concepts**
    - **Algorithms**
    - **Utilities**
    - **Views**
      - Adaptors that can be combined to pipelines with operator |
  - Header: `<ranges>`
  - Namespace: `std::ranges` (and `std::views`)

Let's Insert Elements

C++20

```
template<typename CollT, typename T>
void add(CollT& coll, const T& val)
{
  if constexpr (requires { coll.push_back(val); }) {
    coll.push_back(val);
  }
  else {
    coll.insert(val);
  }
}
```

```
std::vector<int> coll1;
std::list<int> coll2;
add(coll1, 42);     // OK, calls push_back()
add(coll2, 42);     // OK, calls push_back()
add(coll1, "42");   // ERROR deep inside coll.push_back(val)
```


Let's Insert Elements

C++20

```
template<typename
void add(CollT& co
{
  if constexpr (re
    coll.push_back
  }
  else {
    coll.insert(va
  }
}
```

```
std::vector<int> c
std::list<int> col
add(coll1, 42);
add(coll2, 42);
add(coll1, "42");
```

clang

```
insert.cpp:13:10: error: no matching member function for call to 'insert'
   13 |     coll.insert(val);
      |     ~~~~~^~~~~
insert.cpp:25:1: note: in instantiation of function template specialization
'add<std::vector<int>, char[3]>' requested here
   25 | add(coll1, "42");  // ERROR deep inside coll.push_back(val)
      | ^
/opt/clang/bin/../include/c++/v1/vector:662:64: note: candidate function not viable:
requires 2 arguments, but 1 was provided
  662 |   _LIBCPP_CONSTEXPR_SINCE_CXX20 _LIBCPP_HIDE_FROM_ABI iterator insert
(const_iterator __position, const_reference __x);
      |                                                                ^
/opt/clang/bin/../include/c++/v1/vector:664:64: note: candidate function not viable:
requires 2 arguments, but 1 was provided
  664 |   _LIBCPP_CONSTEXPR_SINCE_CXX20 _LIBCPP_HIDE_FROM_ABI iterator insert
(const_iterator __position, value_type&& __x);
      |                                                                ^
/opt/clang/bin/../include/c++/v1/vector:701:3: note: candidate function not viable:
requires 2 arguments, but 1 was provided
  701 |   insert(const_iterator __position, initializer_list<value_type> __il) {
      |   ^
/opt/clang/bin/../include/c++/v1/vector:669:3: note: candidate function not viable:
requires 3 arguments, but 1 was provided
  669 |   insert(const_iterator __position, size_type __n, const_reference __x);
      |   ^
/opt/clang/bin/../include/c++/v1/vector:676:3: note: candidate function template not viable:
requires 3 arguments, but 1 was provided
  676 |   insert(const_iterator __position, _InputIterator __first, _InputIterator __last);
      |   ^
/opt/clang/bin/../include/c++/v1/vector:697:3: note: candidate function template not viable:
requires 3 arguments, but 1 was provided
  697 |   insert(const_iterator __position, _ForwardIterator __first, _ForwardIterator __last);
      |   ^
1 error generated.
Compiler returned: 1
```

Let's Insert Elements

C++20

```
template<std::ranges::range CollT, typename T>
void add(CollT& coll, const T& val)
requires std::convertible_to<T, std::ranges::range_value_t<CollT>>
{
  if constexpr (requires { coll.push_back(val); }) {
    coll.push_back(val);
  }
  else {
    coll.insert(val);
  }
}
```

Range concepts

Range utilities

```
std::vector<int> coll1;
std::list<int> coll2;
add(coll1, 42);     // OK, calls push_back()
add(coll2, 42);     // OK, calls push_back()
add(coll1, "42");   // ERROR: no matching add() found: convertible_to<> not satisfied
```


Let's Insert Elements

C++20

```
template<std::ranges::range C
void add(CollT& coll, const T
requires std::convertible_to<
{
  if constexpr (requires { col
    coll.push_back(val);
  }
  else {
    coll.insert(val);
  }
}
```

clang

```
insert.cpp:25:1: error: no matching function for call to 'add'
   25 | add(coll1, "42");  // ERROR deep inside coll.push_back(val)
      | ^~~
insert.cpp:7:6: note: candidate template ignored: constraints not satisfied
[with CollT = std::vector<int>, T = char[3]]
    7 | void add(CollT& coll, const T& val)
      |      ^
insert.cpp:8:10: note: because 'std::convertible_to<char[3],
std::ranges::range_value_t<vector<int, allocator<int> > > >' evaluated to false
    8 | requires std::convertible_to<T, std::ranges::range_value_t<CollT>>
      |          ^
/opt/clang/bin/../include/c++/v1/__concepts/convertible_to.h:27:26:
note: because 'is_convertible_v<char[3], int>' evaluated to false
   27 | concept convertible_to = is_convertible_v<_From, _To> && requires {
static_cast<_To>(std::declval<_From>()); };
      |                          ^
1 error generated.
Compiler returned: 1
```

```
std::vector<int> coll1;
std::list<int> coll2;
add(coll1, 42);     // OK, calls push_back()
add(coll2, 42);     // OK, calls push_back()
add(coll1, "42");   // ERROR: no matching add() found: convertible_to<> not satisfied
```

Let's Sort Elements

C++20

```
template<std::ranges::range CollT, typename T>
void add(CollT& coll, const T& val)
requires std::convertible_to<T, std::ranges::range_value_t<CollT>>
{
  if constexpr (requires { coll.push_back(val); } ) {
    coll.push_back(val);
  }
  else {
    coll.insert(val);
  }
}
```

```
std::vector<int> coll1;
std::list<int> coll2;
add(coll1, 42);     // OK, calls push_back()
add(coll2, 42);     // OK, calls push_back()

std::sort(coll1.begin(), coll1.end());          // OK, sorts the elements the C++98 way
std::sort(coll2.begin(), coll2.end());          // ERROR deep in header <list>
```


Let's Sort Elements

C++20

```
template<std::rang
void add(CollT& co
requires std::conv
{
  if constexpr (re
    coll.push_back
  }
  else {
    coll.insert(va
  }
}
```

```
std::vector<int> c
std::list<int> col
add(coll1, 42);
add(coll2, 42);

std::sort(coll1.be
std::sort(coll2.be
```

clang

```
In file included from insert.cpp:5:
In file included from /opt/clang/bin/../include/c++/v1/algorithm:1831:
/opt/clang/bin/../include/c++/v1/__algorithm/make_heap.h:35:34: error: invalid operands to
binary expression ('std::__list_iterator<int, void *>' and 'std::__list_iterator<int, void *>')
   35 |   difference_type __n   = __last - __first;
      |                           ~~~~~~ ^ ~~~~~~~

/opt/clang/bin/../include/c++/v1/__algorithm/partial_sort.h:41:8: note: in instantiation of
function template specialization 'std::__make_heap<std::_ClassicAlgPolicy, std::__less<void,
void> &, std::__list_iterator<int, void *>>' requested here
   41 |   std::__make_heap<_AlgPolicy>(__first, __middle, __comp);
      |        ^

/opt/clang/bin/../include/c++/v1/__algorithm/partial_sort.h:65:12: note: in instantiation of
function template specialization 'std::__partial_sort_impl<std::_ClassicAlgPolicy,
std::__less<void, void> &, std::__list_iterator<int, void *>, std::__list_iterator<int, void
*>>' requested here
   65 |       std::__partial_sort_impl<_AlgPolicy>(__first, __middle, __last,
static_cast<__comp_ref_type<_Compare> >(__comp));
      |            ^

/opt/clang/bin/../include/c++/v1/__algorithm/sort.h:992:10: note: in instantiation of function
template specialization 'std::__partial_sort<std::_ClassicAlgPolicy, std::__less<void, void>,
std::__list_iterator<int, void *>, std::__list_iterator<int, void *>>' requested here
  992 |     std::__partial_sort<_AlgPolicy>(
      |          ^

/opt/clang/bin/../include/c++/v1/__algorithm/sort.h:1003:8: note: in instantiation of function
template specialization 'std::__sort_impl<std::_ClassicAlgPolicy, std::__list_iterator<int,
void *>, std::__less<void, void>>' requested here
 1003 |   std::__sort_impl<_ClassicAlgPolicy>(std::move(__first), std::move(__last), __comp);
      |        ^

/opt/clang/bin/../include/c++/v1/__algorithm/sort.h:1009:8: note: in instantiation of function
template specialization 'std::sort<std::__list_iterator<int, void *>, std::__less<void, void>>'
requested here
 1009 |   std::sort(__first, __last, __less<>());
      |        ^

insert.cpp:27:6: note: in instantiation of function template specialization
'std::sort<std::__list_iterator<int, void *>>' requested here
   27 | std::sort(coll2.begin(), coll2.end());         // ERROR deep in header <list>
      |     ^
/opt/clang/bin/../include/c++/v1/__ios/fpos.h:60:40: note: candidate template ignored: could
not match 'fpos' against '__list_iterator'
... (188 lines)
```

Range Algorithms Use Standard Concepts

C++20

```
template<std::ranges::range CollT, typename T>
void add(CollT& coll, const T& val)
requires std::convertible_to<T, std::ranges::range_value_t<CollT>>
{
  if constexpr (requires { coll.push_back(val); }) {
    coll.push_back(val);
  }
  else {
    coll.insert(val);
  }
}
```

Definition of `std::ranges::sort()` for iterators:

```
namespace std::ranges {
  template<std::random_access_iterator I,
           std::sentinel_for<I> S,
           typename Comp = std::ranges::less, ...>
  requires std::sortable<std::ranges::iterator_t<R>,
                         Comp, ...>
  ... sort(I beg, S end, Comp comp = {}, ...);
}
```

```
std::vector<int> coll1;
std::list<int> coll2;
add(coll1, 42);     // OK, calls push_ba
add(coll2, 42);     // OK, calls push_b

std::sort(coll1.begin(), coll1.end());          // OK, sorts the elements the C++98 way
std::sort(coll2.begin(), coll2.end());          // ERROR deep in header <list>
std::ranges::sort(coll2.begin(), coll2.end()); // ERROR: "no random_access_iterator"
```


Range Algorithms Use Standard Concepts

C++20

```
template<std::ranges::range CollT, typename T>
void add(CollT& coll, const T& val)
requires std::convertible_to<T, std::ranges::range_value_t<CollT>>
{
  if constexpr (requires { coll.push_back(val); }) {
    coll.push_back(val);
  }
  else {
    coll.insert(val);
  }
}
```

Definition of `std::ranges::sort()` for iterators:

```
namespace std::ranges {
  template<std::random_access_iterator I,
           std::sentinel_for<I> S,
           typename Comp = std::ranges::less, ...>
  requires std::sortable<std::ranges::iterator_t<R>,
                         Comp, ...>
  ... sort(I beg, S end, Comp comp = {}, ...);
}
```

```
std::vector<int> coll1;
std::list<int> coll2;
add(coll1, 42);     // OK, calls push_ba
add(coll2, 42);     // OK, calls push_b

std::sort(coll1.begin(), coll1.end());          // OK, sorts the elements the C++98 way
std::sort(coll2.begin(), coll2.end());          // ERROR deep in header <list>
std::ranges::sort(coll2.begin(), coll2.end()); // ERROR: "no random_access_iterator"
```

C++
©2024 by josuttis.com
80
josuttis | eckstein
IT communication

Sentinels

C++20

```
void print(auto beg, auto end) {
  for (auto pos = beg; pos != end; ++pos) {
    std::cout << *pos << ' ';
  }
  std::cout << '\n';
}
```

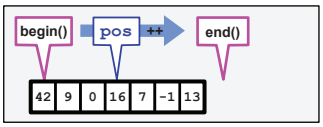
C++20 rewriting converts:
`pos != end`
to:
`!(end == pos)`

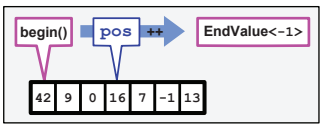
```
template<auto Val>
struct EndValue {
  bool operator== (auto pos) const {
    return *pos == Val;   // Val is end value
  }
};
```

Sentinel type for values

```
std::vector coll{42, 9, 0, 16, 7, -1, 13};
print(coll.begin(), coll.end());        // sentinel with same type
print(coll.begin(), EndValue<-1>{});    // sentinel has other type
```

Sentinel (end of range)






```
std::sort(coll.begin(), EndValue<-1>{});          // Error: std::sort() needs same type
std::ranges::sort(coll.begin(), EndValue<-1>{}); // OK, sorts all elems before -1
```


C++20: Algorithms for Ranges as a Whole

C++20

- **Algorithms for single-argument ranges**
  - Declared in header `<algorithm>` in namespace `std::ranges`
  - Support for all kind of ranges (including arrays)
    - Provided the range supports all requirements
- **No support yet for**
  - Numeric algorithms (e.g. `accumulate()`)
  - Parallel execution

```
std::vector coll{42, 8, 0, 15, 7, -1, 15};

std::sort(coll.begin(), coll.end());   // sort elements in coll since C++98
std::ranges::sort(coll);               // sort elements in coll since C++20

std::string arr[] {"Rio", "Berlin", "LA"};
std::ranges::sort(arr);                // sort elements in arr since C++20
```

Range Algorithms

C++20

```
template<std::ranges::range CollT, typename T>
void add(CollT& coll, const T& val)
requires std::convertible_to<T, std::ranges::range_value_t<CollT>>
{
  if constexpr (requires { coll.push_back(val); }) {
    coll.push_back(val);
  }
  else {
    coll.insert(val);
  }
}
```

```
std::vector<int> coll1;
std::list<int> coll2;
add(coll1, 42);     // OK, calls push_back()
add(coll2, 42);     // OK, calls push_back()

std::sort(coll1.begin(), coll1.end());          // OK, sorts the elements the C++98 way
std::sort(coll2.begin(), coll2.end());          // ERROR deep in header <list>
std::ranges::sort(coll2.begin(), coll2.end()); // ERROR: "no random_access_iterator"
std::ranges::sort(coll1);                       // OK, sorts the range as a whole
```


Range Algorithms Use Standard Concepts

C++20

```
template<std::ranges::range CollT, typename T>
void add(CollT& coll, const T& val)
requires std::convertible_to<T, std::ranges::range_value_t<CollT>>
{
  if constexpr (requires { coll.push_back(val); }) {
    coll.push_back(val);
  }
  else {
    coll.insert(val);
  }
}
```

Definition of `std::ranges::sort()` for ranges:

```
namespace std::ranges {
  template<std::ranges::random_access_range R,
           typename Comp = std::ranges::less, ...>
  requires std::sortable<std::ranges::iterator_t<R>,
                         Comp, ...>
  ... sort(R&& r, Comp comp = {}, ...);
}
```

```
std::vector<int> coll1;
std::list<int> coll2;
add(coll1, 42);     // OK, calls push_back()
add(coll2, 42);     // OK, calls push_back()

std::sort(coll1.begin(), coll1.end());          // OK, sorts the elements the C++98 way
std::sort(coll2.begin(), coll2.end());          // ERROR deep in header <list>
std::ranges::sort(coll2.begin(), coll2.end()); // ERROR: "no random_access_iterator"
std::ranges::sort(coll1);                       // OK, sorts the range as a whole
std::ranges::sort(coll2);                       // ERROR : "no random_access_range"
```

Using Operator <=> for `sort()`

C++20

```
class Person {
  std::string first;  // firstname
  std::string last;   // lastname
 public:
  Person(std::string f, std::string l)
   : first{std::move(f)}, last{std::move(l)} {
  }
  ...
  bool operator== (const Person& p2) const {
    return last == p2.last;    // equality uses lastname only
  }
  auto operator<=> (const Person& p2) const {
    return last <=> p2.last;   // ordering uses lastname only
  }
};
```

Used by ==, !=

Used by <, <=, >, >=

```
std::vector<Person> coll{{"Aretha", "Franklin"}, {"Muddy", "Waters"},
                         {"Bob", "Dylan"}, {"Ray", "Charles"}};
Person p{"Elton", "John"};
if (p > coll[0]) ...                          // OK, rewritten as: p <=> coll[0] < 0
std::sort(coll.begin(), coll.end());          // OK
std::ranges::sort(coll.begin(), coll.end());  // OK
std::ranges::sort(coll);                      // OK
```


Using Operator < for `sort()`

C++20

```
class Person {
  std::string first;  // firstname
  std::string last;   // lastname
 public:
  Person(std::string f, std::string l)
   : first{std::move(f)}, last{std::move(l)} {
  }
  ...
  bool operator== (const Person& p2) const {
    return last == p2.last;    // equality uses lastname only
  }
  bool operator< (const Person& p2) const {
    return last < p2.last;     // ordering uses lastname only
  }
};
```

Used by ==, !=

`sortable` requires all operators <, >, <=, >=

```
std::vector<Person> coll{{"Aretha", "Franklin"}, {"Muddy", "Waters"},
                         {"Bob", "Dylan"}, {"Ray", "Charles"}};
Person p{"Elton", "John"};
if (p > coll[0]) ...                          // ERROR: no operator > defined
std::sort(coll.begin(), coll.end());          // OK
std::ranges::sort(coll.begin(), coll.end());  // ERROR: concept sortable not satisfied
std::ranges::sort(coll);                      // ERROR: concept sortable not satisfied
```

The Various Ways to Call Algorithms

C++98 / C++17 / C++20

```
std::sort(coll.begin(), coll.end());                                  // begin and end of range

std::sort(coll.begin(), coll.end(),                                   // begin and end of range
          [] (const auto& x, const auto& y) { return f(x) < f(y); }); // sort criterion
```

```
std::sort(std::execution::par,                  // with parallel execution
          coll.begin(), coll.end());

std::sort(std::execution::par,                  // with parallel execution and sort criterion
          coll.begin(), coll.end(),
          [] (const auto& x, const auto& y) { return f(x) < f(y); });
```

since C++17

```
std::ranges::sort(coll.begin(), coll.end());    // supporting concepts and sentinels (end of other type)

std::ranges::sort(coll.begin(), coll.end(),     // supporting concepts, sentinels, and sort criterion
                  [] (const auto& x, const auto& y) { return f(x) < f(y); });

std::ranges::sort(coll.begin(), coll.end(),     // supporting concepts, sentinels, sort criterion, projection
                  [] (const auto& x, const auto& y) { return x < y; },   // or std::less{}
                  [] (const auto& val) { return f(val); });

std::ranges::sort(coll);                        // for ranges supporting concepts

std::ranges::sort(coll,                         // for ranges supporting sort criterion and projection
                  [] (const auto& x, const auto& y) { return f(x) < f(y); });

std::ranges::sort(coll,                         // for ranges supporting concepts, sort criterion, projection
                  [] (const auto& x, const auto& y) { return x < y; },   // or std::less{}
                  [] (const auto& val) { return f(val); });
```

since C++20

Namespace Mess

C++20

- **For ranges you need different namespaces**
  - `std::ranges` for direct support of ranges
  - `std::views` to create views (alias for `std::ranges::views`)
  - `std::` for other and old standard features
- **For example:**
  - Concepts `std::forward_iterator`, but `std::ranges::forward_range`
  - Type trait ~~`std::iter_value_t`~~, but `std::ranges::range_value_t`
  - ~~`std::begin()`~~ and `std::ranges::begin()`
  - ~~`std::equal_to()`~~ and `std::ranges::equal_to()`
  - ~~`std::advance()`~~ and `std::ranges::advance()`

Prefer `std::ranges::`
(usually better than `std::`)

```
template<std::ranges::input_range R,
         typename T,
         typename Proj = std::identity>
requires std::indirect_binary_predicate<std::ranges::equal_to,
                                        std::projected<std::ranges::iterator_t<R>,
                                                       Proj>,
                                        const T*>
constexpr std::ranges::borrowed_iterator_t<R>
find (R&& r, const T& value, Proj proj = {});
```

C++20

Views

C++20: Views


```
void print(const std::ranges::input_range auto& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

```
std::vector<int> coll1{0, 8, 15, 47, 11, 42, 1};
std::set<int> coll2{0, 8, 15, 47, 11, 42, 1};
print(coll1);
print(coll2);
print(std::views::take(coll1, 3));       // print first three elements
print(std::views::take(coll2, 3));       // print first three elements
print(coll1 | std::views::take(3));      // print first three elements
print(coll2 | std::views::take(3));      // print first three elements

print(coll2 | std::views::take(3)
            | std::views::transform([](auto v){
                                      return std::to_string(v) + 's';
                                    }));
```

Output:

```
0 8 15 47 11 42 1
0 1 8 11 15 42 47
0 8 15
0 1 8
0 8 15
0 1 8
0s 1s 8s
```

Example of Pipeline of Range Adaptors

C++20

```
int main()
{
  std::map<std::string, int> composers {
    {"Bach", 1685},
    {"Mozart", 1756},
    {"Beethoven", 1770},
    {"Tchaikovsky", 1840},
    {"Chopin", 1810},
    {"Vivaldi", 1678},
  };

  // iterate over the names of the first 3 composers since 1700:
  namespace vws = std::views;
  for (const auto& elem : composers
   | vws::filter([](const auto& y) {   // since 1700
                   return y.second >= 1700;
                 })
   | vws::take(3)                      // first 3
   | vws::keys                         // names only
                                 ) {
    std::cout << "- " << elem << '\n';
  }
}
```

Output:

```
- Beethoven
- Chopin
- Mozart
```


Using a Pipeline of Views

C++20

```
// view 4th to 11th value that are multiples of 3 with suffix "s":
auto v = std::views::iota(1)                          // generates 1, 2, 3, ...
         | std::views::filter([] (auto val) {         // multiples of 3 only
                                return val % 3 == 0;
                              })
         | std::views::drop(3)                        // skip first 3
         | std::views::take(8)                        // take next 8
         | std::views::transform([] (auto val) {      // append "s"
                                   return std::to_string(val) + "s";
                                 });

for (const auto& elem : v) {
  std::cout << elem << '\n';
}
```

Output:

```
12s
15s
18s
...
33s
```

Using a Pipeline of Views

C++20

```
// view 4th to 11th value that are multiples of 3 with suffix "s":
auto v = std::views::iota(1)                          // generates 1, 2, 3, ...
         | std::views::filter([] (auto val) {         // multiples of 3 only
                                return val % 3 == 0;
                              })
         | std::views::drop(3)                        // skip first 3
         | std::views::take(8)                        // take next 8
         | std::views::transform([] (auto val) {      // append "s"
                                   return std::to_string(val) + "s";
                                 });

for (const auto& elem : v) {
  std:
}
```

Output:

```
12s
```

Type of v is:

```
std::ranges::transform_view<
 std::ranges::take_view<
  std::ranges::drop_view<
   std::ranges::filter_view<
    std::ranges::iota_view<int, std::unreachable_sentinel_t>,
    typeOf1stLambda>>>,
 typeOf2ndLambda>
```


Using Zip Views

C++23

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>

int main()
{
  std::vector<std::string> coll{"tic", "tac", "toe"};

  // print elements with position:
  int i = 0;
  for (const auto& elem : coll) {
    std::cout << ++i << ": " << elem << '\n';
  }

  // print elements with position using the zip view (since C++23):
  for (const auto& elem : std::views::zip(std::views::iota(1), coll)) {
    std::cout << elem.first << ": " << elem.second << '\n';
  }
}
```

Output:

```
1: tic
2: tac
3: toe

1: tic
2: tac
3: toe
```

Using Zip Views

C++23

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>

int main()
{
  std::vector<std::string> coll{"tic", "tac", "toe"};

  // print elements with position:
  for (int i = 0; const auto& elem : coll) {
    std::cout << ++i << ": " << elem << '\n';
  }

  // print elements with position using the zip view (since C++23):
  for (const auto& [i, elem] : std::views::zip(std::views::iota(1), coll)) {
    std::cout << i << ": " << elem << '\n';
  }
}
```

Range-based `for` loop with initialization (C++20)

Structured bindings (C++17)

Output:

```
1: tic
2: tac
3: toe

1: tic
2: tac
3: toe
```


C++20: Standard Filtering and Transforming Views

C++20

- **Range adaptors supporting operator |**

Adaptor for |	Type	Effect
take(*num*)	**take**_view	First *num* elements of a range
take_while(*pred*)	**take_while**_view	Leading elements that match a predicate
drop(*num*)	**drop**_view	All except the first *num* elements of a range
drop_while(*pred*)	**drop_while**_view	All except all leading elements that match a predicate
filter(*pred*)	**filter**_view	All elements that match a predicate
transform(*func*)	**transform**_view	All elements transformed by *func*
elements<*idx*>	**elements**_view	*idx*-th member of each tuple-like element
keys	**keys**_view (alias)	1st member of each tuple-like element
values	**values**_view (alias)	2nd member of each tuple-like element
reverse	**reverse**_view	All elements in reverses order
split(*sep*)	**split**_view	All elements split into sub-ranges
lazy_split(*sep*)	**lazy_split**_view	Same for input ranges (with P2210)
join	**join**_view	All elements of multiple ranges

C++20

How Views Operate

C++20: How Views Operate


```
std::vector<int> coll{47, 11, 0, 88, 15, 20};
print(coll);
```

print():

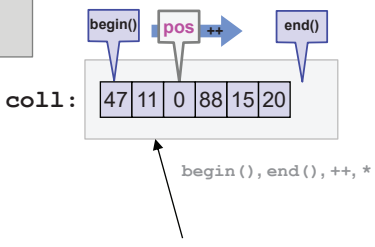
```
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
  std::cout << *pos << '\n';
}
```

print() step by step:

```
auto pos = coll.begin();
if (pos != coll.end())
  std::cout << *pos << '\n';
  ++pos;
if (pos != coll.end())
  std::cout << *pos << '\n';
  ++pos;
if (pos != coll.end())
  std::cout << *pos << '\n';
  ++pos;
...
```

Output:

```
47 11 0 88 15 20
47
11
0
...
```

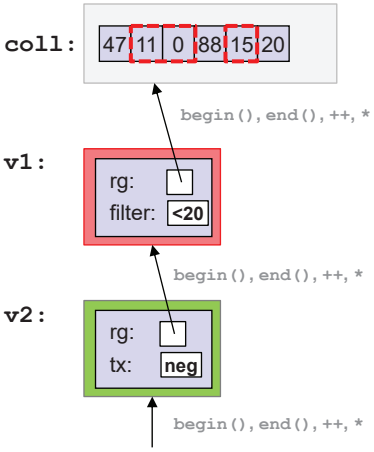


C++20: How Views Operate

C++20

```
std::vector<int> coll{47, 11, 0, 88, 15, 20};
print(coll);

auto less20 = [] (auto val) { return val < 20; };
print(coll | std::views::filter(less20)
           | std::views::transform(std::negate{}));

  auto v1 = std::views::filter(coll, less20);
  auto v2 = std::views::transform(v1, std::negate{});
```

Output:
```
47 11 0 88 15 20
-11 0 -15
```



C++20: How Views Operate

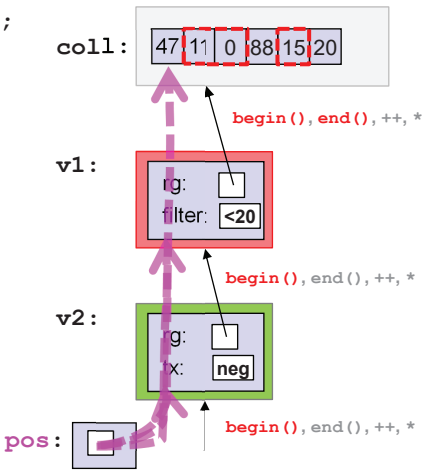
C++20

```
std::vector<int> coll{47, 11, 0, 88, 15, 20};
print(coll);

auto less20 = [] (auto val) { return val < 20; };
print(coll | std::views::filter(less20)
           | std::views::transform(std::negate{}));

  auto v1 = std::views::filter(coll, less20);
  auto v2 = std::views::transform(v1, std::negate{});
```

Output:
```
47 11 0 88 15 20
-11 0 -15
```

print() step by step:

```
auto pos = v2.begin();
```



C++20: How Views Operate

C++20

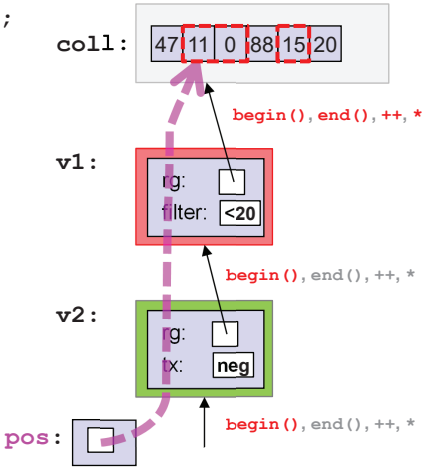
```
std::vector<int> coll{47, 11, 0, 88, 15, 20};
print(coll);

auto less20 = [] (auto val) { return val < 20; };
print(coll | std::views::filter(less20)
           | std::views::transform(std::negate{}));
```

Output:
```
47 11 0 88 15 20
-11 0 -15
```

```
auto v1 = std::views::filter(coll, less20);
auto v2 = std::views::transform(v1, std::negate{});
```

print() step by step:

```
auto pos = v2.begin();
```



C++20: How Views Operate

C++20

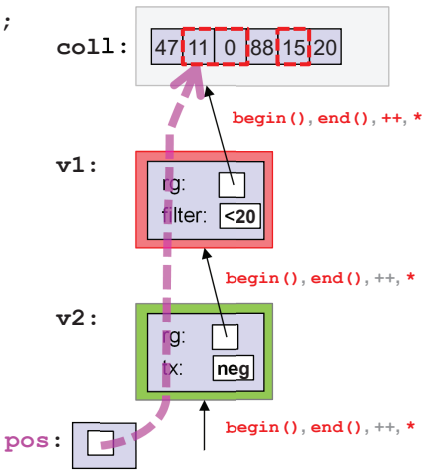
```
std::vector<int> coll{47, 11, 0, 88, 15, 20};
print(coll);

auto less20 = [] (auto val) { return val < 20; };
print(coll | std::views::filter(less20)
           | std::views::transform(std::negate{}));
```

Output:
```
47 11 0 88 15 20
-11 0 -15
-11
```

```
auto v1 = std::views::filter(coll, less20);
auto v2 = std::views::transform(v1, std::negate{});
```

print() step by step:

```
auto pos = v2.begin();
if (pos != v2.end())
  std::cout << *pos << '\n';
```



C++20: How Views Operate

C++20

```
std::vector<int> coll{47, 11, 0, 88, 15, 20};
print(coll);

auto less20 = [] (auto val) { return val < 20; };
print(coll | std::views::filter(less20)
           | std::views::transform(std::negate{}));
```

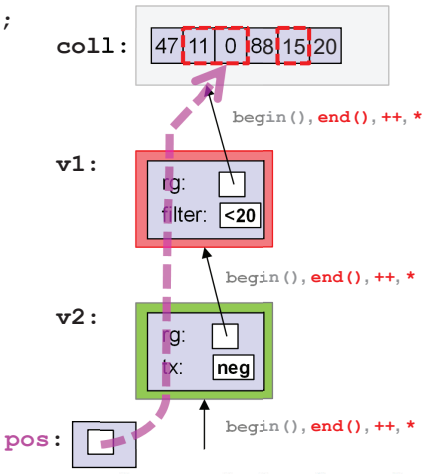
Output:

```
47 11 0 88 15 20
-11 0 -15
-11
0
```

```
auto v1 = std::views::filter(coll, less20);
auto v2 = std::views::transform(v1, std::negate{});
```

print() step by step:

```
auto pos = v2.begin();
if (pos != v2.end())
  std::cout << *pos << '\n';
  ++pos;
if (pos != v2.end())
  std::cout << *pos << '\n';
```



C++20: How Views Operate

C++20

```
std::vector<int> coll{47, 11, 0, 88, 15, 20};
print(coll);

auto less20 = [] (auto val) { return val < 20; };
print(coll | std::views::filter(less20)
           | std::views::transform(std::negate{}));
```

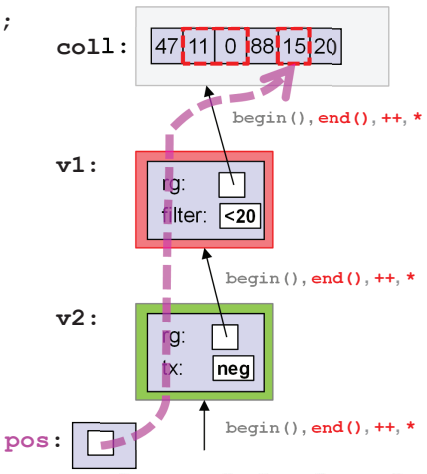
Output:

```
47 11 0 88 15 20
-11 0 -15
-11
0
-15
```

```
auto v1 = std::views::filter(coll, less20);
auto v2 = std::views::transform(v1, std::negate{});
```

print() step by step:

```
auto pos = v2.begin();
if (pos != v2.end())
  std::cout << *pos << '\n';
  ++pos;
if (pos != v2.end())
  std::cout << *pos << '\n';
  ++pos;
if (pos != v2.end())
  std::cout << *pos << '\n';
```



C++20: How Views Operate


```
std::vector<int> coll{47, 11, 0, 88, 15, 20};
print(coll);

auto less20 = [] (auto val) { return val < 20; };
print(coll | std::views::filter(less20)
           | std::views::transform(std::negate{}));
```

Output:

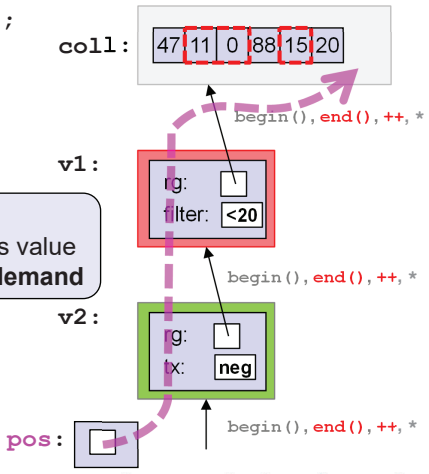
```
47 11 0 88 15 20
-11 0 -15
-11
0
-15
```

```
auto v1 = std::views::filter(coll, less20);
auto v2 = std::views::transform(v1, std::negate{});
```

print() step by step:

```
auto pos = v2.begin();
if (pos != v2.end())
  std::cout << *pos << '\n';
  ++pos;
if (pos != v2.end())
  std::cout << *pos << '\n';
  ++pos;
if (pos != v2.end())
  std::cout << *pos << '\n';
  ++pos;
if (pos != v2.end())
```

Pull model:
next element and its value are processed **on demand**



C++20

Concepts, Constraints, and Requirements in Detail

Concepts Terminology

C++20

- **Requirements**
  - Expressions to specify a restriction with `requires{...}`
    - Operations that have to be valid
    - Types that have to be defined/returned
- **Concepts**
  - Names for one or more requirements
- **Constraints**
  - Restrictions for the availability/usability of generic code
  - Specified as
    - `requires` clauses of concepts or ad-hoc requirements
    - Type constraints (concepts applied to template parameters or `auto`)
- **No code is generated**
  - Code is evaluated only to decide *whether/what* to compile

Concepts, Requirements, and Constraints

C++20

Concept (named requirements)

```
template<typename T>
concept IsPointer = requires (T p) {
                      *p;                                  // requirement for operator*
                      { p == p } -> std::convertible_to<bool>;   // operator==
                    };
```

Requirements with *requires expression*

```
template<typename T>
requires (!IsPointer<T>)
T maxValue(T a, T b)
{
  return b < a ? a : b;
}
```

Constraint with *requires clause* (needs parentheses if not only concepts, `&&`, `||`)

```
template<IsPointer T>
T maxValue(T a, T b)
{
  return *b < *a ? *a : *b;
}
```

Type Constraint (can only use concepts)

```
auto maxValue(IsPointer auto a, IsPointer auto b)
requires std::same_as<decltype(*a), decltype(*b)>
{
  return maxValue(*a, *b);
}
```

Type Constraints (can only use concepts)

Trailing requires clause

C++20: Requires Expressions

C++20

- **Requires expressions**

  requires $(parameter\text{-}list)_{opt}$ **{** *requirement-expressions* **}**

```
template<typename T1, typename T2> ...
requires(T1 x, T2 p) {
  // simple requirements (code that has to be valid):
  *p;                                  // OK to call operator * for an object of type T2
  p->value();                          // OK to call referred member function value() without args
  p == nullptr;                        // OK to compare a T2 with nullptr (not whether it is the nullptr)

  // compound requirements:
  { &x } -> std::input_or_output_iterator;
  { x == x }noexcept -> std::convertible_to<bool>;

  // type requirements:
  typename T1::value_type;             // T1 has type member value_type
  typename std::common_type_t<T1, T2>; // T1 and T2 have a common type

  // nested requirements (requirements that also have to be met (are true)):
  requires std::integral<std::remove_cvref_t<decltype(*p)>>;

  std::is_const_v<T1>;                 // "Can we check whether T1 is const?" Worthless (always valid)
  requires std::is_const_v<T1>;        // "Is T1 const?" Useful
}
```

A declaration is not allowed:
`T1 y{x};` // ERROR
`T1{x};` // OK

only concepts after ->

Different Constraints Can Create Ambiguities

C++20

```
template<typename CollT>
concept HasSize = requires (CollT c) {
                    { c.size() } -> std::convertible_to<int>;
                  };

template<typename CollT>
concept HasIndexOp = requires (CollT c) { c[0]; };

template<typename CollT>
requires HasSize<CollT>                          // has to support size()
void foo(CollT& coll) {
  std::cout << "foo() for container with size()\n";
}

template<typename CollT>
requires HasIndexOp<CollT>                       // has to support []
void foo(CollT& coll) {
  std::cout << "foo() for container []\n";
}

std::list<int> lst{0, 8, 15};
std::vector<int> vec{0, 8, 15};
foo(lst);   // OK: calls first foo()
foo(vec);   // ERROR: ambiguous
```

Output:
```
foo() for container with size()
```

Constraints with Concepts Can Subsume

C++20

```
template<typename CollT>
concept HasSize = requires (CollT c) {
                    { c.size() } -> std::convertible_to<int>;
                  };
```

```
template<typename CollT>
concept HasIndexOp = requires (CollT c) { c[0]; };
```

```
template<typename CollT>
requires HasSize<CollT>                          // has to support size()
void foo(CollT& coll) {
  std::cout << "foo() for container with size()\n";
}
```

HasSize<> && HasIndexOp<> *subsumes* HasSize<>

```
template<typename CollT>
requires HasSize<CollT> && HasIndexOp<CollT>   // has to support size() and []
void foo(CollT& coll) {
  std::cout << "foo() for container with size() and []\n";
}

std::list<int> lst{0, 8, 15};
std::vector<int> vec{0, 8, 15};
foo(lst);   // OK: calls first foo()
foo(vec);   // OK: calls second foo()
```

Output:

```
foo() for container with size()
foo() for container with size() and []
```


Constraints without Concepts Cannot Subsume

C++20

```
template<typename CollT>
constexpr bool HasSize = requires (CollT c) {
                           { c.size() } -> std::convertible_to<int>;
                         };
```

```
template<typename CollT>
constexpr bool HasIndexOp = requires (CollT c) { c[0]; };
```

```
template<typename CollT>
requires HasSize<CollT>                          // has to support size()
void foo(CollT& coll) {
  std::cout << "foo() for container with size()\n";
}
```

Does *not* subsume
(only concepts subsume)

```
template<typename CollT>
requires HasSize<CollT> && HasIndexOp<CollT>   // has to support size() and []
void foo(CollT& coll) {
  std::cout << "foo() for container with size() and []\n";
}

std::list<int> lst{0, 8, 15};
std::vector<int> vec{0, 8, 15};
foo(lst);   // OK: calls first foo()
foo(vec);   // ERROR: ambiguous
```

Output:

```
foo() for container with size()
```

Concepts Do Not Automatically Subsume

C++20

```
template<typename T>
concept GeoObject = requires(T obj) {
                      obj.draw();
                    };

template<GeoObject T>
void print(T) {
  …
}
```

```
class Circle {
 public:
  void draw() const;
  …
};

Circle c;
print(c);   // OK
```


Concepts Do Not Automatically Subsume

C++20

```
template<typename T>
concept GeoObject = requires(T obj) {
                      obj.draw();
                    };

template<GeoObject T>
void print(T) {
  …
}
```

```
template<typename T>
concept Cowboy = requires(T obj) {
                   obj.draw();
                   obj = obj;
                 };

template<Cowboy T>
void print(T) {
  …
}
```

```
class Circle {
 public:
  void draw() const;
  …
};

Circle c;
print(c);   // ambiguous
```

- `Cowboy` does ***not*** automatically subsume `GeoObject`

Concepts Do Not Automatically Subsume

C++20

```
template<typename T>
concept GeoObject = requires(T obj) {
                      obj.draw();
                    };

template<GeoObject T>
void print(T) {
  …
}
```

```
template<typename T>
concept Cowboy = requires(T obj) {
                   obj.draw();
                   obj = obj;
                 };

template<Cowboy T>
void print(T) {
  …
}
```

```
class Circle {
 public:
  void draw() const;
  …
};

Circle c;
print(c);   // ambiguous
```

- `Cowboy` does ***not*** automatically subsume `GeoObject`
- **Would** subsume if `GeoObject` is explicitly required

by the concept:

```
template<typename T>
concept Cowboy = GeoObject<T> &&
                 requires(T obj) {
                   obj = obj;
                 };
```

or by the function:

```
template<typename T>
requires Cowboy<T> && GeoObject<T>
void print(T) {
  …
}
```


Concepts with Non-Functional Requirements

C++20

- **Concepts usually have functional requirements**
  - Checkable by the compiler
- **Concepts might have non-functional requirements**
  - Either not checkable by the compiler (documentation only)
  - Or with opt-in / opt-out interface
  - **For example:**
    - **`std::ranges::sized_range<`***rg*`>`**
      - Guarantees that computing the size is cheap (constant complexity)
    - **`std::regular_invocable<`***op*`, `*args...*`>`**
      - Guarantees that *op* is stateless and does not modify passed arguments
    - **`std::increment<`***obj*`>`**
      - Guarantees that equal object are still equal after calling ++ for them (required for forward iterators/ranges)

Concept `std::ranges::sized_range` C++20

- **Concepts might have non-functional requirements**
  - Either not checkable by the compiler (documentation only)
  - Or with opt-in / opt-out interface

`std::ranges::sized_range`

- **Semantically satisfied** if it is **cheap** to yield the size
- **Formally satisfied** if for a range
  - If `std::ranges::size()` is available (opt in)
    - *rg*`.size()` is available or
    - `size(`*rg*`)` is available for the collection type or
    - Difference of `begin()` and `end()` exists
  - `std::ranges::disable_sized_range<`*Rg*`>` is not `true` (opt out)

```
class MyCont {
  …
  std::size_t size() const;   // assume this is expensive, so that this is not a sized range
};
constexpr bool std::ranges::disable_sized_range<MyCont> = true;
```


Where Concepts can be Used C++20

- **Concepts can be used for**
  - Function templates
  - Class templates
    - Including their member functions
  - Alias templates
  - Variable templates
  - Non-type template parameters
- **Concepts cannot be used for concepts**

Constraints for Member Functions

C++20

```
template<typename T>
class MyType {
  T value;
 public:
  …
  void print() const {
    std::cout << value << '\n';
  }
  bool isZero() const requires std::integral<T> || std::floating_point<T> {
    return value == 0;
  }
  bool isEmpty() const requires requires { value.empty(); } {
    return value.empty();
  }
};
```

```
MyType<double> mt1;
mt1.print();                    // OK
if (mt1.isZero()) { … }         // OK
if (mt1.isEmpty()) { … }        // ERROR

MyType<std::string> mt2;
mt2.print();                    // OK
if (mt2.isZero()) { … }         // ERROR
if (mt2.isEmpty()) { … }        // OK
```


Constraints for Member Functions

C++20

```
template<typename T>
class MyType {
 public:
  T none()      const requires std::integral<T> && (!std::same_as<T, bool>);
  double none() const requires std::floating_point<T>;
};

template<typename T>
T MyType<T>::none() const requires std::integral<T> && (!std::same_as<T, bool>) {
    return {};
}

template<typename T>
double MyType<T>::none() const requires std::floating_point<T> {
    return std::numeric_limits<double>::quiet_NaN();
}
```

Declaration and definition need the same constraints

```
MyType<int> mt1;
std::cout << mt1.none()  << '\n';   // prints 0

MyType<double> mt3;
std::cout << mt3.none()  << '\n';   // prints nan

MyType<bool> mt2;
std::cout << mt2.none()  << '\n';   // ERROR
```

Concepts for Non-Type Template Parameters

C++20

- **Concepts can be used for non-type template parameters:**

```
template<auto Val>
concept LessThan10 = Val > 0 && Val < 10;


template <typename ElemT, int MaxSize>
requires LessThan10<MaxSize>
class MyQueue {
  ElemT values[MaxSize];     // or: std::array<ElemT,MaxSize>
  ...
};


MyQueue<int, 5> mt1;              // OK

MyQueue<int, 10> mt2;             // ERROR
MyQueue<int, 0>  mt0;             // ERROR
```


Constraints for Non-Type Template Parameters

C++14

```
constexpr bool isPrime(int val)
{
  for (int i = 2; i <= val/2; ++i) {
    if (val % i == 0) {
      return false;
    }
  }
  return val > 1;   // 2 and 3 are primes, 0 and 1 not
}
```

```
template<auto Val>
requires (isPrime(Val))
class C1
{
  ...
};


C1<6> c1;   // ERROR: constraint not satisfied
C1<7> c2;   // OK
```

```
template<auto Val>
concept IsPrime = Val > 0 && isPrime(Val);

template<auto Val>
requires IsPrime<Val>
class C2
{
  ...
};

C2<6> c3;   // ERROR: concept IsPrime not satisfied
C2<7> c4;   // OK
```

C++20

Standard Concepts

Standard Concepts

- **The C++ standard library defines several concepts**
  - Basic concepts in `<concepts>`
  - Specific concepts in other parts
    - Iterator concepts in `<iterator>`
    - Range concepts in `<ranges>` (in namespace `std::ranges`)
    - Random number generator concepts in `<random>`
    - ...
  - Helper concepts to specify other concepts or components
    - Pure documentation
    - Not usable by programs

```
template<class T, class U>
concept weakly-equality-comparable-with =     // exposition only
  requires(const remove_reference_t<T>& t,
           const remove_reference_t<U>& u) {
    { t == u } -> boolean-testable;
    { t != u } -> boolean-testable;
    { u == t } -> boolean-testable;
    { u != t } -> boolean-testable;
};
```

C++20: Basic Standard Concepts

C++20

- **Basic concepts for types**
  - `std::integral`
  - `std::signed_integral`
  - `std::unsigned_integral`
  - `std::floating_point`
  - `std::movable`
  - `std::copyable`
  - `std::semiregular`
  - `std::regular`
- **Comparison concepts**
  - `std::equality_comparable`
  - `std::totally_ordered`
  - `std::three_way_comparable`
- **Auxiliary concepts**
  - `std::default_initializable`
  - `std::move_constructible`
  - `std::copy_constructible`
  - `std::destructible`
  - `std::swappable`
  - `std::weakly_incrementable`
  - `std::incrementable`

Regular Types

C++ / C++20

- **You can create, copy, assign, compare**
- **Values are equal after copy and assign**
- **Value semantics** (each object has its own value)

All copies behave the same until one is modified

For an object `x` of type `T`, you can call:

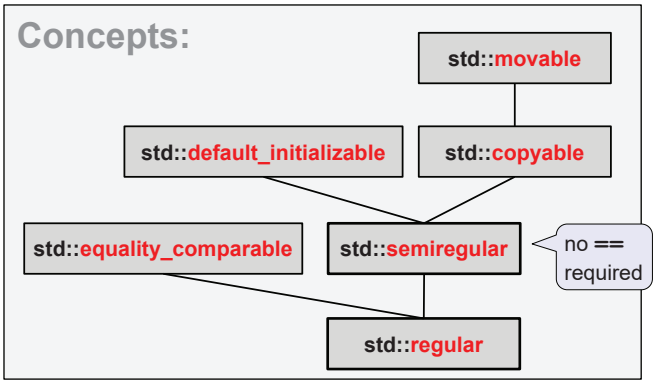
```
T y;      T z = x;
y = x;
```

Then:

```
x == y == z
```

After *modify*`(x)`:

```
x != y  and  x != z  and
y == z
```



Note:
Stepanov also requires total ordering.
The concept `std::regular` does not.

stepanovpapers.com/DeSt98.pdf
elementsofprogramming.com/

C++20: Basic Standard Concepts

C++20

- **Basic concepts for multiple objects and types**
  - `std::same_as`
  - `std::derived_from`
  - `std::convertible_to`
  - `std::constructible_from`
  - `std::assignable_from`
  - `std::swappable_with`
  - `std::common_with`
  - `std::common_reference_with`
- **Comparison concepts**
  - `std::equality_comparable_with`
  - `std::totally_ordered_with`
  - `std::three_way_comparable_with`

Concepts for Type Relations

C++20

```
template<typename T1, typename T2>
requires std::same_as<T1, T2>
void fSame(T1 from, T2 to)
{ …
}


template<typename T1, typename T2>
requires std::convertible_to<T1, T2>
void fConvert(T1 from, T2 to)
{ …
}


template<typename T1, typename T2>
requires std::constructible_from<T2, T1>
void fConstruct(T1 from, T2 to)
{ …
}
```

Note: type constraints have different order of arguments

Also as:

```
template<typename T2,
         std::convertible_to<T2> T1>
```

```
int i = 0;
fSame(42, i);          // OK
fConvert(42, i);       // OK
fConstruct(42, i);     // OK

double d = 0;
fSame(42, d);          // ERROR
fConvert(42, d);       // OK
fConstruct(42, d);     // OK

std::vector<int> v;
fSame(42, v);          // ERROR
fConvert(42, v);       // ERROR
fConstruct(42, v);     // OK

std::string s;
fSame(42, s);          // ERROR
fConvert(42, s);       // ERROR
fConstruct(42, s);     // ERROR
```

C++20: Concept `std::same_as`

C++20

- **`std::same_as<>`**
  - Standard concept to compare two types
  - Improved type trait **`std::is_same_v<>`**
    - Can subsume (even if order of types is swapped)

```
if (std::is_same<T, int>::value) {                    // since C++11
  ...
}

if constexpr (std::is_same_v<T, int>) {               // since C++17
  ...
}

if constexpr (std::same_as<T, int>) {                 // since C++20
  ...
}

if constexpr (std::same_as<char, unsigned char>) {    // since C++20
  ...
}
```


C++20: Standard Concepts for Ranges

C++20

- **Range concepts:**
  - Declared in header **`<ranges>`** in namespace **`std::ranges`**

Concept	Requires
`std::ranges::range`	range
`std::ranges::view`	range is a view (cheap copy/move)
`std::ranges::viewable_range`	convertible to view (with `std::ranges::all()`)
`std::ranges::output_range`	range to assign values of a certain type
`std::ranges::input_range`	range to read element values from
`std::ranges::forward_range`	range you can iterate over and read multiple times
`std::ranges::bidirectional_range`	range you can iterate over backwards
`std::ranges::random_access_range`	range with random access (jump back and forth)
`std::ranges::contiguous_range`	range with all elems in contiguous memory
`std::ranges::sized_range`	range with constant-time `std::ranges::size()`
`std::ranges::borrowed_range`	iterators are not tied to lifetime of the range
`std::ranges::common_range`	begin and end (sentinel) have same type

Concept `std::ranges::sized_range` C++20

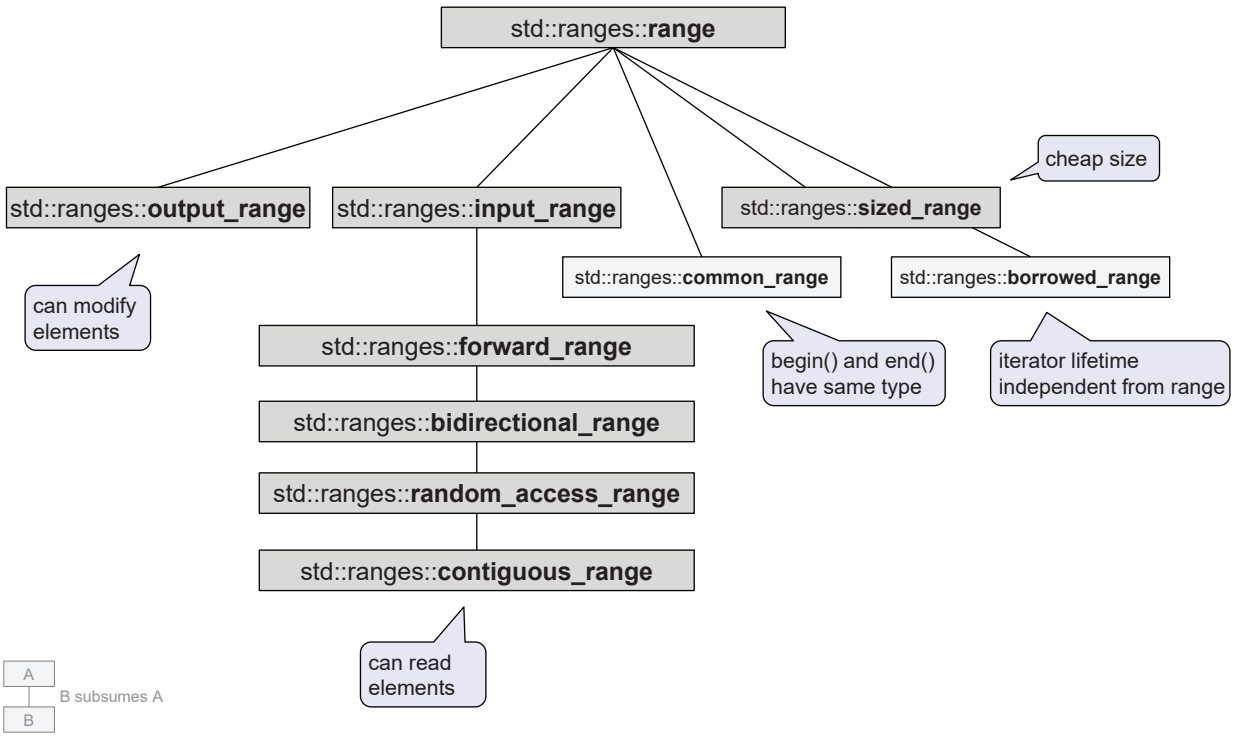
- **Concepts might have non-functional requirements**
  - Either not checkable by the compiler (documentation only)
  - Or with opt-in / opt-out interface

`std::ranges::sized_range`

- **Semantically satisfied** if it is **cheap** to yield the size
- **Formally satisfied** if for a range
  - If `std::ranges::size()` is available (opt in)
    - *rg*`.size()` is available or
    - `size(`*rg*`)` is available for the collection type or
    - Difference of `begin()` and `end()` exists
  - `std::ranges::disable_sized_range<`*Rg*`>` is not `true` (opt out)

```
class MyCont {
  …
  std::size_t size() const;   // assume this is expensive, so that this is not a sized range
};
constexpr bool std::ranges::disable_sized_range<MyCont> = true;
```


Subsumptions of Standard Range Concepts C++20



C++20: Standard Concepts for Iterators and Algorithms

C++20

- **Iterator concepts**
  - `std::input_or_output_iterator`
  - `std::output_iterator`
  - `std::input_iterator`
  - `std::forward_iterator`
  - `std::bidirectional_iterator`
  - `std::random_access_iterator`
  - `std::contiguous_iterator`
  - `std::sentinel_for`
  - `std::sized_sentinel_for`

Corresponding range concepts:
- `std::range`
- `std::output_range`
- `std::input_range`
- ...
- `std::contiguous_range`

- **Common algorithm concepts**
  - `std::permutable`
  - `std::mergeable`
  - `std::sortable`

- **Concepts for Pointer-Like Objects**
  - `std::indirectly_writable`
  - `std::indirectly_readable`
  - `std::indirectly_movable`
  - `std::indirectly_movable_storable`
  - `std::indirectly_copyable`
  - `std::indirectly_copyable_storable`
  - `std::indirectly_swappable`
  - `std::indirectly_comparable`

C++20: Standard Concepts for Callables

C++20

- **Basic concepts for callables:**

`std::invocable<`*Op*`, `*args...*`>`
Op is callable with *args*

`std::regular_invocable<`*Op*`, `*args...*`>`
Op is callable with *args* (no modifications)

`std::predicate<`*Op*`, `*args...*`>`
Op is callable with *args* and returns bool

```
template<typename Callable>
requires std::invocable<Callable, int, std::string>   // "can pass an int and a string"
void call(Callable op);

// or:
template<std::invocable<int, std::string> Callable>
void call(Callable op);

// or:
void call(std::invocable<int, std::string> auto op);
```

```
call([](int, int){...});                          // ERROR: constraint not satisfied
call([](int, const std::string&){...});           // OK
call([](long, std::optional<std::string>){...});  // OK
```

C++20: Standard Concepts for Callables

C++20

- **Basic concepts for callables:**

  `std::invocable<`*Op*`, `*args...*`>`
  Op is callable with *args*

  `std::regular_invocable<`*Op*`, `*args...*`>`
  Op is callable with *args* (no modifications)

  `std::predicate<`*Op*`, `*args...*`>`
  Op is callable with *args* and returns bool

Have additional **semantic requirements:**
- Parameters are not modified
  - No requirement that they are `const`
- Callable is stateless
  - It does not change its behavior

These constraint are only documentation and can't be checked by a compiler.

```
void call(std::invocable<int, std::string> auto op);
void callReg(std::regular_invocable<int, std::string> auto op);
```

```
void foo([](int i, const std::string& s)
{
  static int numCalls = 0;
  if (++numCalls > 100) …      // note: foo() changes behavior over time
  …
}
call(foo);          // OK
callReg(foo);       // concept not satisfied, but compiles
```


C++20: Standard Concepts for Callables

C++20

- **Basic concepts for callables:**

  `std::invocable<`*Op*`, `*args...*`>`
  Op is callable with *args*

  `std::regular_invocable<`*Op*`, `*args...*`>`
  Op is callable with *args* (no modifications)

  `std::predicate<`*Op*`, `*args...*`>`
  Op is callable with *args* and returns bool

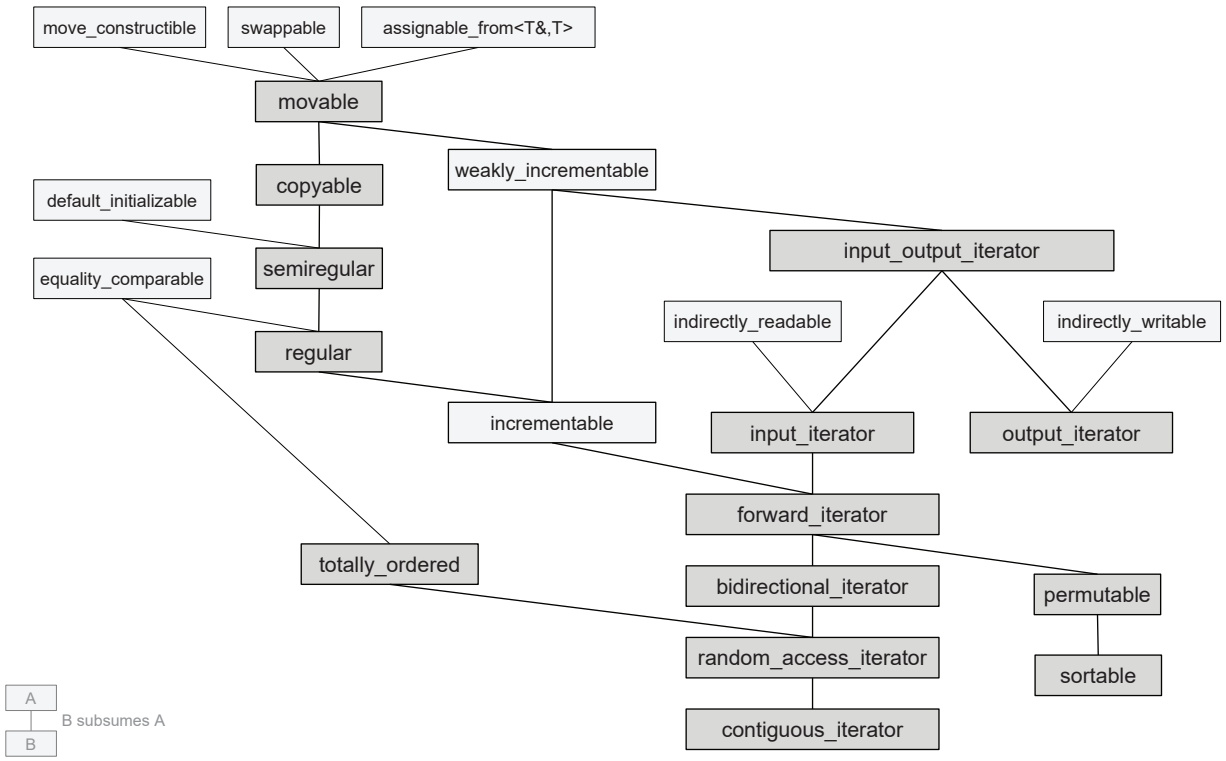
Have additional **semantic requirements:**
- Parameters are not modified
  - No requirement that they are `const`
- Callable is stateless
  - It does not change its behavior

These constraint are only documentation and can't be checked by a compiler.

- **Other concepts for callables:**

```
std::indirectly_unary_invocable
std::indirectly_regular_unary_invocable
std::indirect_unary_predicate
std::indirect_binary_predicate
std::indirect_equivalence_relation
std::indirect_strict_weak_order
std::relation
std::equivalence_relation
std::strict_weak_order
std::uniform_random_bit_generator
```

C++20: Subsumptions of Standard Concepts (Extract)



C++20

Ranges in Detail

C++20: Ranges

C++20

- **Programming with range-objects**
  - For objects that have a begin and end/sentinel
    - Including arrays
  - "*STL 2.0*" (next generation of data structures and algorithms)
- **Content:**
  - **Algorithms** for single-argument-ranges
  - **Concepts**
  - **Utilities**
    - Functions, customization point objects, types
      `std::ranges::begin()` and `std::ranges::end()`
      `std::ranges::iterator_t` and `std::ranges::value_t`
  - **Views**
    - Lightweight ranges
      - Subsets, begin-and-count, transformations
    - Can be used in pipelines with operator |

The Various Ways to Call Algorithms

C++98 / C++17 / C++20

```
std::sort(coll.begin(), coll.end());                                  // begin and end of range

std::sort(coll.begin(), coll.end(),                                   // begin and end of range
          [] (const auto& x, const auto& y) { return f(x) < f(y); }); // sort criterion
```

```
std::sort(std::execution::par,                  // with parallel execution
          coll.begin(), coll.end());

std::sort(std::execution::par,                  // with parallel execution and sort criterion
          coll.begin(), coll.end(),
          [] (const auto& x, const auto& y) { return f(x) < f(y); });
```

since C++17

```
std::ranges::sort(coll.begin(), coll.end());    // supporting concepts and sentinels (end of other type)

std::ranges::sort(coll.begin(), coll.end(),     // supporting concepts, sentinels, and sort criterion
                  [] (const auto& x, const auto& y) { return f(x) < f(y); });

std::ranges::sort(coll.begin(), coll.end(),     // supporting concepts, sentinels, sort criterion, projection
                  [] (const auto& x, const auto& y) { return x < y; },   // or std::less{}
                  [] (const auto& val) { return f(val); });

std::ranges::sort(coll);                        // for ranges supporting concepts

std::ranges::sort(coll,                         // for ranges supporting sort criterion and projection
                  [] (const auto& x, const auto& y) { return f(x) < f(y); });

std::ranges::sort(coll,                         // for ranges supporting concepts, sort criterion, projection
                  [] (const auto& x, const auto& y) { return x < y; },   // or std::less{}
                  [] (const auto& val) { return f(val); });
```

since C++20

C++20: Projections

C++20

- **Range algorithms often accept a *projection* parameter**
  - Transformation of the element before the algorithm applies

```
std::vector<int> coll{25, -42, 2, 0, 122, -5, 7};

std::ranges::sort(coll,
                  [] (auto val1, auto val2) {   // sorting criterion
                    return std::abs(val1) < std::abs(val2);
                  });

std::ranges::sort(coll,
                  std::less{},                  // way of sorting
                  [] (auto val) {               // projection for each value
                    return std::abs(val);
                  });

auto pos = std::ranges::find(coll, 25,                    // find value 25
                             [](auto x) {return x*x;});   // for squared elements
if (pos != coll.end()) {
  std::cout << "first value with square 25: " << *pos << '\n';
}
```


C++20: Standard Concepts for Ranges

C++20

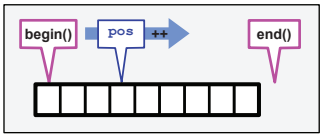
- **Range concepts:**
  - Declared in header `<ranges>` in namespace `std::ranges`

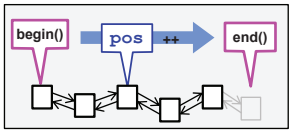
Concept	Requires
`std::ranges::range`	range
`std::ranges::view`	range is a view (cheap copy/move)
`std::ranges::viewable_range`	convertible to view (with `std::ranges::all()`)
`std::ranges::output_range`	range to assign values of a certain type
`std::ranges::input_range`	range to read element values from
`std::ranges::forward_range`	range you can iterate over and read multiple times
`std::ranges::bidirectional_range`	range you can iterate over backwards
`std::ranges::random_access_range`	range with random access (jump back and forth)
`std::ranges::contiguous_range`	range with all elems in contiguous memory
`std::ranges::sized_range`	range with constant-time `std::ranges::size()`
`std::ranges::borrowed_range`	iterators are not tied to lifetime of the range
`std::ranges::common_range`	begin and end (sentinel) have same type

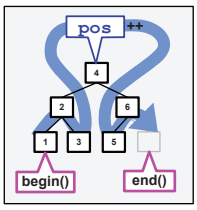
Iterators have different Categories

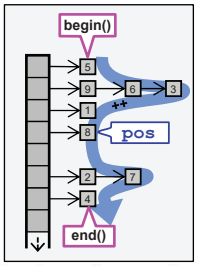
C++98

- **Random access** iterators
  - Can jump to and compare with any other position
    =, *, ++, ==, !=, --, +=, -=, <, <=, ... [], -
  - **vector<>**, **array<>**, **deque<>**, **raw arrays**, **strings**
- **Bidirectional** iterators
  - Can iterate forward and backward
    =, *, ++, ==, !=, --
  - **list<>**, **associative** containers (**set<>**, **map<>**, ...)
- **Forward** iterators
  - Can iterate forward only
    =, *, ++, ==, !=
  - **forward_list<>**, **unordered** containers (hash tables)
- **Input** iterators
  - Can only read once
  - **istream_iterator<>**





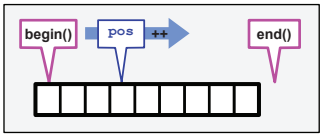


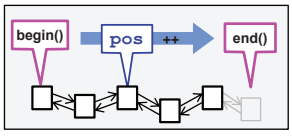


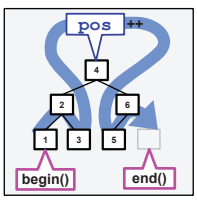
C++20: Algorithms Iterate over Ranges

C++20

```
void print(const std::ranges::input_range auto& rg) {
  for (const auto& elem : rg) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

std::vector coll{42, 9, 0, 16, 7, -1, 13};
std::ranges::sort(coll);      // sort the elements in coll
print(coll);                  // OK

std::string arr[] {"Rio", "Berlin", "LA"};
std::ranges::sort(arr);       // sort the elements in arr
print(arr);                   // OK

std::list<double> lst{1.5, 7.6, 42};
std::ranges::sort(lst);       // ERROR: no random access
print(lst);                   // OK

std::set coll{1.5, 7.6, 42.0};
std::ranges::sort(coll);      // ERROR: read-only and no random access
print(arr);                   // OK
```



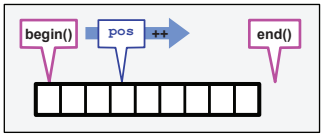




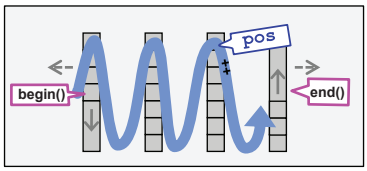
Range/Iterator Categories/Concepts (since C++20)

C++20

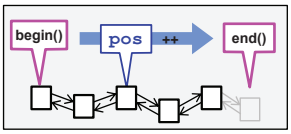
- **Contiguous range/iterator**
  - Can jump to and compare with any other position
    =, *, ++, ==, !=, --, +=, -=, <, <=, … [], -
  - Iterator may be raw pointer, range has std::ranges::`data()`
  - **vector<>**, **array<>**, **raw arrays**, **strings**

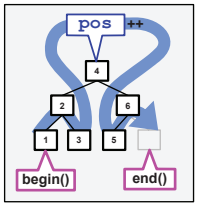


- **Random access range/iterator**
  - Can jump to and compare with any other position
    =, *, ++, ==, !=, --, +=, -=, <, <=, … [], -
  - **deque<>**

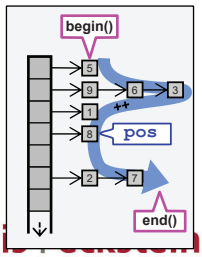


- **Bidirectional range/iterator**
  - Can iterate forward and backward
    =, *, ++, ==, !=, --
  - **list<>**, **associative** containers (**set<>**, **map<>**, ...)





- **Forward range/iterator**
  - Can iterate forward multiple times
    =, *, ++, ==, !=
  - **forward_list<>**, **unordered** containers (hash tables)



New Iterator Categories

C++20

- **Iterator categories changed with C++20**
  - New category ***contiguous***
  - **Input iterators** may no longer be copyable (only movable)
  - **Input iterators** no longer require postfix ++
  - **Iterators yielding values** (prvalues) may be more than input iterators
- **Use concepts instead of iterator traits categories**
  - Use `std::iter_value_t` etc. instead of `iterator_traits<>`

```
std::vector vec{1, 2, 3, 4};
auto pos1 = vec.begin();
decltype(pos1)::iterator_category   // type std::random_access_iterator_tag
decltype(pos1)::iterator_concept    // type std::contiguous_iterator_tag

auto v = std::views::iota(1);
auto pos2 = v.begin();
decltype(pos2)::iterator_category   // type std::input_iterator_tag
decltype(pos2)::iterator_concept    // type std::random_access_iterator_tag
```

C++20: Range Utilities

C++20

- **Types:**

```
std::ranges::iterator_t<>
std::ranges::sentinel_t<>
std::ranges::range_difference_t<>
std::ranges::range_size_t<>
std::ranges::range_value_t<>
std::ranges::range_reference_t<>
std::ranges::range_rvalue_reference_t<>
std::ranges::borrowed_iterator_t<>
std::ranges::borrowed_subrange_t<>
std::ranges::views::all_t<>
```

`std::ranges::` utilities over `std::` utilities

- **Customization Point Objects:**

```
std::ranges::begin(), std::ranges::end(),
std::ranges::cbegin(), std::ranges::cend(), ...
std::ranges::empty(), std::ranges::size(), std::ranges::ssize(),
std::ranges::data(), std::ranges::cdata()
```


C++20: Using Range Utilities

C++20

```
template<typename Coll1T, typename Coll2T>
void printMapped(const Coll1T& rg1, const Coll2T& rg2)
{
  auto pos1 = std::ranges::begin(rg1);
  auto pos2 = std::ranges::begin(rg2);
  auto end1 = std::ranges::end(rg1);
  auto end2 = std::ranges::end(rg2);
  std::ranges::range_value_t<Coll1T> def1{};
  std::ranges::range_value_t<Coll2T> def2{};

  while (pos1 != end1 || pos2 != end2) {
    std::cout << ((pos1 != end1) ? *pos1 : def1) << ':';
              << ((pos2 != end2) ? *pos2 : def2) << '\n';
    if (pos1 != end1) ++pos1;
    if (pos2 != end2) ++pos2;
  }
}
```

Customization Point Objects:
- Support all ranges and views
- Have no ADL issues
- Might have other problems solved

```
std::vector<std::string> names{
  "tic", "tac", "toe"
};
std::list vals{0, 8, 15, 47, 7};
printMapped(names, vals);

char arr[] = {'@', '?'};
printMapped(arr, names);
```

Output:

```
tic:0
tac:8
toe:15
:47
:7

@:tic
?:tac
:toe
```

C++20: Benefit of Range Utilities

C++11

```
class MyColl {
  …
 public:
  typedef … iterator;
  iterator begin() const;
  iterator end() const;
};
```

```
template<typename T>
void print1(const T& coll)
{
  for (auto pos = coll.begin();
       pos != coll.end();
       ++pos) {
    std::cout << *pos << ' ';
  }
  std::cout << '\n';
}


std::vector<std::string> vec;
print1(vec);       // OK
MyColl coll{0, 8, 15};
print1(coll);      // OK
print1("hello");   // ERROR: no member begin()
```


C++20: Benefit of Range Utilities

C++11

- **Ranges utilities**
  - From `<iterator>` in namespace `std`

```
class MyColl {
  …
 public:
  using iterator = …;
  iterator begin() const;
  iterator end() const;
};

class MyType {
  …
  const std::vector<int>& vals() const;
};
auto begin(const MyType& t) … {
  return t.vals().begin();
}
auto end(const MyType& t) … {
  return t.vals().end();
}
```

```
template<typename T>
void print2(const T& coll)
{
  for (auto pos = std::begin(coll);
       pos != std::end(coll);
       ++pos) {
    std::cout << *pos << ' ';
  }
  std::cout << '\n';
}


std::vector<std::string> vec;
print2(vec);       // OK
MyColl coll{0, 8, 15};
print2(coll);      // OK
print2("hello");   // OK
MyType x{0, 8, 15};
print2(x);         // ERROR: can't find begin()
```

C++20: Benefit of Range Utilities

C++11

- **Ranges utilities**
  - From `<iterator>` in namespace `std`

```
class MyColl {
  …
 public:
  using iterator = …;
  iterator begin() const;
  iterator end() const;
};

class MyType {
  …
  const std::vector<int>& vals() const;
};
auto begin(const MyType& t) {
  return t.vals().begin();
}
auto end(const MyType& t) {
  return t.vals().end();
}
```

```
template<typename T>
void print3(const T& coll)
{
  using std::begin;
  using std::end;
  for (auto pos = begin(coll);
       pos != end(coll);
       ++pos) {
    std::cout << *pos << ' ';
  }
  std::cout << '\n';
}

std::vector<std::string> vec;
print3(vec);       // OK
MyColl coll{0, 8, 15};
print3(coll);      // OK
print3("hello");   // OK
MyType x{0, 8, 15};
print3(x);         // OK
```


C++20: Benefit of Range Utilities

C++20

- **Ranges utilities**
  - From `<ranges>` in namespace `std::ranges`
  - Improve earlier C++ utilities
    - Supporting arrays and fixing several issues (such as ADL)

```
class MyColl {
  …
 public:
  using iterator = …;
  iterator begin() const;
  iterator end() const;
};

class MyType {
  …
  const std::vector<int>& vals() const;
};
auto begin(const MyType& t) {
  return t.vals().begin();
}
auto end(const MyType& t) {
  return t.vals().end();
}
```

```
template<typename T>
void print4(const T& coll)
{
  for (auto pos = std::ranges::begin(coll);
       pos != std::ranges::end(coll);
       ++pos) {
    std::cout << *pos << ' ';
  }
  std::cout << '\n';
}

std::vector<std::string> vec;
print4(vec);       // OK
MyColl coll{0, 8, 15};
print4(coll);      // OK
print4("hello");   // OK
MyType x{0, 8, 15};
print4(x);         // OK
```

C++20

Views
in Detail

C++20: Views

- **Views**:
  - Lightweight ranges that are **cheap** to move (and copy) and assign
    - Have reference semantics or own values
  - **Requirements:**
    - Range
    - Movable
    - Move constructor/assignment (and copy if available) have constant complexity
      - Performance does not depend on the number of elements
  - Can be infinite
  - Examples:
    - New view types like `std::ranges::drop_view`
      - With adaptors and factories like `std::views::drop()` to create them
    - Special views: `std::ranges::subrange`, `std::string_view`, `std::span`
    - Strings and containers are **not** views

Creating Views

C++20

- **Prefer helper functions to create views**
  - Adaptors: Views on existing ranges (containers, views, ...)
  - Factories: Views from iterators (and count) or that generate values
  - For example:
    Use `std::views::drop()` to create a `std::ranges::drop_view<>`
- **Use `auto` as type**
  - The exact view type depends on
    - Whether it refers to a container or a view
    - Whether it refers to a range or owns a range
    - Special cases handled by the adaptors

`std::views` is an alias for `std::ranges::views`

```
std::vector<int> c;
auto v1 = c | std::views::drop(2);          // drop_view that refers to c
auto v2 = v1 | std::views::drop(2);         // drop_view that refers to v1
auto v3 = getColl() | std::views::drop(2);  // drop_view owning the moved return value
```

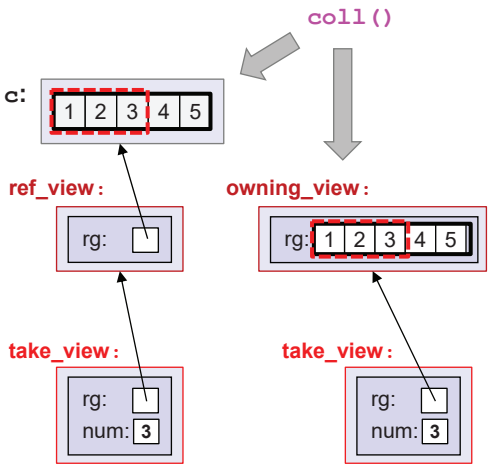

Exact Type of Views to Containers

C++20

- **std::ranges::ref_view for named objects (lvalues)**
- **std::ranges::owning_view for temporaries (rvalues)**
  - Added later as fix to C++20 with http://wg21.link/**P2415**

```
std::vector<int> coll()
{
  return {1, 2, 3, 4, 5};
}
```

```
std::vector<int> c = coll();
auto v1 = c | std::views::take(3);
          // take_view<ref_view<vector<int>>>

auto v2 = coll() | std::views::take(3);
          // take_view<owning_view<vector<int>>>
```



Views and their Adaptors/Factories with later C++20 Fixes P2415 and P2432

C++20

View Type	Adaptor	Effect
std::ranges::**take_view**	std::views::**take**(*rg*, *num*)	first *num* elems
std::ranges::**take_while_view**	std::views::**take_while**(*rg*, *pred*)	leading elems that match *pred*
std::ranges::**drop_view**	std::views::**drop**(*rg*, *num*)	skip first *num* elems
std::ranges::**drop_while_view**	std::views::**drop_while**(*rg*, *pred*)	skip leading elems that match *pred*
std::ranges::**filter_view**	std::views::**filter**(*rg*, *pred*)	only elems that match *pred*
std::ranges::**transform_view**	std::views::**transform**(*rg*, *func*)	all elements transformed by *func*
std::ranges::**elements_view**	std::views::**elements**<*idx*>	*idx*-th attribute of each all elems
std::ranges::**keys_view**	std::views::**keys**(*rg*)	1st attribute of each elems (elements_view)
std::ranges::**values_view**	std::views::**values**(*rg*)	2nd attribute of each elems (elements_view)
std::ranges::**reverse_view**	std::views::**reverse**(*rg*)	all elems in revers order
std::ranges::**split_view**	std::views::**split**(*rg*, *sep*)	all elems split into multiple sub-ranges
std::ranges::**lazy_split_view**	std::views::**lazy_split**(*rg*, *sep*)	all elems split into multiple sub-ranges
std::ranges::**join_view**	std::views::**join**(*rg*)	all elems of range of ranges in one range
std::ranges::**common_view**	std::views::**common**(*rg*)	ensure begin and end iterator have same type
std::ranges::**ref_view**	std::views::**all**(*rg*), std::views::**common**(*rg*)	all elems of a range/view
std::ranges::**owning_view**	std::views::**all**(*rg*)	view with moved ownership of elems of *rg*
	Factory	
std::ranges::**subrange**	std::views::**counted**(*beg*, *sz*)	(sub-)sequence of elems
std::**span**	std::views::**counted**(*beg*, *sz*)	(sub-)sequence of elems in contiguous memory
std::**string_view**		read-only character sequence (since C++17)
std::ranges::**basic_istream_view**	std::views::**istream**<T>(*strm*)	all values of type T read from an input stream
std::ranges::**wistream_view**		
std::ranges::**iota_view**	std::views::**iota**(*val*), std::views::**iota**(*val*, *endval*)	sequence of integral values (with optional end)
std::ranges::**single_view**	std::views::**single**(*val*)	range with a single value (with reference semantics)
std::ranges::**empty_view**	std::views::**empty**<T>	empty sequence of type T

Views and their Adaptors/Factories

C++23

View Type	Adaptor	Effect
std::ranges::**zip_view**	std::views::**zip**(*rg1*, *rg2*, ...)	pairs of elements of multiple ranges
std::ranges::**zip_transform_view**	std::views::**zip_transform**(*func*, *rg1*, *rg2*, ...,)	combine elements of multiple ranges
std::ranges::**reverse_view**	std::views::**reverse**(*rg*)	all elems in revers order
std::ranges::**split_view**	std::views::**split**(*rg*, *sep*)	all elems split into multiple sub-ranges
std::ranges::**lazy_split_view**	std::views::**lazy_split**(*rg*, *sep*)	all elems split into multiple sub-ranges
std::ranges::**join_view**	std::views::**join**(*rg*)	all elems of range of ranges in one range
std::ranges::**common_view**	std::views::**common**(*rg*)	ensure begin and end iterator have same type
std::ranges::**drop_view**	std::views::**drop**(*rg*, *num*)	skip first *num* elems
std::ranges::**drop_while_view**	std::views::**drop_while**(*rg*, *pred*)	skip leading elems that match *pred*
std::ranges::**filter_view**	std::views::**filter**(*rg*, *pred*)	only elems that match *pred*
std::ranges::**transform_view**	std::views::**transform**(*rg*, *func*)	all elements transformed by *func*
std::ranges::**elements_view**	std::views::**elements**<*idx*>	*idx*-th attribute of each all elems
std::**string_view**		read-only character sequence (since C++17)
std::ranges::**basic_istream_view**	std::views::**istream**<T>(*strm*)	all values of type T read from an input stream
std::ranges::**wistream_view**		
std::ranges::**iota_view**	std::views::**iota**(*val*), std::views::**iota**(*val*, *endval*)	sequence of integral values (with optional end)
std::ranges::**single_view**	std::views::**single**(*val*)	range with a single value (with reference semantics)
as_rvalue_view as_const_view repeat_view join_with_view	std::views::**empty**<T>	empty sequence of type T

C++23: New Standard Views

C++23

- as_rvalue_view
- as_const_view
- repeat_view
- join_with_view
- zip_view
- zip_transform_view
- adjacent_view
- adjacent_transform_view
- chunk_view
- chunk_by_view
- slide_view
- stride_view
- cartesian_product_view

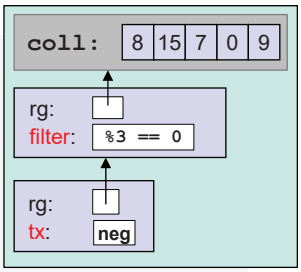
C++20

Performance of Filter Views

Pipelines: Lazy Evaluation

C++20

```
namespace vws = std::views;

std::vector<int> coll{ 8, 15, 7, 0, 9 };

// define a view:
auto vColl
  = coll
    | vws::filter([] (int i) {
              std::cout << " filter: " << i << '\n';
              return i % 3 == 0;
            })
    | vws::transform([] (int i) {
              std::cout << "  trans: " << i << '\n';
              return -i;
            });

// and use it:
std::cout << "coll | filter | tx:\n";

for (int val : vColl) {
  std::cout << "          => " << val << '\n';
}
```



Output:

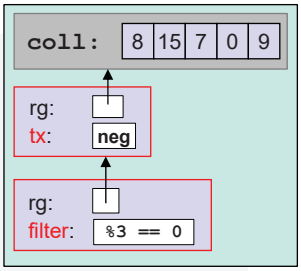
```
coll | filter | tx:
 filter: 8
 filter: 15
  trans: 15
         => -15
 filter: 7
 filter: 0
  trans: 0
         => 0
 filter: 9
  trans: 9
         => -9
```

Pull model **(lazy evaluation):** next element and its value are processed **on demand**

Pipelines: Price of Filter Views

C++20

```
namespace vws = std::views;

std::vector<int> coll{ 8, 15, 7, 0, 9 };

// define a view:
auto vColl
  = coll
    | vws::transform([] (int i) {
              std::cout << " trans: " << i << '\n';
              return -i;
            })
    | vws::filter([] (int i) {
              std::cout << "  filter: " << i << '\n';
              return i % 3 == 0;
            });

// and use it:
std::cout << "coll | tx | filter:\n";

for (int val : vColl) {
  std::cout << "          => " << val << '\n';
}
```



Output:

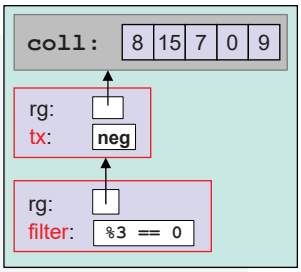
```
coll | tx | filter:
 trans: 8
  filter: -8
 trans: 15
  filter: -15
 trans: 15
         => -15
 trans: 7
  filter: -7
 trans: 0
  filter: 0
 trans: 0
         => 0
 trans: 9
  filter: -9
 trans: 9
         => -9
```

Pipelines: Price of Filter Views

C++20

```
namespace vws = std::views;

std::vector<int> coll{ 8, 15, 7, 0, 9 };

// define a view:
auto vColl
  = coll
    | vws::transform([] (int i) {
                std::cout << " trans: " << i << '\n';
                return -i;
              })
    | vws::filter([] (int i) {
                std::cout << "  filter: " << i << '\n';
                return i % 3 == 0;
              });

// and use it:
std::cout << "coll | tx | filter:\n";

for (int val : vColl) {
  std::cout << "          => " << val << '\n';
}
```

- **Filters** need the value to check, but pass its position (iterator)
- **Callers** need the value **again** to process



Output:

```
coll | tx | filter:
 trans: 8
  filter: -8
 trans: 15
  filter: -15
 trans: 15
         => -15
 trans: 7
  filter: -7
 trans: 0
  filter: 0
 trans: 0
         => 0
 trans: 9
  filter: -9
 trans: 9
         => -9
```

- Use **filters early**
- Avoid expensive stuff before using them

Performance of Views

C++20

- **Good compilers optimize a lot**
  - but not everything (yet)

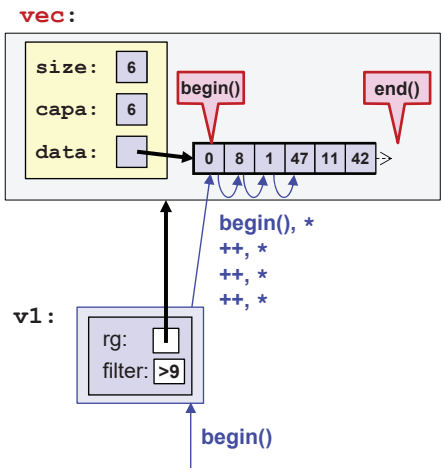
```
for (int v : rg | std::views::filter([] (int i) { return i % 3 == 0; })
                | std::views::transform([] (int i) { return -i; }) ) {
  process(v);
}
```

```
for (auto pos = rg.begin(); pos != rg.end(); ++pos) {
  if (*pos % 3 == 0) {
    process(-*pos);
  }
}
```

```
for (int v : rg | std::views::transform([] (int i) { return -i; })
                | std::views::filter([] (int i) { return i % 3 == 0; }) ) {
  process(v);
}
```

```
for (auto pos = rg.begin(); pos != rg.end(); ++pos) {
  if (-*pos % 3 == 0) {
    process(-*pos);
  }
}
```

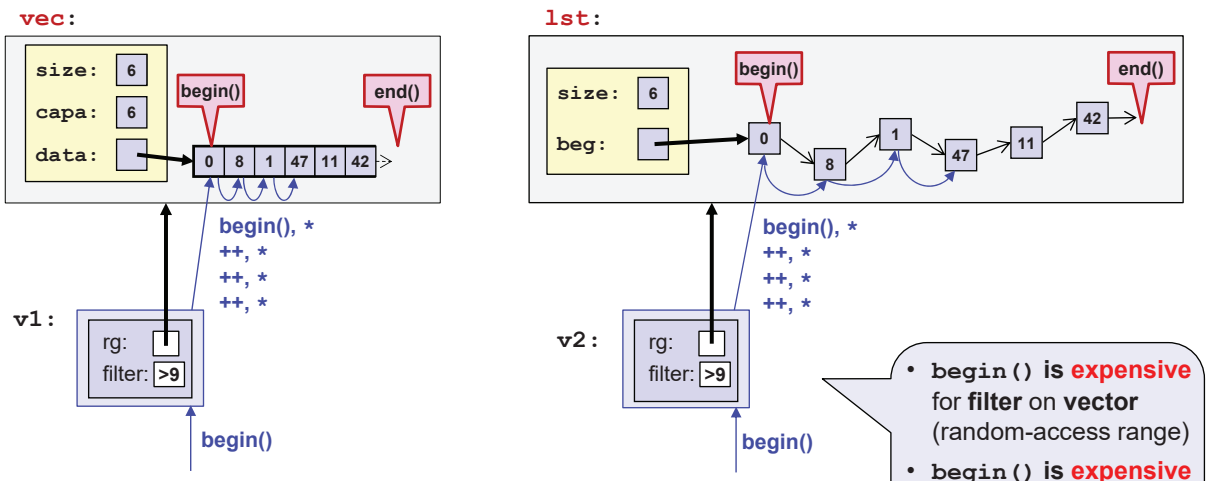
How Expensive is `begin()` ?




```
std::vector<int> vec{0, 8, 1, 47, 11, 42};
auto v1 = vec | std::views::filter(gt9);
auto pos = v1.begin();
```


How Expensive is `begin()` ?

C++20



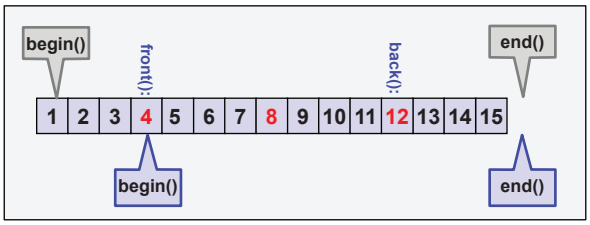
- `begin()` is **expensive** for **filter** on **vector** (random-access range)
- `begin()` is **expensive** for **filter** on **list** (forward range)

```
std::vector<int> vec{0, 8, 1, 47, 11, 42};
auto v1 = vec | std::views::filter(gt9);
auto pos = v1.begin();

std::list<int> lst{0, 8, 1, 47, 11, 42};
auto v2 = lst | std::views::filter(gt9);
auto pos = v2.begin();
```

Member Functions of Views

C++20

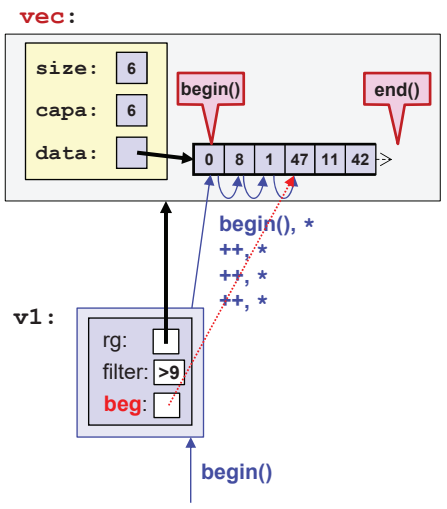
- **Views do not provide expensive member functions**




```
auto vFlt = coll | std::views::filter(multipleOf4);

vFlt.begin()      // slow: pred for leading elements (begin() and forward until first true)
vFlt.end()        // fast: just return end() of underlying range

vFlt.empty()      // slow: pred for leading elements until first true (aka begin() == end())
vFlt.size()       // slow: pred for all elements
vFlt.front()      // slow: pred for leading elements (aka *begin())
vFlt.back()       // slow: pred for all trailing elements (end() and backwards until first true)
vFlt[idx]         // slow: pred for all elements until idx times true
```


Caching `begin()`

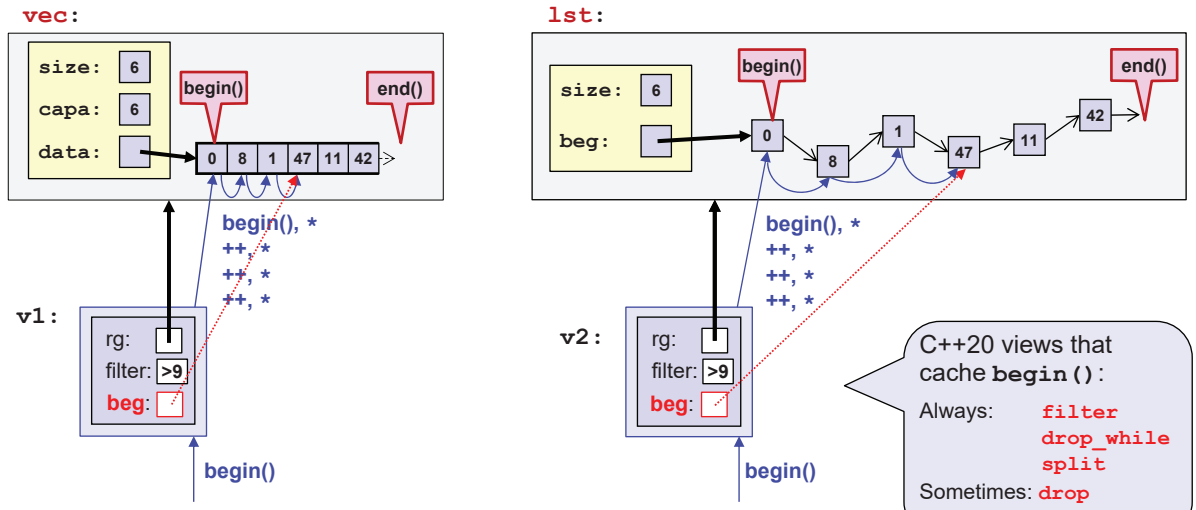
C++20




```
std::vector<int> vec{0, 8, 1, 47, 11, 42};
auto v1 = vec | std::views::filter(gt9);
auto pos = v1.begin();
```

Caching `begin()`

C++20



C++20 views that cache `begin()`:

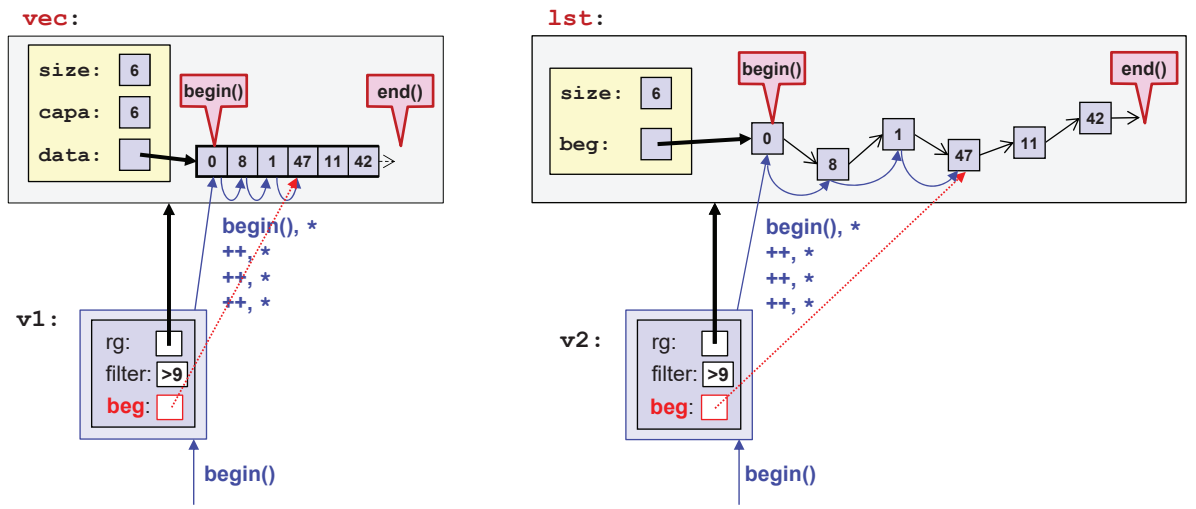
Always: `filter` `drop_while` `split`

Sometimes: `drop` `reverse`

```
std::vector<int> vec{0, 8, 1, 47, 11, 42};
auto v1 = vec | std::views::filter(gt9);
auto pos = v1.begin();

std::list<int> lst{0, 8, 1, 47, 11, 42};
auto v2 = lst | std::views::filter(gt9);
auto pos = v2.begin();
```

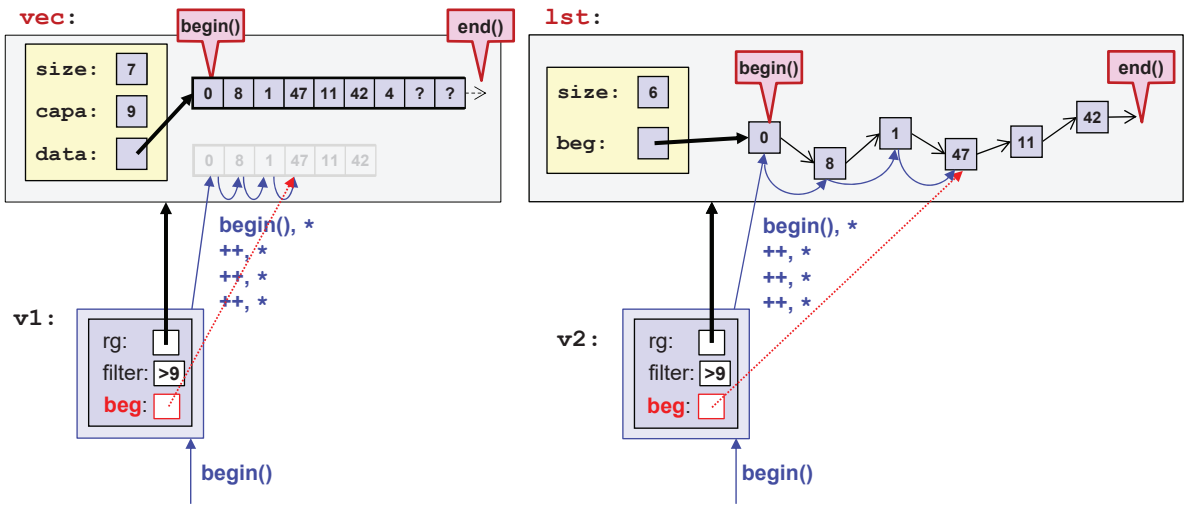

Ways of Caching `begin()`

C++20



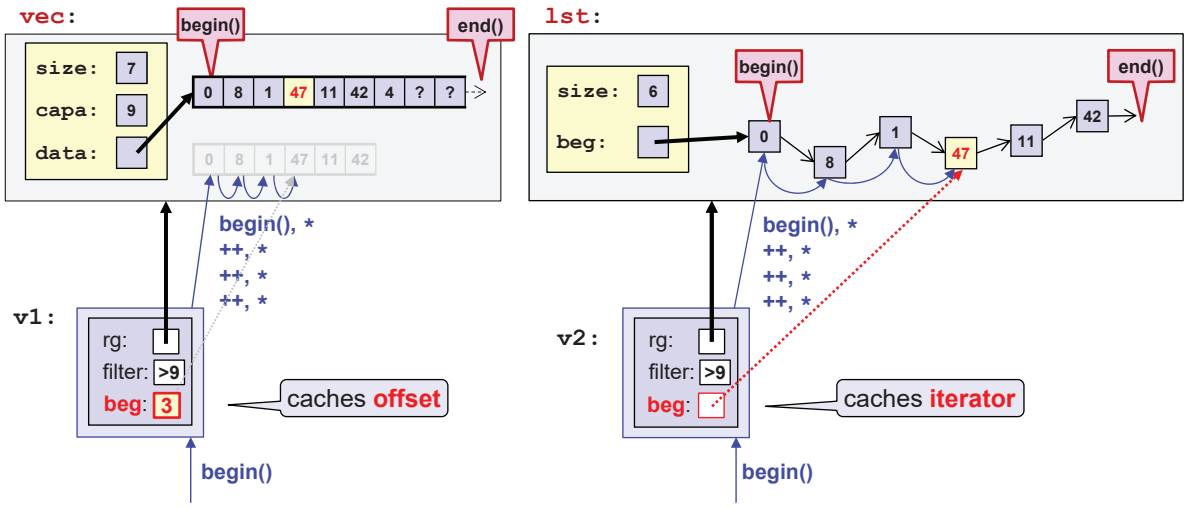

```
std::vector<int> vec{0, 8, 1, 47, 11, 42};
auto v1 = vec | std::views::filter(gt9);
auto pos = v1.begin();
vec.push_back(4);
std::list<int> lst{0, 8, 1, 47, 11, 42};
auto v2 = lst | std::views::filter(gt9);
auto pos = v2.begin();
```

Ways of Caching begin() C++20




```
std::vector<int> vec{0, 8, 1, 47, 11, 42};
auto v1 = vec | std::views::filter(gt9);
auto pos = v1.begin();
vec.push_back(4);
std::list<int> lst{0, 8, 1, 47, 11, 42};
auto v2 = lst | std::views::filter(gt9);
auto pos = v2.begin();
```


Ways of Caching begin() C++20



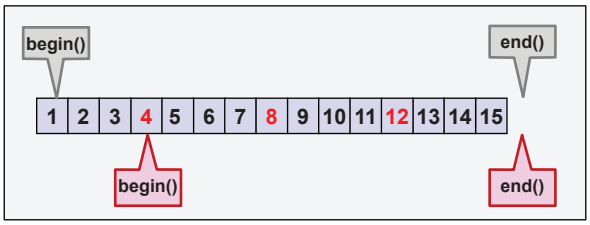

```
std::vector<int> vec{0, 8, 1, 47, 11, 42};
auto v1 = vec | std::views::filter(gt9);
auto pos = v1.begin();
vec.push_back(4);
std::list<int> lst{0, 8, 1, 47, 11, 42};
auto v2 = lst | std::views::filter(gt9);
auto pos = v2.begin();
```

Member Functions of Views

C++20

- **Views do not provide expensive member functions**

cached:




```
auto vFlt = coll | std::views::filter(multipleOf4);
```

`vFlt.begin()`	// slow once: *pred* for leading elements (`begin()` and forward until first true)
`vFlt.end()`	// **fast**: just return `end()` of underlying range
`vFlt.empty()`	// slow once: *pred* for all elements until first true (aka `begin() == end()`)
~~`vFlt.size()`~~	// **slow**: *pred* for all elements
`vFlt.front()`	// slow once: *pred* for leading elements (aka `*begin()`)
`vFlt.back()`	// slow: *pred* for all trailing elements (`end()` and backwards until first true)
~~`vFlt[idx]`~~	// **slow**: *pred* for all elements until *idx* times true

Complexity of Member Functions of Views

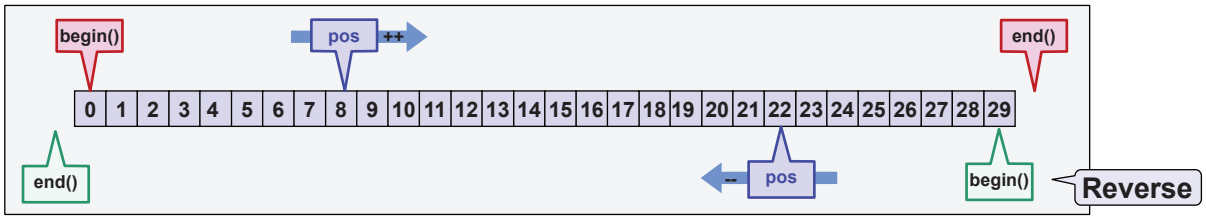
C++20

	1st `begin()`	2nd `begin()`	`size()`	1st `empty()`	2nd `empty()`
`std::vector vec`	constant	constant	constant	constant	constant
`std::list lst`	constant	constant	constant	constant	constant
`vec | filter(...)`	linear	constant	--	linear	constant
`lst | filter(...)`	linear	constant	--	linear	constant

"First time linear, then constant" is called "***amortized constant***" in the C++ standard

C++20 Standard:

[range.range]:
Given an expression `t` such that `decltype((t))` is `T&`,
`T` **models** concept std::**range only if**
...
(3.2) — both `ranges::begin(t)` and `ranges::end(t)` are **amortized constant time** and non-modifying,
...



- **Typical manual iteration:**

```
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
  sum = *pos;
}
```

Multiple calls of **end()**
- might become multiple calls of **begin()**

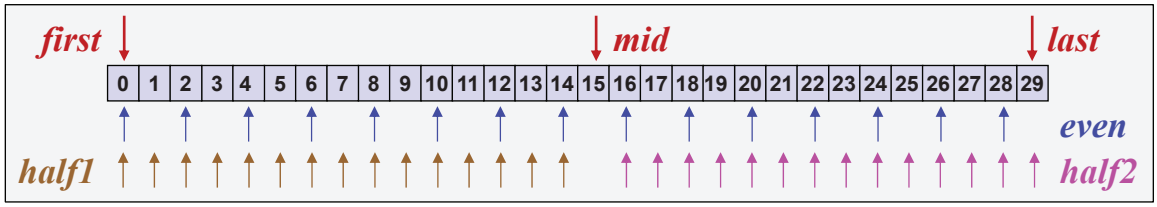
  – Note: *reverse* maps `end()` to `begin()`

- **Iterating with range-based `for` loop:**

```
for (const auto& elem : coll) {
  sum += elem;
}
```

```
auto pos = coll.begin();
auto end = coll.end();
for ( ; pos != end; ++pos) {
  sum = *pos;
}
```


Filter Performance with Caching

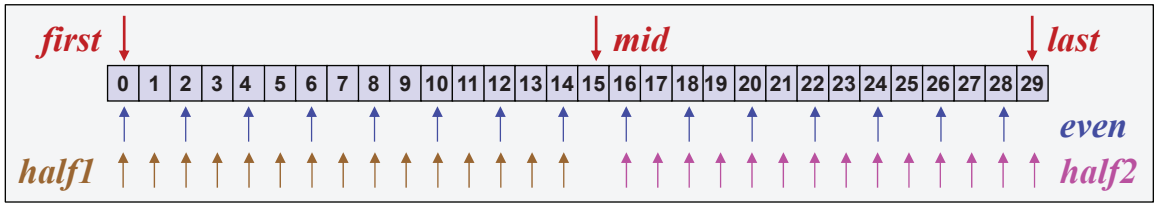


range-based:

	1st	2nd:	1st	2nd:
filter(first)	385.86ms	405.93ms	377.26ms	379.53ms
filter(mid)	386.33ms	194.31ms	373.32ms	181.67ms
filter(last)	381.34ms	0.00ms	375.41ms	0.00ms
filter(even)	450.30ms	460.55ms	503.00ms	503.36ms
filter(half1)	674.71ms	680.82ms	622.90ms	633.79ms
filter(half2)	696.19ms	513.58ms	642.38ms	434.40ms
filter(first) | reverse	649.19ms	608.32ms	605.81ms	604.44ms
filter(mid) | reverse	498.37ms	309.53ms	486.75ms	286.74ms
filter(last) | reverse	407.89ms	0.00ms	379.56ms	0.00ms
filter(even) | reverse	606.94ms	616.81ms	715.44ms	674.08ms
filter(half1) | reverse	779.04ms	682.39ms	895.47ms	903.84ms
filter(half2) | reverse	580.28ms	392.38ms	787.38ms	600.90ms
filter(first) | reverse | filter(all)	610.66ms	600.54ms	578.05ms	560.42ms
filter(mid) | reverse | filter(all)	492.59ms	301.47ms	485.43ms	311.64ms
filter(last) | reverse | filter(all)	385.82ms	0.00ms	390.62ms	0.00ms
filter(even) | reverse | filter(all)	769.80ms	718.78ms	3962.03ms	3937.80ms
filter(half1) | reverse | filter(all)	1033.77ms	856.34ms	4132.64ms	3977.84ms
filter(half2) | reverse | filter(all)	747.44ms	562.06ms	3903.25ms	3715.96ms
	gcc -O2		**cl /Ox**	

Filter Performance with Caching

C++20



pos!=end():

```
filter(first)
filter(mid)
filter(last)
filter(even)
filter(half1)
filter(half2)

filter(first) | reverse
filter(mid) | reverse
filter(last) | reverse
filter(even) | reverse
filter(half1) | reverse
filter(half2) | reverse

filter(first) | reverse | filter(all)
filter(mid) | reverse | filter(all)
filter(last) | reverse | filter(all)
filter(even) | reverse | filter(all)
filter(half1) | reverse | filter(all)
filter(half2) | reverse | filter(all)
```

gcc -O2

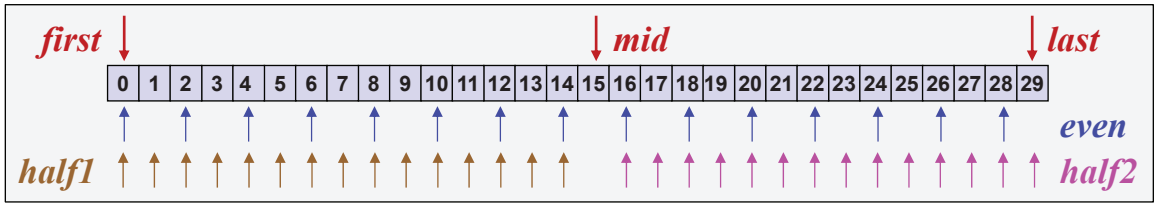
```
   1st          2nd:
------------------------
 401.05ms     378.88ms
 378.15ms     195.94ms
 375.61ms       0.00ms
 481.38ms     531.65ms
 693.86ms     662.76ms
 649.95ms     466.35ms

 609.90ms     589.35ms
 512.04ms     322.72ms
 391.74ms       0.00ms
 778.66ms     792.39ms
 989.19ms     952.68ms
 864.26ms     656.88ms

 620.25ms     598.94ms
 486.63ms     299.74ms
 379.18ms       0.00ms
 796.03ms     780.65ms
1090.91ms     917.37ms
 920.55ms     641.97ms
```

cl /Ox

```
   1st          2nd:
------------------------
 418.88ms     404.31ms
 421.95ms     209.15ms
 433.11ms       0.00ms
 541.36ms     564.08ms
 660.56ms     657.66ms
 677.98ms     455.52ms

 655.51ms     672.01ms
 544.11ms     306.08ms
 387.05ms       0.00ms
1099.95ms    1111.42ms
1406.75ms    1448.75ms
1266.03ms    1070.92ms

 624.22ms     643.03ms
 555.47ms     332.61ms
 439.22ms       0.00ms
3909.79ms    3934.14ms
4303.06ms    4177.14ms
4026.26ms    3797.89ms
```


Filter Performance without Caching

C++20



range-based:

```
filter(first)
filter(mid)
filter(last)
filter(even)
filter(half1)
filter(half2)

filter(first) | reverse
filter(mid) | reverse
filter(last) | reverse
filter(even) | reverse
filter(half1) | reverse
filter(half2) | reverse

filter(first) | reverse | filter(all)
filter(mid) | reverse | filter(all)
filter(last) | reverse | filter(all)
filter(even) | reverse | filter(all)
filter(half1) | reverse | filter(all)
filter(half2) | reverse | filter(all)
```

gcc -O2

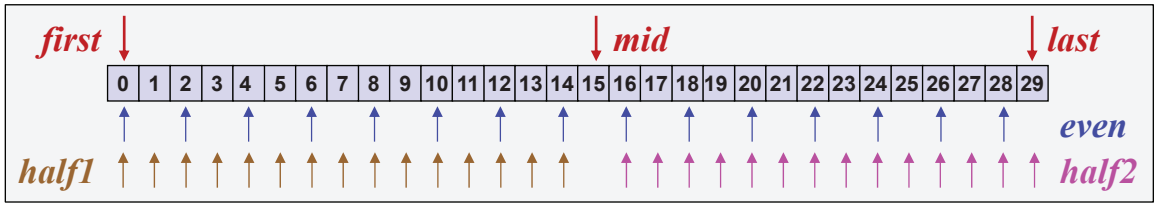
```
   1st          2nd:
------------------------
 410.50ms     435.74ms
 394.27ms     480.11ms
 517.82ms     509.74ms
 488.65ms     459.19ms
 481.02ms     478.87ms
 507.69ms     501.52ms

 637.77ms     648.16ms
 491.52ms     492.13ms
 374.06ms     396.88ms
 722.54ms     729.59ms
 695.50ms     689.06ms
 578.05ms     600.78ms

 618.02ms     643.70ms
 500.27ms     514.75ms
 379.07ms     378.91ms
 683.96ms     656.78ms
1038.81ms    1084.17ms
  timeout      timeout
```

cl /Ox

```
   1st          2nd:
------------------------
 288.43ms     285.99ms
 297.46ms     140.89ms
 286.07ms       0.00ms
 367.48ms     355.06ms
 445.46ms     438.16ms
 458.02ms     306.53ms

 466.71ms     464.53ms
 370.97ms     232.32ms
 290.43ms       0.00ms
 454.68ms     476.29ms
 640.42ms     644.66ms
 571.24ms     420.65ms

 452.73ms     454.20ms
 372.12ms     229.64ms
 286.93ms       0.00ms
2612.16ms    2597.78ms
2809.43ms    2670.83ms
2582.43ms    2459.20ms
```

Filter Performance without Caching

C++20



range-based:

1000 times less elements

```
filter(first)
filter(mid)
filter(last)
filter(even)
filter(half1)
filter(half2)

filter(first) | reverse
filter(mid) | reverse
filter(last) | reverse
filter(even) | reverse
filter(half1) | reverse
filter(half2) | reverse

filter(first) | reverse | filter(all)
filter(mid) | reverse | filter(all)
filter(last) | reverse | filter(all)
filter(even) | reverse | filter(all)
filter(half1) | reverse | filter(all)
filter(half2) | reverse | filter(all)
```

gcc -O2

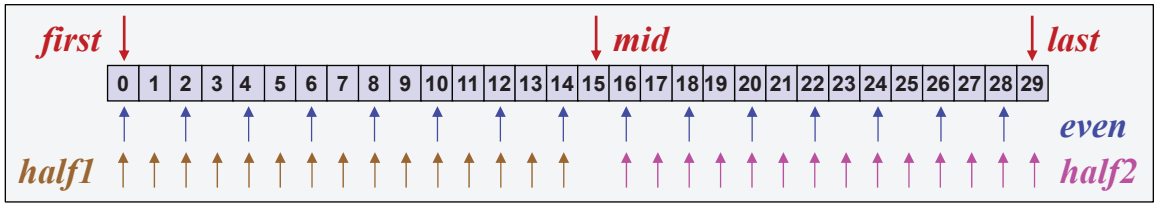
1st	2nd:
0.32ms	0.32ms
0.31ms	0.31ms
0.31ms	0.31ms
0.39ms	0.39ms
0.39ms	0.39ms
0.39ms	0.39ms
0.38ms	0.38ms
0.35ms	0.35ms
0.31ms	0.31ms
0.55ms	0.55ms
0.58ms	0.58ms
0.50ms	0.75ms
0.39ms	0.38ms
0.35ms	0.35ms
0.31ms	0.31ms
0.52ms	0.52ms
0.84ms	0.86ms
41188.76ms	41114.62ms

cl /Ox

1st	2nd:
288.43ms	285.99ms
297.46ms	140.89ms
286.07ms	0.00ms
367.48ms	355.06ms
445.46ms	438.16ms
458.02ms	306.53ms
466.71ms	464.53ms
370.97ms	232.32ms
290.43ms	0.00ms
454.68ms	476.29ms
640.42ms	644.66ms
571.24ms	420.65ms
452.73ms	454.20ms
372.12ms	229.64ms
286.93ms	0.00ms
2612.16ms	2597.78ms
2809.43ms	2670.83ms
2582.43ms	2459.20ms

Filter Performance without Caching

C++20



pos!=end():

```
filter(first)
filter(mid)
filter(last)
filter(even)
filter(half1)
filter(half2)

filter(first) | reverse
filter(mid) | reverse
filter(last) | reverse
filter(even) | reverse
filter(half1) | reverse
filter(half2) | reverse

filter(first) | reverse | filter(all)
filter(mid) | reverse | filter(all)
filter(last) | reverse | filter(all)
filter(even) | reverse | filter(all)
filter(half1) | reverse | filter(all)
filter(half) | reverse
```

gcc -O2

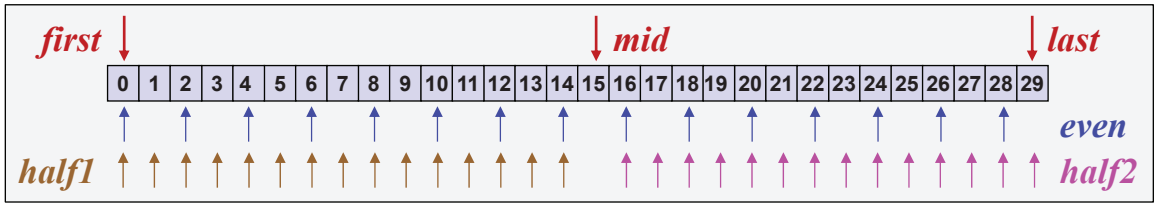
1st	2nd:
321.34ms	339.50ms
312.10ms	312.41ms
312.76ms	311.30ms
363.67ms	430.83ms
369.38ms	359.71ms
362.40ms	380.86ms
513.50ms	507.49ms
600.77ms	606.11ms
682.21ms	627.78ms
626.97ms	624.59ms
925.27ms	900.21ms
timeout	timeout
516.83ms	514.09ms
587.86ms	579.39ms
623.74ms	625.88ms
563.37ms	549.78ms
993.20ms	996.87ms
timeout	timeout

cl /Ox

1st	2nd:
377.73ms	417.56ms
375.68ms	199.24ms
390.90ms	0.00ms
521.64ms	503.84ms
633.26ms	612.58ms
624.08ms	440.49ms
564.66ms	566.09ms
486.85ms	289.45ms
391.39ms	0.00ms
1042.00ms	1093.73ms
1328.03ms	1368.97ms
1281.94ms	1056.53ms
581.35ms	577.78ms
485.63ms	299.63ms
384.29ms	0.00ms
3892.57ms	3864.58ms
4125.22ms	3969.29ms
3991.79ms	3796.46ms

Filter Performance without Caching

C++20



pos!=end():

1000 times less elements

```
filter(first)                          0.42ms       0.39ms      |  377.73ms     417.56ms
filter(mid)                            0.36ms       0.34ms      |  375.68ms     199.24ms
filter(last)                           0.33ms       0.32ms      |  390.90ms       0.00ms
filter(even)                           0.40ms       0.40ms      |  521.64ms     503.84ms
filter(half1)                          0.39ms       0.39ms      |  633.26ms     612.58ms
filter(half2)                          0.39ms       0.39ms      |  624.08ms     440.49ms

filter(first) | reverse                 0.52ms       0.52ms      |  564.66ms     566.09ms
filter(mid) | reverse                   0.54ms       0.54ms      |  486.85ms     289.45ms
filter(last) | reverse                  0.63ms       0.80ms      |  391.39ms       0.00ms
filter(even) | reverse                  1.05ms       0.86ms      | 1042.00ms    1093.73ms
filter(half1) | reverse                 1.05ms       1.03ms      | 1328.03ms    1368.97ms
filter(half2) | reverse                82155.16ms   82142.53ms  | 1281.94ms    1056.53ms

filter(first) | reverse | filter(all)   0.63ms       0.39ms      |  581.35ms     577.78ms
filter(mid) | reverse | filter(all)     0.51ms       0.51ms      |  485.63ms     299.63ms
filter(last) | reverse | filter(all)    0.63ms       0.63ms      |  384.29ms       0.00ms
filter(even) | reverse | filter(all)    0.63ms       0.63ms      | 3892.57ms    3864.58ms
filter(half1) | reverse | filter(all)   3.27ms       1.97ms      | 4125.22ms    3969.29ms
filter(half2) | reverse | filter(all)   83041.35ms   82521.06ms  | 3991.79ms    3796.46ms
```

1st	2nd:
gcc -O2	

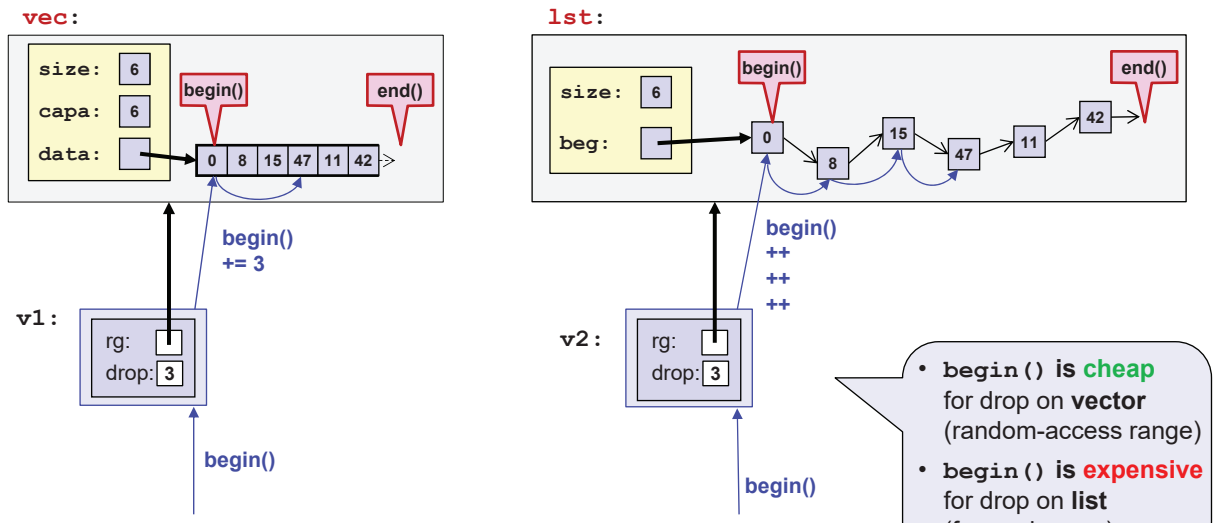
1st	2nd:
cl /Ox	

C++20

Performance of Other Views

How Expensive is `begin()` ?

C++20

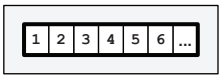


- `begin()` is **cheap** for drop on **vector** (random-access range)
- `begin()` is **expensive** for drop on **list** (forward range)

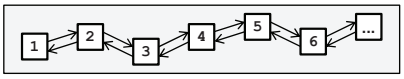
```
std::vector<int> vec{0, 8, 15, 47, 11, 42};
auto v1 = vec | std::views::drop(3);
auto pos = v1.begin();

std::list<int> lst{0, 8, 15, 47, 11, 42};
auto v2 = lst | std::views::drop(3);
auto pos = v2.begin();
```


Member Functions of Views

C++20




```
auto vVec = vec | std::views::drop(n);
vVec.begin()       // fast: vec.begin() + n
vVec.empty()       // fast: vec.size() <= n
vVec.size()        // fast: n >= vec.size() ? 0 : vec.size() - n
vVec[idx]          // fast: vec[idx + n]
```




```
auto vLst = lst | std::views::drop(n);
vLst.begin()       // slow: lst.begin() and n times ++
vLst.empty()       // fast:  lst.size() <= n
vLst.size()        // fast:  n >= lst.size() ? 0 : lst.size() - n
vLst[idx]          // slow: lst.begin() and n + idx times ++
```

```
auto vFlt = coll | std::views::filter(pred);
vFlt.begin()       // slow: ++, ==, end(), pred for all elements until first true
vFlt.empty()       // slow: ++, ==, end(), pred for all elements until first true
vFlt.size()        // slow: ++, ==, end(), pred for all elements
vFlt[idx]          // slow: ++, ==, end(), pred for all elements until idx  times true
```

Views don't provide expensive member functions

```
auto vFD = coll | std::views::filter(pred) | std::views::drop(n);
vFD.begin()        // slow
vFD.empty()        // slow
vFD.size()         // slow
vFD[idx]           // slow
```

View APIs depend on underlying ranges/views

Processing Containers and Views

C++20

```
void print(const auto& coll)
{
  std::cout << coll.size() << " elems\n";
}
```

Output:

```
7 elems
7 elems
3 elems
3 elems
4 elems
4 elems
ERROR
ERROR
```

```
std::vector<int> vec{0, 8, 15, 47, 11, 42, 1};
std::list<int> lst{0, 8, 15, 47, 11, 42, 1};

print(vec);
print(lst);
print(vec | std::views::take(3));                    // OK
print(lst | std::views::take(3));                    // OK
print(vec | std::views::drop(3));                    // OK
print(lst | std::views::drop(3));                    // OK
print(vec | std::views::filter(isEven));             // ERROR
print(vec | std::views::filter(isEven) | std::views::drop(3));   // ERROR
```


Processing Containers and Views

C++20

```
void print(const auto& coll)
{
  std::cout << coll.size() << " elems: ";

  std::cout << coll.front() << " ...\n";
}
```

Output:

```
7 elems: 0 ...
7 elems: 0 ...
3 elems: 0 ...
3 elems: 0 ...
4 elems: 47 ...
ERROR
ERROR
ERROR
```

```
std::vector<int> vec{0, 8, 15, 47, 11, 42, 1};
std::list<int> lst{0, 8, 15, 47, 11, 42, 1};

print(vec);
print(lst);
print(vec | std::views::take(3));                    // OK
print(lst | std::views::take(3));                    // OK
print(vec | std::views::drop(3));                    // OK
print(lst | std::views::drop(3));                    // ERROR
print(vec | std::views::filter(isEven));             // ERROR
print(vec | std::views::filter(isEven) | std::views::drop(3));   // ERROR
```

Processing Containers and Views

C++20

```
void print(const auto& coll)
{
  std::cout << coll.size() << " elems: ";

  std::cout << coll.front() << " ... "
            << coll.back() << '\n';
}
```

Output:

```
7 elems: 0 ... 1
7 elems: 0 ... 42
3 elems: 0 ... 15
ERROR
4 elems: 47 ... 1
ERROR
ERROR
ERROR
```

```
std::vector<int> vec{0, 8, 15, 47, 11, 42, 1};
std::list<int> lst{0, 8, 15, 47, 11, 42, 1};

print(vec);
print(lst);
print(vec | std::views::take(3));                    // OK
print(lst | std::views::take(3));                    // ERROR
print(vec | std::views::drop(3));                    // OK
print(lst | std::views::drop(3));                    // ERROR
print(vec | std::views::filter(isEven));             // ERROR
print(vec | std::views::filter(isEven) | std::views::drop(3));   // ERROR
```


Basic API of Views

C++20/C++23

- The basic API of views depends on constraints

Member function	Meaning	Constraints
`begin()`	Begin iterator	
`end()`	Sentinel (end iterator)	
`cbegin()`	Begin iterator to `const` elem	since C++23
`cend()`	Sentinel to `const` elem	since C++23
`empty()`	is empty?	forward range
conversion to `bool`	has elements?	`std::ranges::empty()` valid
`size()`	number of elements	sized range (with cheap size)
`front()`	first element	forward range
`back()`	last element	bidirectional and common range
`[`*idx*`]`	*idx*-th element	random-access range
`data()`	raw pointer to elements	contiguous range

- No `cdata()`

Base Class for Views `std::ranges::view_interface` C++20

```
namespace std::ranges {
 template<typename D>          // for each derived view type D
 requires is_class_v<D> && same_as<D, remove_cv_t<D>>
 class view_interface {
  public:
   constexpr bool empty() requires sized_range<D> || forward_range<D>;

   constexpr explicit operator bool() requires requires { ranges::empty(asD(*this)); };

   constexpr auto size() requires forward_range<D> &&
                                  sized_sentinel_for<sentinel_t<D>, iterator_t<D>>;

   constexpr decltype(auto) front() requires forward_range<D>;
   constexpr decltype(auto) back() requires bidirectional_range<D> && common_range<D>;

   template<random_access_range R = D>
    constexpr decltype(auto) operator[](range_difference_t<R> n);

   constexpr auto data() requires contiguous_iterator<iterator_t<D>> {
     return to_address(ranges::begin(ranges::empty(asD(*this))));
   }
   ...     // same as const member functions for const D
 };
}
```

Derived view type `D` provides `begin()`, `end()`, and specific member functions

P2278 adds for C++23: `cbegin()` and `cend()`

C++20

Consequences for Using Views

Processing Containers and Views

C++20

```
void print(const auto& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

Output:

```
0 8 1 47 11 42 2
0 8 1
0 -8 -1 -47 -11 -42 -2
ERROR
47 11 42
```

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
print(vec);

print(vec | std::views::take(3));                          // OK

print(vec | std::views::transform(std::negate{}));         // OK

auto gt9 = [] (auto val) { return val > 9; };
print(vec | std::views::filter(gt9));                      // Compile-time ERROR

for (int v : vec | std::views::filter(gt9)) {              // OK
  std::cout << v << ' ';
}
```


Processing Containers and Views

C++20

```
void print(const auto& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

Output:

```
0 8 1 47 11 42 2
0 8 1
0 -8 -1 -47 -11 -42 -2
ERROR
47 11 42
```

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
print(vec);

print(vec | std::views::take(3));                          // OK

print(vec | std::views::transform(std::negate{}));         // OK

auto gt9 = [] (auto val) { return val > 9; };
print(vec | std::views::filter(gt9));                      // Compile-time ERROR

for (int v : vec | std::views::filter(gt9)) {              // OK
  std::cout << v << ' ';
}
```

Processing Containers and Views

C++20

```
void print(const auto& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

Output:

```
0 8 1 47 11 42 2
0 8 1
0 -8 -1 -47 -11 -42 -2
ERROR
47 11 42
```

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
print(vec);
print(vec | std::views::take(3));                    // OK
print(vec | std::views::transform(std::negate{}));   // OK
auto gt9 = [] (auto val) { return val > 9; };
print(vec | std::views::filter(gt9));                // Compile-time ERROR
auto fv{vec | std::views::filter(gt9)};
print(std::ranges::subrange{fv});                    // OK
```

Does ***not*** work in one statement

`subrange` is a view that stores/caches `begin()` on initialization

Processing Containers and Views by Universal Reference

C++20

```
void print(auto&& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

Universal (or *forwarding*) ***reference***
- Can universally refer to every expression (even temporaries/rvalues) without making it `const`

```
0 8 1
0 -8 -1 -47 -11 -42 -2
47 11 42
```

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
print(vec);
print(vec | std::views::take(3));                    // OK
print(vec | std::views::transform(std::negate{}));   // OK
auto gt9 = [] (auto val) { return val > 9; };
print(vec | std::views::filter(gt9));                // OK
```

Concurrent Read Access to Views

C++20

Use `const auto&` in this case

```
void process(auto&& coll) {
  // separate thread to read access the elements:
  std::jthread t1{[&] {
                    for (const auto& elem : coll) {   // DANGER: calls begin()
                      ...
                    }
                  }};

  ...       // so far the code is safe, but...

  if (!coll.empty()) {   // OOPS: concurrent call of begin()
    ...
  }
}
```

Concurrent read access is undefined behavior
- Concurrent iteration with `begin()` or `empty()` or `front()` ...

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
process(vec);                                          // OK
process(vec | std::views::take(3));                    // OK
process(vec | std::views::transform(std::negate{}));   // OK
auto gt9 = [] (auto val) { return val > 9; };
process(vec | std::views::filter(gt9));                // Runtime ERROR
```


Forcing Caching

C++20

```
void process(auto&& coll) {
  (void)coll.empty();     // force caching (DO NOT REMOVE)
  // separate thread to read access the elements:
  std::jthread t1{[&] {
                    for (const auto& elem : coll) {   // OK: begin() was called
                      ...
                    }
                  }};

  ...       // so far the code is safe, but...

  if (!coll.empty()) {   // OK: begin() was called
    ...
  }
}
```

`(void)` to avoid that compilers warn about a useless call

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
process(vec);                                          // OK
process(vec | std::views::take(3));                    // OK
process(vec | std::views::transform(std::negate{}));   // OK
auto gt9 = [] (auto val) { return val > 9; };
process(vec | std::views::filter(gt9));                // works but UB
```

Processing Containers and Views by Value

C++20

```
void print(auto coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

Output:

```
0 8 1 47 11 42 2
0 8 1
0 -8 -1 -47 -11 -42 -2
47 11 42
```

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
print(vec);                                          // expensive copy
print(vec | std::views::take(3));                    // OK
print(vec | std::views::transform(std::negate{}));   // OK
auto gt9 = [] (auto val) { return val > 9; };
print(vec | std::views::filter(gt9));                // OK
```


Processing Containers and Views by Value

C++20

```
void print(std::ranges::view auto coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

Output:

```
ERROR
0 8 1
0 -8 -1 -47 -11 -42 -2
47 11 42
```

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
print(vec);                                          // ERROR
print(vec | std::views::take(3));                    // OK
print(vec | std::views::transform(std::negate{}));   // OK
auto gt9 = [] (auto val) { return val > 9; };
print(vec | std::views::filter(gt9));                // OK
```

Processing Containers and Views by Value

C++20

```
void print(std::ranges::view auto coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

Output:

```
0 8 1 47 11 42 2
0 8 1
0 -8 -1 -47 -11 -42 -2
47 11 42
```

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
print(std::views::all(vec));                          // cheap
print(vec | std::views::take(3));                     // OK
print(vec | std::views::transform(std::negate{}));    // OK

auto gt9 = [] (auto val) { return val > 9; };
print(vec | std::views::filter(gt9));                 // OK
```


Overloading for Containers and Views

C++20

```
void print(std::ranges::view auto coll) { // for views only
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

template<typename T>
void print(const T& coll)                 // not for views
requires (!std::ranges::view<T>)
{
  print(std::views::all(coll));
}
```

Necessary to avoid ambiguities when views are passed

```
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
print(vec);                               // OK, calls 2nd print()
print(vec | std::views::filter(gt9));     // OK, calls 1st print()

print(getColl());                         // OK, calls 2nd print()
print(getColl() | std::views::filter(gt9)); // OK, calls 1st print()
```

C++20

Issues with Views

Processing Containers and Views


```
void print(const auto& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
std::list<int> lst{0, 8, 1, 47, 11, 42, 2};

print(vec);
print(lst);
print(vec | std::views::take(3));          // print first three elements
print(lst | std::views::take(3));          // print first three elements
print(vec | std::views::drop(3));          // print fourth to last element
print(lst | std::views::drop(3));          // Compile-time ERROR
for (int v : lst | std::views::drop(3)) {  // OK: print fourth to last element
   std::cout << v << ' ';
}
```

Output:

```
0 8 1 47 11 42 2
0 8 1 47 11 42 2
0 8 1
0 8 1
47 11 42 2
ERROR
47 11 42 2
```

Processing Containers and Views

C++20

```
void print(const auto& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
std::list<int> lst{0, 8, 1, 47, 11, 42, 2};

print(vec);
print(lst);
print(vec | std::views::take(3));          // print first three elements
print(lst | std::views::take(3));          // print first three elements
print(vec | std::views::drop(3));          // print fourth to last element
print(lst | std::views::drop(3));          // Compile-time ERROR
for (int v : lst | std::views::drop(3)) {  // OK: print fourth to last element
   std::cout << v << ' ';
}
```

Output:

```
0 8 1 47 11 42 2
0 8 1 47 11 42 2
0 8 1
0 8 1
47 11 42 2
ERROR
47 11 42 2
```


Processing Containers and Views

C++20

```
void print(const auto& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
std::list<int> lst{0, 8, 1, 47, 11, 42, 2};

print(vec);
print(lst);
print(vec | std::views::take(3));          // print first three elements
print(lst | std::views::take(3));          // print first three elements
print(vec | std::views::drop(3));          // print fourth to last element
print(lst | std::views::drop(3));          // Compile-time ERROR
auto greater9 = [] (auto val) { return val > 9; };
print(vec | std::views::filter(greater9)); // Compile-time ERROR
```

Output:

```
0 8 1 47 11 42 2
0 8 1 47 11 42 2
0 8 1
0 8 1
47 11 42 2
ERROR
ERROR
```

Processing Containers and Views

C++20

```
void print(const auto& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
std::list<int> lst{0, 8, 1, 47, 11, 42, 2};

print(vec);
print(lst);
print(vec | std::views::take(3));          // print first three elements
print(lst | std::views::take(3));          // print first three elements
print(vec | std::views::drop(3));          // print fourth to last element
print(lst | std::views::drop(3));          // Compile-time ERROR
auto dv{lst | std::views::drop(3)}
print(std::ranges::subrange{dv});          // OK
```

Output:

```
0 8 1 47 11 42 2
0 8 1 47 11 42 2
0 8 1
0 8 1
47 11 42 2
ERROR
47 11 42 2
```

Does ***not*** work in one statement

`subrange` is a view that stores/caches `begin()` on initialization

Passing Containers and Views by Universal Reference

C++20

```
void print(auto&& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

Universal (or *forwarding*) ***reference***
- Can universally refer to every expression (even temporaries/rvalues) without making it `const`

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
std::list<int> lst{0, 8, 1, 47, 11, 42, 2};

print(vec);
print(lst);
print(vec | std::views::take(3));          // print first three elements
print(lst | std::views::take(3));          // print first three elements
print(vec | std::views::drop(3));          // print fourth to last element
print(lst | std::views::drop(3));          // OK: print fourth to last element
auto greater9 = [] (auto val) { return val > 9; };
print(vec | std::views::filter(greater9)); // OK
```

Output:

```
0 8 1 47 11 42 2
0 8 1 47 11 42 2
0 8 1
0 8 1
47 11 42 2
47 11 42 2
47 11 42
```

Concurrent Read Access to Views

C++20

Use `const auto&` in this case

```
void process(auto&& coll) {
  // separate thread to read access the elements:
  std::jthread t1{[&] {
                    for (const auto& elem : coll) {   // DANGER: calls begin()
                      ...
                    }
                  }};

  ...       // so far the code is safe, but...

  if (!coll.empty()) {   // OOPS: concurrent call of begin()
    ...
  }
}
```

Concurrent read access is undefined behavior
- Concurrent iteration with `begin()` or `empty()` or `front()` ...

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
process(vec);                                          // OK
process(vec | std::views::take(3));                    // OK
process(vec | std::views::transform(std::negate{}));   // OK
auto gt9 = [] (auto val) { return val > 9; };
process(vec | std::views::filter(gt9));                // Runtime ERROR
```


Forcing Caching

C++20

```
void process(auto&& coll) {
  (void)coll.empty();     // force cashing (DO NOT REMOVE)
  // separate thread to read access the elements:
  std::jthread t1{[&] {
                    for (const auto& elem : coll) {   // OK: begin() was called
                      ...
                    }
                  }};

  ...       // so far the code is safe, but...

  if (!coll.empty()) {   // OK: begin() was called
    ...
  }
}
```

`(void)` to avoid that compilers warn about a useless call

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
process(vec);                                          // OK
process(vec | std::views::take(3));                    // OK
process(vec | std::views::transform(std::negate{}));   // OK
auto gt9 = [] (auto val) { return val > 9; };
process(vec | std::views::filter(gt9));                // OK
```

Passing Containers and Views by Value

C++20

```
void print(auto coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
std::list<int> lst{0, 8, 1, 47, 11, 42, 2};
```

```
print(vec);                                  // expensive
print(lst);                                  // expensive
print(vec | std::views::take(3));            // print first three elements
print(lst | std::views::take(3));            // print first three elements
print(vec | std::views::drop(3));            // print fourth to last element
print(lst | std::views::drop(3));            // OK: print fourth to last element
for (int v : lst | std::views::drop(3)) {    // OK: print fourth to last element
   std::cout << v << ' ';
}
```

Output:

```
0 8 1 47 11 42 2
0 8 1 47 11 42 2
0 8 1
0 8 1
47 11 42 2
47 11 42 2
47 11 42 2
```


Accepting Views by Value Only

C++20

```
void print(std::ranges::view auto coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
std::list<int> lst{0, 8, 1, 47, 11, 42, 2};
```

```
print(vec);                                  // ERROR
print(lst);                                  // ERROR
print(vec | std::views::take(3));            // print first three elements
print(lst | std::views::take(3));            // print first three elements
print(vec | std::views::drop(3));            // print fourth to last element
print(lst | std::views::drop(3));            // OK: print fourth to last element
for (int v : lst | std::views::drop(3)) {    // OK: print fourth to last element
   std::cout << v << ' ';
}
```

Output:

```
0 8 1 47 11 42 2
0 8 1 47 11 42 2
0 8 1
0 8 1
47 11 42 2
47 11 42 2
47 11 42 2
```

Accepting Views by Value Only

C++20

```
void print(std::ranges::view auto coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
std::list<int> lst{0, 8, 1, 47, 11, 42, 2};
```

```
print(std::views::all(vec));                    // cheap
print(std::views::all(lst));                    // cheap
print(vec | std::views::take(3));               // print first three elements
print(lst | std::views::take(3));               // print first three elements
print(vec | std::views::drop(3));               // print fourth to last element
print(lst | std::views::drop(3));               // OK: print fourth to last element
for (int v : lst | std::views::drop(3)) {       // OK: print fourth to last element
   std::cout << v << ' ';
}
```

Output:

```
0 8 1 47 11 42 2
0 8 1 47 11 42 2
0 8 1
0 8 1
47 11 42 2
47 11 42 2
47 11 42 2
```


Overloading for Containers and Views

C++20

```
void print(std::ranges::view auto coll) { // for views only
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

template<typename T>
void print(const T& coll)                 // not for views
requires (!std::ranges::view<T>)
{
  print(std::views::all(coll));
}
```

Necessary to avoid ambiguities when views are passed

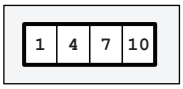
```
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
print(vec);                               // OK, calls 2nd print()
print(vec | std::views::take(3));         // OK, calls 1st print()

print(getColl());                         // OK, calls 2nd print()
print(getColl() | std::views::take(3));   // OK, calls 1st print()
```

C++20

Modifying View Elements

Modifying Filtered Elements


```
std::vector<int> coll{1, 4, 7, 10};
print(coll);

auto isEven = [] (auto&& i) { return i % 2 == 0; };
auto collEven = coll | std::views::filter(isEven);

// add 2 to even elements:
for (int& i : collEven) {
  i += 2;
}
print(coll);

// add 2 to even elements:
for (int& i : collEven) {
  i += 2;
}
print(coll);
```



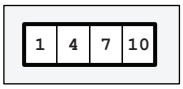
Output:

```
1 4 7 10
1 6 7 12
1 8 7 14
```

Modifying Filtered Elements

C++20

```
std::vector<int> coll{1, 4, 7, 10};
print(coll);

auto isEven = [] (auto&& i) { return i % 2 == 0; };
auto collEven = coll | std::views::filter(isEven);

// increment even elements:
for (int& i : collEven) {
  i += 1;          // Runtime Error: UB: predicate broken
}
print(coll);

// increment even elements:
for (int& i : collEven) {
  i += 1;          // Runtime Error: UB: predicate broken
}
print(coll);
```



Output:

```
1 4 7 10
1 5 7 11
1 6 7 11
```


Modifying Filtered Elements

C++20

- **Key use case of a filter:**
  - Fix broken elements

has undefined behavior:

> **[range.filter.iterator]:**
> Modification of the element a filter_view::*iterator* denotes is permitted, but results in undefined behavior if the resulting value does not satisfy the filter predicate.

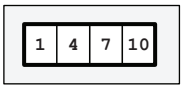
```
// as a shaman:
for (auto& m : monsters | std::views::filter(isDead)) {
  m.resurrect();   // undefined behavior: because no longer dead
  m.burn();        // OK (because it is still dead)
}
```

Thanks to Patrice Roy for this example

Modifying Filtered Elements

C++20

```
std::vector<int> coll{1, 4, 7, 10};
print(coll);

auto isEven = [] (auto&& i) { return i % 2 == 0; };
```




```
// increment even elements:
for (int& i : coll | std::views::filter(isEven)) {
  i += 1;          // UB: but works
}
print(coll);
```

Output:

```
1 4 7 10
1 5 7 11
1 5 7 11
```

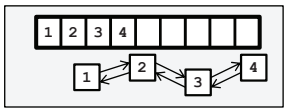
```
// increment even elements:
for (int& i : coll | std::views::filter(isEven)) {
  i += 1;          // UB: but works
}
print(coll);
```

Use (and reuse) views ad hoc

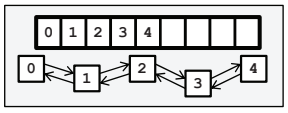
Filter Views on Modified Ranges

C++20

```
std::vector vec{1, 2, 3, 4};
vec.reserve(9);
std::list lst{1, 2, 3, 4};
auto vVec = vec | std::views::filter(even);
auto vLst = lst | std::views::filter(even);
```

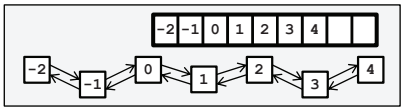



```
// insert new elements at the front:
vec.insert(vec.begin(), 0);
lst.insert(lst.begin(), 0);
print(vVec, vLst);
```




```
0 2 4
0 2 4
```

```
vec.insert(vec.begin(), {-2, -1});
lst.insert(lst.begin(), {-2, -1});
print(vVec, vLst);
```

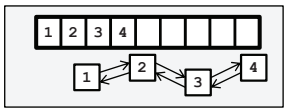



```
-2 0 2 4
0 2 4
```

Filter Views on Modified Ranges

C++20

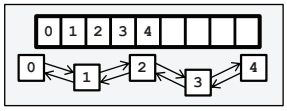
```
std::vector vec{1, 2, 3, 4};
vec.reserve(9);
std::list lst{1, 2, 3, 4};
auto vVec = vec | std::views::filter(even);
auto vLst = lst | std::views::filter(even);
print(vVec, vLst);
```



```
2 4
2 4
```

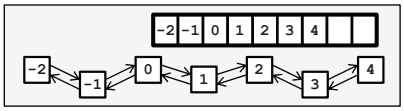
A `print()` or other read **changes state**

```
// insert new elements at the front:
vec.insert(vec.begin(), 0);
lst.insert(lst.begin(), 0);
print(vVec, vLst);
```



```
1 2 4
2 4
```

```
vec.insert(vec.begin(), {-2, -1});
lst.insert(lst.begin(), {-2, -1});
print(vVec, vLst);
```

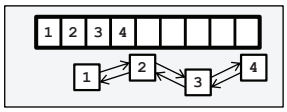


```
-1 0 2 4
2 4
```


Copying Filter Views

C++20

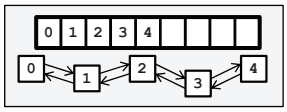
```
std::vector vec{1, 2, 3, 4};
vec.reserve(9);
std::list lst{1, 2, 3, 4};
auto vVec = vec | std::views::filter(even);
auto vLst = lst | std::views::filter(even);
print(vVec, vLst);
```



```
2 4
2 4
```

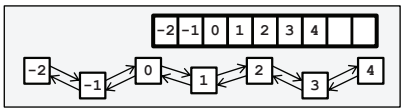
A `print()` or other read **changes state**

```
// insert new elements at the front:
vec.insert(vec.begin(), 0);
lst.insert(lst.begin(), 0);
print(vVec, vLst);
```



```
1 2 4
2 4
```

```
vec.insert(vec.begin(), {-2, -1});
lst.insert(lst.begin(), {-2, -1});
print(vVec, vLst);
```



```
-1 0 2 4
2 4
```

```
// copy the views:
auto vVecB = vVec;
auto vLstB = vLst;
print(vVecB, vLstB);
```

Copying the view ***might*** uncache

```
-1 0 2 4
-2 0 2 4
```

Filter Views and `empty()`

C++20

```
std::vector vec{0, 8, 4, 16, 42};
auto vOdd = vec |
            std::views::filter(odd);
vec.insert(vec.begin(), 1);
print(vOdd);       // OK: 1
```

```
std::list lst{1, 0, 8, 4, 16, 42};
auto vOdd = lst |
            std::views::filter(odd);
lst.push_front(1);
print(vOdd);       // OK: 1
```

```
std::vector vec{0, 8, 4, 16, 42};
auto vOdd = vec |
            std::views::filter(odd);
if (vOdd.empty()) {
  vec.insert(vec.begin(), 1);
}
print(vOdd);       // OOPS: 42
```

```
std::list lst{0, 8, 4, 16, 42};
auto vOdd = lst |
            std::views::filter(odd);
if (vOdd.empty()) {
  lst.push_front(1);
}
print(vOdd);       // Runtime Error: UB
```

Calling `empty()` might change the state in different ways

Filter Views and Call-by-Value

C++20

```
void printByVal(auto coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

```
void printByRef(auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

```
std::set coll{1, 2, 3, 4};
auto collEven = coll | std::views::filter(even);
if (!collEven.empty()) {
  coll.insert(0);
}
printByVal(collEven);
printByRef(collEven);
```

Output:

0 2 4

2 4

Different behavior
on **call-by-value**
and **call-by-reference**

How to Use Views

C++20

- **Create and use views ad-hoc**
- **Do not modify elements**
  - Or keep predicates alive
- **Do not use views in concurrent code**
- **Put filters early in a pipeline**
- **Pass views by rvalue references (`&&`)**
- **Be careful with `const` (might not work as expected)**

C++20

Views and `const`

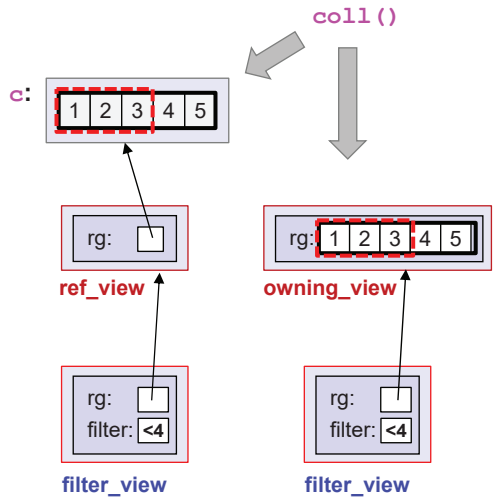
Exact Type of Views to Containers

C++20

- std::ranges::ref_view for named objects (lvalues)
- std::ranges::owning_view for temporaries (rvalues)
  - Added later as fix to C++20 with http://wg21.link/P2415

```
std::vector<int> coll()
{
  return {1, 2, 3, 4, 5};
}
```

```
std::vector<int> c = coll();
auto v1 = c | std::views::filter(lt4);
      filter_view<ref_view<vector<int>>>

auto v2 = coll() | std::views::filter(lt4);
      filter_view<owning_view<vector<int>>>
```



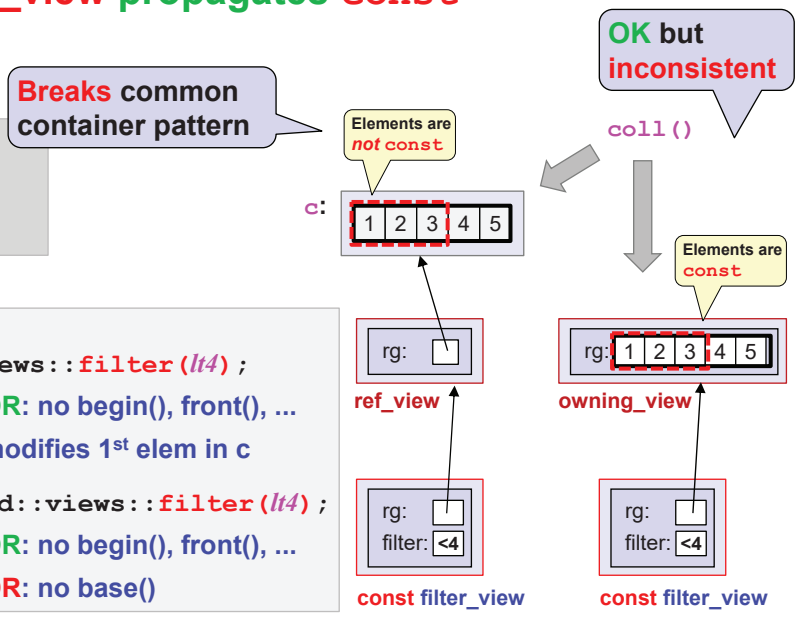
const Propagation of Views

C++20

- std::ranges::ref_view does *not* propagate const
- std::ranges::owning_view propagates const

```
std::vector<int> coll()
{
  return {1, 2, 3, 4, 5};
}
```

```
auto c = coll();
const auto v1 = c | std::views::filter(lt4);
v1.front() = 42;      // ERROR: no begin(), front(), ...
v1.base()[0] = 42;    // OK: modifies 1st elem in c

const auto v2 = coll() | std::views::filter(lt4);
v2.front() = 42;      // ERROR: no begin(), front(), ...
v2.base()[0] = 42;    // ERROR: no base()
```



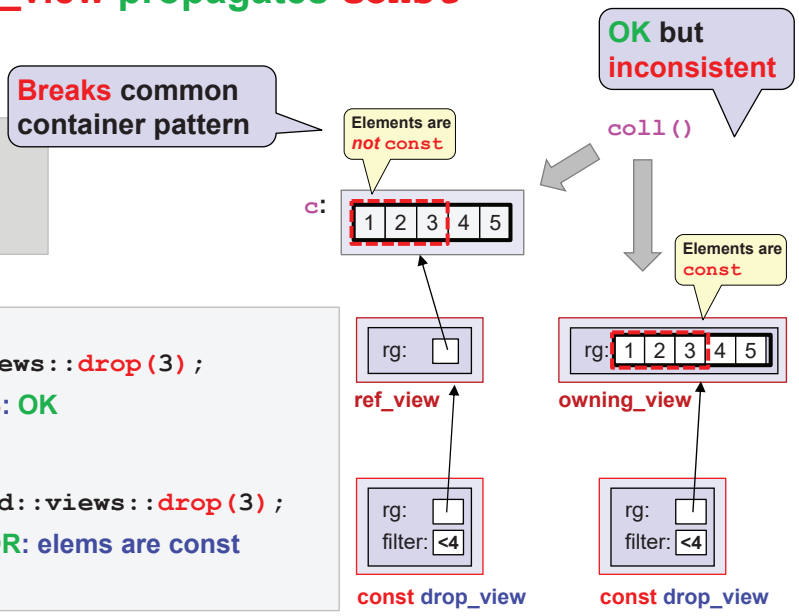
const Propagation of Views

C++20

- std::ranges::ref_view does *not* propagate const
- std::ranges::owning_view propagates const

```
std::vector<int> coll()
{
  return {1, 2, 3, 4, 5};
}
```

```
auto c = coll();
const auto v1 = c | std::views::drop(3);
v1.front() = 42;      // OOPS: OK

const auto v2 = coll() | std::views::drop(3);
v2.front() = 42;      // ERROR: elems are const
```



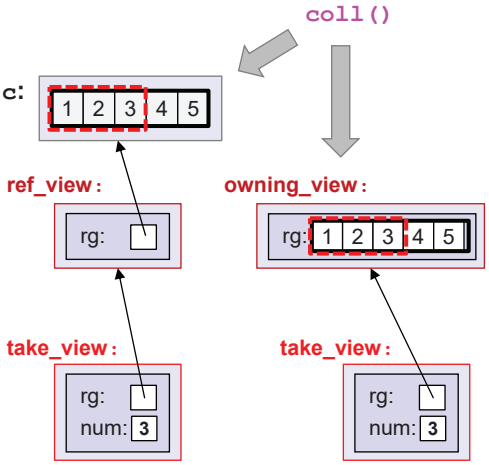
Exact Type of Views to Containers

C++20

- std::ranges::ref_view for named objects (lvalues)
- std::ranges::owning_view for temporaries (rvalues)
  - Added later as fix to C++20 with http://wg21.link/P2415

```
std::vector<int> coll()
{
  return {1, 2, 3, 4, 5};
}
```

```
std::vector<int> c = coll();
auto v1 = c | std::views::take(3);
          take_view<ref_view<vector<int>>>

auto v2 = coll() | std::views::take(3);
          take_view<owning_view<vector<int>>>
```



const Propagation of Views

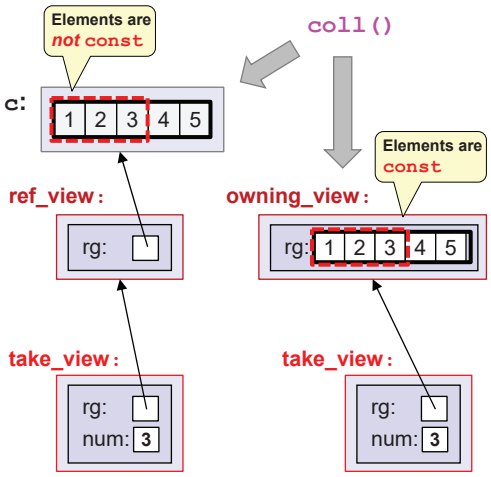
C++20

- std::ranges::ref_view does *not* propagate const
- std::ranges::owning_view propagates const

```
std::vector<int> coll()
{
  return {1, 2, 3, 4, 5};
}
```

```
std::vector<int> c = coll();
const auto v1 = c | std::views::take(3);
                take_view<ref_view<vector<int>>>

const auto v2 = coll() | std::views::take(3);
                take_view<owning_view<vector<int>>>

*v1.begin() = 42;   // OOPS: compiles and modifies
*v2.begin() = 42;   // ERROR
```

Temporary views are safer



cbegin()

C++20

```
template<typename CollT>
void foo(CollT&& coll) {
  auto pos = coll.cbegin();          // not available for views (C++20)
  std::cout << *pos;

}
```

or:
```
void foo(auto&& coll) {
  ...
}
```

C++20

```
std::vector coll{1, 2, 3, 4, 5, 6, 7};

foo(coll);                          // OK

foo(coll | std::views::take(3));    // OOPS: does not compile in C++20
foo(coll | std::views::drop(3));    // OOPS: does not compile in C++20
foo(coll | std::views::filter(…));  // OOPS: does not compile in C++20
```

cbegin() C++20

```
template<typename CollT>                                          C++20
void foo(CollT&& coll) {
  auto pos = coll.cbegin();                  // not available for views (C++20)
  auto pos2 = std::cbegin(coll);             // OK (available since C++14)
  auto pos3 = std::ranges::cbegin(coll);     // OK (available since C++20)
}
```

```
std::vector coll{1, 2, 3, 4, 5, 6, 7};
foo(coll);                                   // OK
foo(coll | std::views::take(3));             // OK
foo(coll | std::views::drop(3));             // OK
foo(coll | std::views::filter(…));           // OK
```


cbegin() C++20

```
template<typename CollT>                                          C++20
void foo(CollT&& coll) {
  auto pos = coll.cbegin();                  // not available for views (C++20)
  *std::cbegin(coll) = 0;                    // OOPS: modifies first element
  *std::ranges::cbegin(coll) = 0;            // OOPS: modifies first element
}
```

```
std::vector coll{1, 2, 3, 4, 5, 6, 7};
foo(coll);                                   // Compile-time ERROR
foo(coll | std::views::take(3));             // OOPS: compiles and modifies coll
foo(coll | std::views::drop(3));             // OOPS: compiles and modifies coll
foo(coll | std::views::filter(…));           // OOPS: compiles and modifies coll

foo(getColl() | std::views::take(3));        // Compile-time ERROR
foo(getColl() | std::views::drop(3));        // Compile-time ERROR
foo(getColl() | std::views::filter(…));      // Compile-time ERROR
```

cbegin() C++23

```
template<typename CollT>
void foo(CollT&& coll) {
  auto pos = coll.cbegin();                 // OK for views (since C++23)
  *std::cbegin(coll) = 0;                   // OOPS: still broken
  *std::ranges::cbegin(coll) = 0;           // ERROR now also for views
}
```

C++23

Don't use `std::cbegin()` since C++20
=> Prefer `std::ranges::` over `std::`

```
std::vector coll{1, 2, 3, 4, 5, 6, 7};
foo(coll);                                  // Compile-time ERROR
foo(coll | std::views::take(3));            // OOPS: std::cbegin() still broken
foo(coll | std::views::drop(3));            // OOPS: std::cbegin() still broken
foo(coll | std::views::filter(…));          // OOPS: std::cbegin() still broken

foo(getColl() | std::views::take(3));       // Compile-time ERROR
foo(getColl() | std::views::drop(3));       // Compile-time ERROR
foo(getColl() | std::views::filter(…));     // Compile-time ERROR
```


Declare Elements const C++20

```
void foo(auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';    // can't modify elem here
  }
  std::cout << '\n';
}
```

zip_view

C++23

```
void printPairs(const auto& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem.first << ':' << elem.second << '\n';
  }
}
```

```
std::vector v1{1, 2, 3};
std::vector v2{10, 20, 30};
printPairs(std::views::zip(v1, v2));
```

Output:

```
1:10
2:20
3:30
```


zip_view

C++23

```
void printPairs(const auto& coll)
{
  for (const auto& elem : coll) {
    if (elem.first == 2) {
      std::cout << "* ";
    }
    std::cout << elem.first << ':' << elem.second << '\n';
  }
}
```

```
std::vector v1{1, 2, 3};
std::vector v2{10, 20, 30};
printPairs(std::views::zip(v1, v2));
```

Output:

```
1:10
* 2:20
3:30
```

zip_view

C++23

```
void printPairs(const auto& coll)
{
  for (const auto& elem : coll) {
    if (elem.first = 2) {
      std::cout << "* ";
    }
    std::cout << elem.first << ':' << elem.second << '\n';
  }
}
```

Some views ignore this **const**

Some views ignore this **const**

```
std::vector v1{1, 2, 3};
std::vector v2{10, 20, 30};
printPairs(std::views::zip(v1, v2));
```

Output:

```
* 2:10
* 2:20
* 2:30
```


C++20

Final Aspects of Ranges and Views

C++20: Using Temporary Ranges in Algorithms

C++20

```
template<std::ranges::input_range R,
         typename T,
         typename Proj = std::identity>
requires std::indirect_binary_predicate<std::ranges::equal_to,
                                        std::projected<std::ranges::iterator_t<R>,
                                                       Proj>,
                                        const T*>
constexpr std::ranges::borrowed_iterator_t<R>
find(R&& r, const T& value, Proj proj = {});
```

```
std::vector<int> getColl();     // forward declaration

auto pos = std::ranges::find(getColl(), 42);   // find position in temporary object
std::cout << *pos;                             // runtime ERROR
```


C++20: Using Temporary Ranges in Algorithms

C++20

```
template<std::ranges::input_range R,
         typename T,
         typename Proj = std::identity>
requires std::indirect_binary_predicate<std::ranges::equal_to,
                                        std::projected<std::ranges::iterator_t<R>,
                                                       Proj>,
                                        const T*>
constexpr std::ranges::borrowed_iterator_t<R>
find(R&& r, const T& value, Proj proj = {});
```

```
std::vector<int> getColl();     // forward declaration

auto pos = std::ranges::find(getColl(), 42);   // yields type std::ranges::dangling
std::cout << *pos;                             // compile-time ERROR

const auto& coll = getColl();
auto pos = std::ranges::find(coll, 42);        // does not yield dangling iterator
if (pos != coll.end()) {                       // OK
  std::cout << *pos;
}
```

C++20: Using Temporary Ranges in Algorithms

C++20

```
template<std::ranges::input_range R,
         typename T,
         typename Proj = std::identity>
requires std::indirect_binary_predicate<std::ranges::equal_to,
                                        std::projected<std::ranges::iterator_t<R>,
                                                       Proj>,
                                        const T*>
constexpr std::ranges::borrowed_iterator_t<R>
find(R&& r, const T& value, Proj proj = {});
```

```
std::vector<int> getColl();     // forward declaration

auto pos = std::ranges::find(getColl(), 42);   // yields type std::ranges::dangling
std::cout << *pos;                             // compile-time ERROR

const auto& coll = getColl();
auto pos = std::ranges::find(coll, 42);        // does not yield dangling iterator
if (pos != coll.end()) {                       // OK
  std::cout << *pos;
}
```


Using Big Pipelines

C++20

```
// view 4th to 11th value that are multiples of 3 with suffix "s":
auto v = std::views::iota(1)                          // values from 1
         | std::views::filter([] (auto val) {         // multiples of 3 only
                                return val % 3 == 0;
                              })
         | std::views::drop(3)                        // skip first 3
         | std::views::take(8)                        // take next 8
         | std::views::transform([] (auto val) {      // append "s"
                                   return std::to_string(val) + "s";
                                 });
```

```
for (const auto& elem : v) {
  std::cout << elem << '\n';
}
```

Output:

```
12s
15s
18s
...
33s
```

```
std::vector<std::string> coll{v.begin(), v.end()};   // ERROR
```

Common View

C++20

```
// view 4th to 11th value that are multiples of 3 with suffix "s":
auto v = std::views::iota(1)                          // values from 1
       | std::views::filter([] (auto val) {           // multiples of 3 only
                              return val % 3 == 0;
                            })
       | std::views::drop(3)                          // skip first 3
       | std::views::take(8)                          // take next 8
       | std::views::transform([] (auto val) {        // append "s"
                                 return std::to_string(val) + "s";
                               });
```

random-access common view

bidirectional common view

no longer common (due to bidirectional)

```
for (const auto& elem : v) {
  std::cout << elem << '\n';
}

std::vector<std::string> coll{v.begin(), v.end()};   // ERROR: itor types differ
auto b = std::ranges::common_range<decltype(v)>;     // false

auto cv = v | std::views::common;                    // harmonize iterator types
std::vector<std::string> vec{cv.begin(), cv.end()};  // OK
```


std::ranges::to<>()

C++23

```
// view 4th to 11th value that are multiples of 3 with suffix "s":
auto v = std::views::iota(1)                          // values from 1
       | std::views::filter([] (auto val) {           // multiples of 3 only
                              return val % 3 == 0;
                            })
       | std::views::drop(3)                          // skip first 3
       | std::views::take(8)                          // take next 8
       | std::views::transform([] (auto val) {        // append "s"
                                 return std::to_string(val) + "s";
                               });

auto cv = v | std::views::common;                    // harmonize iterator types
std::vector<std::string> vec{cv.begin(), cv.end()};  // OK
std::println("vec:    {}", vec);

auto vec23 = v | std::ranges::to<std::vector>();     // store in vector (C++23)
std::println("vec23: {}", vec23);
```

Output:

```
vec:    ["12s", "15s", "18s", … "33s"]
vec23: ["12s", "15s", "18s", … "33s"]
```

https://godbolt.org/z/xbPj61Pa7

Basic Idioms Broken by Standards Views

C++20/C++23

- You can iterate if the range is `const` — Broken for views
- A read iteration does not change state — Broken for views
- Concurrent read iterations are safe — Broken for views
- `const` collections have `const` elements — Broken for views
- `cbegin()` makes elements immutable — Broken for views
- A copy of a range has the same state — Broken for views
- `const`-declared elements are `const` (C++23) — Broken for views

How to Use Views

C++20

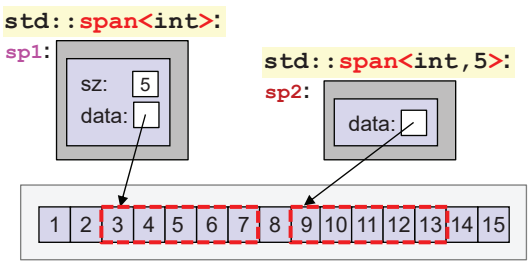
- Create and use views ad-hoc
- Do not modify elements
  - Or keep predicates alive
- Do not use views in concurrent code
- Put filters early in a pipeline
- Pass views by rvalue references (`&&`)
- Be careful with `const` (might not work as expected)

C++20

std::span<>

C++20: std::span<>

- **Range API for elements in contiguous memory**
  - Generalization of what `std::string_view` is for character sequences
  - Supports write access
  - Supports dynamic or fixed extent
    - Dynamic or fixed number of elements
  - A span is a view
    - Has **reference semantics**
    - Does **not** propagate `const`
    - Satisfies concept `std::ranges::view`




```
int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};

std::span<int> sp1{arr + 2, 5};          // span with dynamic extent to: 3 ... 7
for (auto elem : sp1) { … }              // iterates over 5 elements of arr using begin() and end()
process(sp1.last(3));                    // processes last 3 elements of sp1 in arr
sp1[0] = 42;                             // modifies 3rd element of arr
std::ranges::sort(sp1);                  // sorts 5 elements of sp1 in arr
std::span<int,5> sp2{arr + 8, 5};        // span with fixed extent to: 9 ... 13
```

Example Using a Span with Dynamic Extent

C++20

```
std::vector<std::string> vec{"Rome", "Tokyo", "Cairo", "Berlin", "Sydney"};
print("vec:      ", vec);

// view to first 3 elements:
std::span<std::string> sp{vec.data(), 3};          // note: elements are not const
print("first 3: ", sp);

// sort elements in the referenced vector:
std::ranges::sort(sp);
print("vec:      ", vec);
print("sorted:   ", sp);

// insert new element:
vec.push_back("Rio");
print("more:     ", sp);    // Runtime ERROR
// - must re-assign reallocated memory of vector
sp = std::span{vec.data(), 3};
print("more:     ", sp);    // OK

// let span refer to the vector as a whole:
sp = vec;
print("all:      ", sp);

// constant view to first 4 elements:
const std::span<std::string> spc{vec.data(), 4};   // note: elements are still not const
std::ranges::sort(spc);    // OK !
print("vec:      ", vec);
```

Output:

```
vec:      Rome Tokyo Cairo Berlin Sydney
first 3: Rome Tokyo Cairo
vec:      Cairo Rome Tokyo Berlin Sydney
sorted:   Cairo Rome Tokyo
more:     undefined behavior
more:     Cairo Rome Tokyo
all:      Cairo Rome Tokyo Berlin Sydney Rio
vec:      Berlin Cairo Rome Sydney Tokyo Rio
```


Example Using a Span with Dynamic Extent

C++20

```
std::vector<std::string> vec{"Rome", "Tokyo", "Cairo", "Berlin", "Sydney"};
print("vec:      ", vec);

// view to first 3 elements:
std::span sp{vec.data(), 3};                       // note: elements are not const
print("first 3: ", sp);

// sort elements in the referenced vector:
std::ranges::sort(sp);
print("vec:      ", vec);
print("sorted:   ", sp);

// insert new element:
vec.push_back("Rio");
print("more:     ", sp);    // Runtime ERROR
// - must re-assign reallocated memory of vector
sp = std::span{vec.data(), 3};
print("more:     ", sp);    // OK

// let span refer to the vector as a whole:
sp = vec;
print("all:      ", sp);

// constant view to first 4 elements:
const std::span spc{vec.data(), 4};                // note: elements are still not const
std::ranges::sort(spc);    // OK !
print("vec:      ", vec);
```

You can skip the element type due to Class Template Argument Deduction (C++17)

Output:

```
vec:      Rome Tokyo Cairo Berlin Sydney
first 3: Rome Tokyo Cairo
vec:      Cairo Rome Tokyo Berlin Sydney
sorted:   Cairo Rome Tokyo
more:     undefined behavior
more:     Cairo Rome Tokyo
all:      Cairo Rome Tokyo Berlin Sydney Rio
vec:      Berlin Cairo Rome Sydney Tokyo Rio
```

Example Using a Span with Dynamic Extent

C++20

```
std::vector<std::string> vec{"Rome", "Tokyo", "Cairo", "Berlin", "Sydney"};
print("vec:      ", vec);

// view to first 3 elements:
std::span<const std::string> sp{vec.data(), 3};     // note: elements are const
print("first 3: ", sp);

// sort elements in the referenced vector:
std::ranges::sort(sp);      // ERROR
std::ranges::sort(vec);     // OK
print("vec:      ", vec);
print("sorted:   ", sp);

// insert new element:
vec.push_back("Rio");
print("more:     ", sp);     // Runtime ERROR
// - must re-assign reallocated memory of vector
sp = std::span{vec.data(), 3};
print("more:     ", sp);     // OK

// let span refer to the vector as a whole:
sp = vec;
print("all:      ", sp);

// let span refer to the last five elements:
sp = std::span{vec.end()-5, vec.end()};
print("last 5:   ", sp);
```

Output:

```
vec:      Rome Tokyo Cairo Berlin Sydney
first 3: Rome Tokyo Cairo

vec:      Berlin Cairo Rome Sydney Tokyo
sorted:   Berlin Cairo Rome

more:     undefined behavior
more:     Berlin Cairo Rome

all:      Berlin Cairo Rome Sydney Tokyo Rio

last 5:   Cairo Rome Sydney Tokyo Rio
```

Also with:

```
sp = std::span{vec}.last(5);
```


Example Using a Span with Fixed Extent

C++20

```
std::vector<std::string> vec{"Rome", "Tokyo", "Cairo", "Berlin", "Sydney"};
print("vec:      ", vec);

// view to first 3 elements:
std::span<std::string, 3> sp{vec.data(), 3};        // note: elements are not const
print("first 3: ", sp);

// sort elements in the referenced vector:
std::ranges::sort(sp);
print("vec:      ", vec);
print("sorted:   ", sp);

// insert new element:
vec.push_back("Rio");
print("more:     ", sp);     // Runtime ERROR
// - must re-assign reallocated memory of vector
sp = std::span{vec.data(), 3};
print("more:     ", sp);     // OK

// let span refer to the vector as a whole:
sp = vec;                   // ERROR: wrong size
```

Output:

```
vec:      Rome Tokyo Cairo Berlin Sydney
first 3: Rome Tokyo Cairo

vec:      Cairo Rome Tokyo Berlin Sydney
sorted:   Cairo Rome Tokyo

more:     undefined behavior
more:     Cairo Rome Tokyo

all:      Cairo Rome Tokyo Berlin Sydney Rio
```

std::span<> Operations

C++20

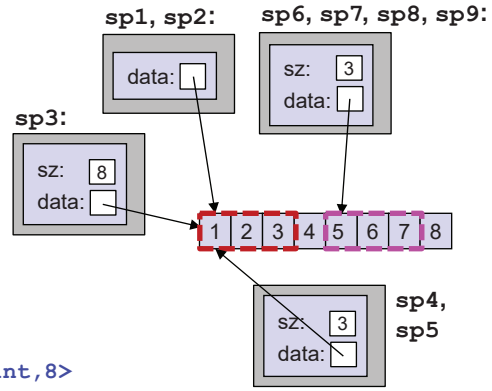
Operation	Effect
constructors	Creates or copies a span
destructor	Destroys a span
=	Assigns a new sequence of values
sp.empty()	Returns whether the span is empty
sp.size()	Returns the number of elements
sp.size_bytes()	Returns the size of memory used for all elements
sp.begin(), *sp*.end()	Return begin and end iterators
sp.rbegin(), *sp*.rend()	Return iterators for reverse iterations
sp[*idx*]	Accesses an element
sp.front(), *sp*.back()	Accesses the first or last element
sp.first(*n*)	Returns a sub-span with dynamic extent of the first *n* elements
sp.first<*n*>()	Returns a sub-span with fixed extent of the first *n* elements
sp.last(*n*)	Returns a sub-span with dynamic extent of the last *n* elements
sp.last<*n*>()	Returns a sub-span with fixed extent of the last *n* elements
sp.subspan(*offs*)	Returns a sub-span with dynamic extent skipping the first *offs* elements
sp.subspan(*offs*, *n*)	Returns a sub-span with dynamic extent of *n* after *offs* elements
sp.subspan<*offs*>()	Returns a sub-span with same extent skipping the first *offs* elements
sp.subspan<*offs*, *n*>()	Returns a sub-span with fixed extent of *n* after *offs* elements
sp.data()	Returns a raw pointer to the elements
sp.as_bytes()	Returns the memory of the elements as a span of read-only std::bytes
sp.as_writable_bytes()	Returns the memory of the elements as a span of writable std::bytes

- **No** comparisons (not even ==), swap(), assign(), at(), I/O
- cbegin(), cend(), crbegin(), and crend() with C++23

std::span<> Initializations

C++20

- **Spans may be initialized by**
  - Raw array or std::array
  - Continuous sized range
  - Begin and end/sentinel
  - Begin and size
  - std::views::counted(*beg*, *sz*)




```
int a8[] = {1, 2, 3, 4, 5, 6, 7, 8};
std::array arr{1, 2, 3, 4, 5, 6, 7, 8};
std::vector vec{1, 2, 3, 4, 5, 6, 7, 8};

std::span sp1 = a8;                  // std::span<int,8>
std::span sp2 = arr;                 // std::span<int,8>
std::span sp3 = vec;                 // std::span<int> (initially 8 elements)
std::span sp4{a8, 3};                // std::span<int> (initially 3 elements)
std::span sp5{arr.data(), 3};        // std::span<int> (initially 3 elements)
std::span sp6{a8 + 4, 3};            // std::span<int> (initially 3 elements)
std::span sp7{vec.data() + 4, 3};    // std::span<int> (initially 3 elements)
std::span sp8{std::ranges::find(arr, 5),          // std::span<int> (initially 3 elements)
              std::ranges::find(arr, 8)};
auto sp9 = std::views::counted(arr.data() + 4, 3);  // std::span<int> (initially 3 elements)
std::span sp0{arr.data(), 10};       // undefined behavior (arr doesn't have 10 elements)
```

std::span<> Caveats

C++20

- **Don't use spans when they refer to invalid elems/memory**
- **Spans do not propagate `const`**

```
std::vector<std::string> vec1{"one", "two", "3", "some other long string"};
std::span sp1 = vec1;          // span with 4 elements

vec1.clear();                  // remove/destroy all elements
print(sp1);                    // RUNTIME ERROR (printing destroyed strings)

vec1 = {"other", "values"};
sp1 = vec1;                    // span now has 2 elements
print(sp1);                    // OK
```

```
std::vector<std::string> vec2{"one", "two", "three", "four"};
const std::span sp2 = vec2;
std::ranges::sort(sp2);        // OK (elements are not const)
print(vec2);                   // elements in vec2 are sorted

std::span<const std::string> sp3 = vec2;
std::ranges::sort(sp3);        // ERROR: elements are const

std::span sp4 = std::as_const(vec2);
std::ranges::sort(sp4);        // ERROR: elements are const
```


std::span<> Caveats

C++20

- **Span sizes have to fit**
  - Otherwise, you get compile-time or runtime errors

```
std::vector v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::span<int> spDyn{v.data()+3, 5};
print(spDyn.last(3));                    // print span with dynamic extent
print(spDyn.last<3>());                  // print span with fixed extent
print(spDyn.last(6));                    // runtime ERROR (undefined behavior)
print(spDyn.last<6>());                  // runtime ERROR (undefined behavior)

std::span<int, 2> spTmp{v.data()+3, 5};  // runtime ERROR (undefined behavior)

std::span<int, 5> spFix{v+3, 5};         // OK
print(spFix.last(3));                    // print span with dynamic extent
print(spFix.last<3>());                  // print span with fixed extent
print(spFix.last(6));                    // runtime ERROR  (undefined behavior)
print(spFix.last<6>());                  // compile-time ERROR

spFix = spDyn;                           // compile-time ERROR
spFix = v;                               // compile-time ERROR
spDyn = spFix;                           // OK (spDyn has 5 elements)
spDyn = v;                               // OK (spDyn has 10 elements)
```

Passing Spans as Parameters

C++20

Spans are usually passed by value

```
template<typename T, std::size_t Sz>
void printSpan(std::span<T, Sz> sp)   // for all spans
{
  for (const auto& elem : sp) {
    std::cout << '"' << elem << "\" ";
  }
  std::cout << '\n';
}
```

For dynamic span's, Sz is `std::dynamic_extent`

```
std::vector<std::string> vec{"New York", "Tokyo", "Rio", "Berlin", "Sydney"};

std::span<const std::string, 3> sp3{vec.begin(), 3};   // fixed span with const elements
printSpan("sp3", sp);

std::span<std::string, 3> sp3b{vec.begin(), 3};        // fixed span with non-const elements
printSpan("sp3b", sp3b);

std::span<const std::string> sp{vec.begin(), 3};       // dynamic span with const elements
printSpan("sp", sp);

printSpan(vec);              // ERROR: template type deduction fails
printSpan(std::span{vec});   // OK
```


C++20: Conditional `explicit` for `std::span<>`

C++20

- **Conditional `explicit` is used in the Standard Library**
  - **`std::pair`, `std::tuple`, `std::optional`, `std::span`, ...**

```
namespace std {
  template<typename ElemType, size_t Extent = dynamic_extent>
  class span {
    ...
    template<typename It>
    constexpr explicit(extent != dynamic_extent) span(It beg, size_type count);
    template<typename It1, typename It2>
    constexpr explicit(extent != dynamic_extent) span(It1 beg, It2 end);
    ...
}
```

Support implicit type conversions only for spans with dynamic extent

```
void fooDyn(std::span<int>);
void fooFix(std::span<int, 3>);

std::vector coll{1, 2, 3, 4, 5};
fooDyn({coll.begin(), 3});                  // OK
fooFix({coll.begin(), 3});                  // ERROR: no implicit conversion supported

std::span<int> spDyn1{coll.begin(), 3};     // OK: direct initialization
std::span<int> spDyn2 = {coll.begin(), 3};  // OK: copy initialization with implicit conversion
std::span<int, 3> spFix1{coll.begin(), 3};  // OK: direct initialization
std::span<int, 3> spFix2 = {coll.begin(), 3};  // ERROR: no implicit conversion supported
fooDyn(spFix1);                             // OK
fooFix(spDyn1);                             // ERROR: no implicit conversion supported
```

C++23: std::mdspan<>

C++23

```
std::vector v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};

// 2 rows of 6 values each:
std::mdspan mds1{v.data(), 2, 6};
print2D(mds1);

// fixed or dynamic extent:
std::mdspan<int, std::extents<std::size_t, 2, 6>> mds2{v.data()};
std::mdspan<int,
            std::extents<std::size_t, std::dynamic_extent, std::dynamic_extent>,
            std::layout_right> mds3{v.data(), 2, 6};

// 2 rows of 6 values each (opposite layout):
std::mdspan<int, std::extents<std::size_t, 2, 6>, std::layout_left> mds4{v.data()};
print2D(mds4);

// 3 rows (every 3rd elem) of 4 values:
std::mdspan<int,
            std::extents<size_t, std::dynamic_extent, std::dynamic_extent>,
            std::layout_stride> mds5{v.data(),
                                     {std::dextents<size_t,2>{3, 4},
                                      std::array{1, 3}}};
print2D(mds5);
```

Output:

```
1 2 3 4 5 6
7 8 9 10 11 12

1 3 5 7 9 11
2 4 6 8 10 12

1 4 7 10
2 5 8 11
3 6 9 12
```

https://godbolt.org/z/K785xbK9e

C++20

Formatted Output

C++20: Formatted Output

C++20

- **Flexible formatting like with `sprintf()`**
  - type safe
  - extensible

```
#include <format>

std::string str{"hello"};
std::cout << std::format("String '{}' has {} characters\n", str, str.size());
                                  // => String 'hello' has 5 characters
std::cout << std::format("{1} is the size of string '{0}'\n", str, str.size());
                                  // => 5 is the size of string 'hello'

std::format("{:7}",     42)          // yields "     42"
std::format("{:7}",     42.0)        // yields "     42"
std::format("{:7.2f}", 42.0)         // yields "  42.00"
std::format("{:7}",     true)        // yields "true   "
std::format("{:7}",     '@')         // yields "@      "
std::format("{0}: {0:+07d}", '@')    // yields "@: +000064"

std::format("{:<7}",  42)            // yields "42     "
std::format("{:^7}",  42)            // yields "  42   "
std::format("{:*^7}",  42)           // yields "**42***"
std::format("{:7b}", 1000)           // yields "1111101000"
```

Yields `std::string` (with allocated memory)

C++20: Formatted Output

C++20

- **You can write formatted**
  - directly into reserved/existing memory
  - directly to a stream

```
// formatted output to a buffer:
char buffer[64];
auto ret = std::format_to_n(buffer, std::size(buffer) - 1,
                            "String '{}' has {} chars\n", str, str.size());
buffer[ret.size] = '\0';   // append trailing null terminator
std::cout << "buffer: " << buffer << '\n';

// formatted output directly to std::cout:
std::format_to(std::ostreambuf_iterator<char>{std::cout},
               "String '{}' has {} chars\n", str, str.size());

// append formatted output to a string s:
std::string s;
std::format_to(std::back_inserter(s),
               "String '{}' has {} chars\n", str, str.size());
```

might cut output if not enough room

C++20: Formatting Type Specifiers

C++20

Spec.	Meaning	42	'@'	true	"hello"	-1.0	1785856.0
Default format with {}		42	@	true	hello	-1	1.785856e+06
d	decimal	42	64	1			
b, B	binary	101010	1000000	1			
#b, #B	binary with prefix	0b101010	0b100000	0b1			
o	octal	52	100	1			
x, X	hex	2a	40	1			
#x, #X	hex with prefix	0x2a	0x40	0x1			
c	character	*	@				
s	string			true	hello		
#	force dot					-1.	1.785856e+06
g, G	general (fixed or exponential)					-1	1.78586e+06
#g, #G	general (force dot/zeros)					-1.00000	1.78586e+06
f, F	fixed					-1.000000	1785856.000000
e, E	exponential					-1.000000e+00	1.785856e+06
a, A	hex float					-1p+0	1.b4p+20
p	hex ptr (void* or nullptr)						

Also strings and string views

C++20: Formatting Format

C++20

- **Format strings:**

  { arg-id$_{opt}$: fill$_{opt}$ align$_{opt}$ *sign*$_{opt}$ #$_{opt}$ 0$_{opt}$ *width*$_{opt}$.*prec*$_{opt}$ L$_{opt}$ *type*$_{opt}$ }

  - align: < (left), > (right), ^ (centered)
  - sign: - (– only (default)), + (– or +), *space* (– or *space*)
  - #: alternate form (prefix for int, force dot for floats)
  - 0: leading 0
  - *width*: integer or { arg-id$_{opt}$ }
  - *precision*: integer or { arg-id$_{opt}$ }
  - L: locale-specific form
  - type: d (decimal), b,B, o, x,X (bin, oct, hex),
    f,F, e,E, g,G, a,A (fixed, scientific, general, hex float),
    c (char), s (string), p (hex ptr)

```
int width = 10;
int precision = 3;
std::format("{:0{}.{}}", 12.345678, width, precision);      // yields "00000012.3"
std::format("{0:0{1}.{2}f}", 12.345678, width, precision);  // yields "000012.346"
```

C++20: Formatting Error Handling

C++20

- **Format errors are detected at compile time (if possible)**
  - According to C++20 fixes with http://wg21.link/p2216r3
- **Runtime format errors throw `std::format_error`**

```
try {
  std::format("{}:{}", 42, 4.2)                          // OK

  std::format("{}:{}", 42)                               // compile-time ERROR

  const char* fmt1 = "{}:{}";                            // runtime format string
  std::format(fmt1, 42, 4.2)                             // compile-time ERROR

  constexpr const char* fmt2 = "{}:{}";                  // compile-time format string
  std::format(fmt2, 42, 4.2)                             // OK

  const char* fmt3 = "{}:{}";
  std::vformat(fmt3, std::make_format_args(42, 4.2))     // OK

  const char* fmt4 = "{}:{}";
  std::vformat(fmt4, std::make_format_args(42))    // std::format_error exception
}
catch (const std::format_error& e) {
  std::cerr << "FORMATTING EXCEPTION: " << e.what() << std::endl;
}
```


C++20: Performance of Formatted Output

C++20

- **`format()` should have better performance than `sprintf()`**
  - Especially with compile-time parsing (see wg21.link/p2216)
  - Implementations are working on it
    - On VC++, compiling with `/utf-8` might make a difference
- **Some numbers:**
  (different platforms, very rough tendencies)
  - Marc Gregoire 2019: http://youtu.be/Y652wQqbYEI?t=3168
  - Nico Josuttis 2023: https://www.godbolt.org/z/MEz9PYTEb

	M. Gregoire	N. Josuttis				
Method	clang	gcc		clang	VC++	
`sprintf()`	882	343	2337	309	199	251
`ostringstream`	1892	448	2537	314	579	606
`std::to_string()`	1167	365	2461	219	281	280
`std::to_chars()`		143	290	94	55	54
`std::format()`	676	338	289	236	151	152
`std::format_to()`	499	343	292	251	125	124

No formatting, just the value

C++20: `std::format()` Using a Locale

C++20

```
#include <locale>
#include <format>
...

std::locale getLocale()
{
  // try to use German locale:
#ifdef _MSC_VER
  auto nameDE = "deu_deu.1252";
#else
  auto nameDE = "de_DE";
#endif
  try {
    return std::locale{nameDE};     // return German locale
  }
  catch (const std::exception& e) {
    std::cerr << "EXCEPTION: " << e.what() << '\n';
    std::cerr << " '" << nameDE << "' not supported\n";
    return std::locale{""};         // return locale from environment
  }
}

// initialize locale and use it for a floating-point value:
std::locale loc = getLocale();
std::cout << std::format(loc, "{0} {0:L}\n", 1000.7);  // 1000.7 1.000,7
```

Possible Output:

```
1000.7 1.000,7
```

Possible Output :

```
EXCEPTION: ...
 'de_DE' not supported
1000.7 1000.7
```


C++20: `std::format()` with User-Defined Facet

C++20

```
#include <locale>
#include <format>
...

class GermanBoolNames : public std::numpunct_byname<char> {
 public:
  GermanBoolNames (const std::string& name)
    : std::numpunct_byname<char>(name) {
  }
 protected:
  virtual std::string do_truename() const override {
    return "wahr";
  }
  virtual std::string do_falsename() const override {
    return "falsch";
  }
};

// create locale with German bool names:
std::locale locBool{std::cout.getloc(), new GermanBoolNames{""}};

// use local to print Boolean values:
std::cout << std::format(locBool, "{0} {0:L}\n", false);
```

Output:

```
false falsch
```

C++20: Implementing a Formatter

C++20

- **You can provide user-defined formatters**

```
MyType x = 42;
std::format("{}", x)       // OK: "42"
std::format("{0:7}", x)    // OK: "     42"
std::format("{:^7}", x)    // OK: "  42   "
std::format("{:_^7}", x)   // OK: "__42___"
```

- **You have to specialize `std::formatter<>`**
  - Implement the member function `parse()`
    - for parsing the (relevant part of the) format string
    - callable at compile time
  - Implement the member function `format()`
    - for performing the formatting
- **Can be simplified by delegating to or deriving from standard formatters**

Example of Implementing a Formatter

C++20

```
template<>
class std::formatter<MyType>
{
  int width = 0;   // specified width
 public:
  constexpr auto parse(std::format_parse_context& ctx) -> decltype(ctx.begin()) {
    auto pos = ctx.begin();
    for ( ; pos != ctx.end() && *pos != '}'; ++pos) {
      if (*pos < '0' || *pos > '9') {
        throw std::format_error{"MyType only allows digits for width"};
      }
      width = width * 10 + *pos - '0'; // process next digit for the width
    }
    if (width == 0) width = 1;   // format_to() below needs a positive width
    return pos;   // return position of '}'
  }

  auto format(const MyType& v, std::format_context& ctx) const -> decltype(ctx.out())
  {
    return std::format_to(ctx.out(), "{:{}}", v.getValue(), width);
  }
};
```

ctx.begin()

```
MyType x = 42;
std::format("{}\n", x)     // OK
std::format("{0:20}", x)   // OK
std::format("{:^7}", x)    // Error/Exception
```

code for compile-time computing (except `throw` on error)

Formatter Delegating to Standard Formatters

C++20

```
class MyType {
 public:
  …
  int getValue() const;
};
```

Delegate the work to the formatter of the type the getter return

```
template<>
class std::formatter<MyType>
{
  std::formatter<int> f;                    // standard formatter that does the work
 public:
  constexpr auto parse(std::format_parse_context& ctx) {
    return f.parse(ctx);                    // delegate parsing of the format string
  }
  auto format(const MyType& obj, std::format_context& ctx) const {
    return f.format(obj.getValue(), ctx);   // delegate formatting of the value
  }
};
```

```
MyType x{42};
std::format("{}mm", x)          // OK: "42mm"
std::format("{0:7}mm", x)       // OK: "     42mm"
std::format("{:^7}", x)         // OK: "  42   "
std::format("{:_^7}", x)        // OK: "__42___"
std::format("{0:X}:{0}", x);    // OK: "2A:42"
```


Formatter Deriving from Standard Formatters

C++20

```
class MyType {
 public:
  …
  int getValue() const;
};
```

Delegate the work to the formatter of the type the getter returns
- Formatter and `parse()` are derived

```
template<>
struct std::formatter<MyType> : std::formatter<int>
{
  auto format(const MyType& obj, std::format_context& ctx) const {
    return std::formatter<int>::format(obj.getValue(), ctx); // format value
  }
};
```

```
MyType x{42};
std::format("{}mm", x)          // OK: "42mm"
std::format("{0:7}mm", x)       // OK: "     42mm"
std::format("{:^7}", x)         // OK: "  42   "
std::format("{:_^7}", x)        // OK: "__42___"
std::format("{0:X}:{0}", x);    // OK: "2A:42"
```

C++20: Formatters Using String Formatters

C++20

```
enum class Color { red, green, blue };

template<> struct std::formatter<Color> : std::formatter<std::string> {
  auto format(Color c, format_context& ctx) const {
    std::string value;
    switch (c) {
      using enum Color;
      case red:   value = "red";
                  break;
      case green: value = "green";
                  break;
      case blue:  value = "blue";
                  break;
      default:    value = std::format("Color{}", static_cast<int>(c));
                  break;
    }
    return std::formatter<std::string>::format(value, ctx);
  }
};

for (auto col : {Color::red, Color::green, Color::blue, Color{13}}) {
  std::cout << std::format(" {:.^8} for {:>2}\n", col,
                                                  static_cast<int>(col));
}
```

If only string literals are used, use `std::string_view` formatter

Output:

```
 ..red... for  0
 .green.. for  1
 ..blue.. for  2
 Color13. for 13
```


Formatted Output for Containers/Ranges

C++23

```
...
#include <format>
#include <print>

int main()
{
  std::vector<int> coll{0, 8, 15, 47, 11, -1};

  std::cout << std::format("{}\n", coll);
  std::cout << std::format("{::+}\n", coll);
  std::cout << std::format("{::03x}\n", coll);

  std::print("{1} elems: {0}\n", coll, coll.size());
  std::println("coll: {}", coll);
  std::println("");

  std::vector<std::string> cities{"Kiev", "Rio", "Sydney", "Berlin"};
  std::cout << std::format("{}\n", cities);
  std::cout << std::format("{::}\n", cities);
  std::cout << std::format("{::.3}\n", cities);
}
```

Output:

```
[0, 8, 15, 47, 11, -1]
[+0, +8, +15, +47, +11, -1]
[000, 008, 00f, 02f, 00b, -01]

6 elems: [0, 8, 15, 47, 11, -1]
coll: [0, 8, 15, 47, 11, -1]

["Kiev", "Rio", "Sydney", "Berlin"]
[Kiev, Rio, Sydney, Berlin]
[Kie, Rio, Syd, Ber]
```

https://www.godbolt.org/z/56zqc45ra

C++20

Calendars and Time Zones

Example of Chrono Durations

C++11

```
#include <chrono>
...
```

Shortcut for
`std::chrono::duration<int, std::milli>`
or
`std::chrono::duration<int, std::ratio<1,1000>>`

```
std::chrono::milliseconds a{10};
```

a: 10 $\frac{1}{1000}$

```
std::chrono::duration<double, std::ratio<2,3>> b{7.9};
```

b: 7.9 $\frac{2}{3}$

```
auto c = a + b;   // compiler computes resulting type 1/3000
                  // and generates code for: c = a*3 + b*2000
```

c: 15830.0 $\frac{1}{3000}$

```
std::this_thread::sleep_for(c);

std::cout << c.count() << ' '
          << decltype(c)::period::num << '/'
          << decltype(c)::period::den << "s\n";

std::chrono::milliseconds d1{c};        // ERROR
std::chrono::duration<double, std::milli> d2{c};
std::cout << d2.count() << "ms\n";      // OK
auto d3 = std::chrono::round<std::chrono::milliseconds>(c);
std::cout << d3.count() << "ms\n";      // OK
```

Output:

```
15830 1/3000s


5276.67ms

5277ms
```

Example of Chrono Durations with Time Literals

C++14

```
#include <chrono>
...
using namespace std::literals;   // or: using namespace std;
auto a = 10ms;

std::chrono::duration<double, std::ratio<2,3>> b{7.9};

auto c = a + b;   // compiler computes resulting type 1/3000
                  // and generates code for: c = a*3 + b*2000

std::this_thread::sleep_for(c);

std::cout << c.count() << ' '
          << decltype(c)::period::num << '/'
          << decltype(c)::period::den << "s\n";

std::chrono::milliseconds d1{c};        // ERROR
std::chrono::duration<double, std::milli> d2{c};
std::cout << d2.count() << "ms\n";      // OK
auto d3 = std::chrono::round<std::chrono::milliseconds>(c);
std::cout << d3.count() << "ms\n";      // OK
```

a: 10 1/1000

b: 7.9 2/3

c: 15830.0 1/3000

Output:

```
15830 1/3000s


5276.67ms

5277ms
```


Example of Chrono Durations with Time Literals

C++11 / C++20

```
#include <chrono>
...
using namespace std::literals;   // or: using namespace std;
auto a = 10ms;

std::chrono::duration<double, std::ratio<2,3>> b{7.9};

auto c = a + b;   // compiler computes resulting type 1/3000
                  // and generates code for: c = a*3 + b*2000

std::this_thread::sleep_for(c);

std::cout << c.count() << ' '
          << decltype(c)::period::num << '/'
          << decltype(c)::period::den << "s\n";

std::chrono::milliseconds d1{c};        // ERROR
std::chrono::duration<double, std::milli> d2{c};
std::cout << d2.count() << "ms\n";      // OK
auto d3 = std::chrono::round<std::chrono::milliseconds>(c);
std::cout << d3.count() << "ms\n";      // OK
std::cout << d3 << " " << c << '\n';    // since C++20
```

a: 10 1/1000

b: 7.9 2/3

c: 15830.0 1/3000

Output:

```
15830 1/3000s


5276.67ms

5277ms
5276ms 15830[1/3000]s
```

Measurements with `std::chrono`

C++11 / C++20

```
#include <chrono>
...

auto t0 = std::chrono::steady_clock::now();
  ... // do whatever we have to measure
auto t1 = std::chrono::steady_clock::now();

auto diff = std::chrono::round<std::chrono::milliseconds>(t1 - t0);
std::cout << "diff:  " << diff.count() << "ms\n";

std::cout << "diff:  " << diff << '\n';                    // since C++20
std::cout << "diff:  " << t1 - t0 << '\n';                 // since C++20

std::chrono::duration<double, std::milli> diff2 = t1 - t0;
std::cout << "exact: " << diff2.count() << "ms\n";

std::cout << "exact: " << diff2 << '\n';                   // since C++20

std::cout << std::format("{} aka {}\n", t1 - t0, diff2);   // since C++20
```

Possible Output:

```
diff:  80ms
diff:  80ms
diff:  79810300ns
exact: 79.8103ms
exact: 79.8103ms
79810300ns aka 79.8103ms
```


C++20: Chrono with Calendar and Time-Zone Library

C++20

- **Extensions to `<chrono>`** (introduced with C++11):
  - Direct (formatted) output of
    - Durations, time points, dates, ...
  - Calendar functionality
    - Canonical calendar sys_days (days from 1.1.70)
    - Civil calendar with various typical representations and types
  - New alias names for durations, timepoints, months, weekdays, logical days (e.g. "last"), ...
    - e.g.: `month`, `months`, `November`, `last`, `Sunday[2]`
  - Time zone support
  - New clocks
    - `utc_clock`, `gps_clock`, `tai_clock`, `file_clock`, `local_t`
    - Conversions between clocks
- **Type safe, fast/small, `constexpr` as possible**

Let's Have a Party

C++20

```
#include <iostream>
#include <chrono>
using namespace std::literals;


auto firstDay = 2021y / 1 / 1;
auto lastDay = 31d / 12 / 2021;

for (auto d = firstDay; d <= lastDay; d += std::chrono::months{1}) {
  std::cout << d << ": Party\n";
}
```

Output:

```
2021-01-01: Party
2021-02-01: Party
2021-03-01: Party
2021-04-01: Party
2021-05-01: Party
2021-06-01: Party
2021-07-01: Party
2021-08-01: Party
2021-09-01: Party
2021-10-01: Party
2021-11-01: Party
2021-12-01: Party
```

firstDay is initialized as follows:

```
std::chrono::year / int
=> std::chrono::year_month / int
   => std::chrono::year_month_day
```

lastDay is initialized as follows:

```
std::chrono::day / int
=> std::chrono::month_day / int
   => std::chrono::year_month_day
```


Let's Have a Party

C++20

```
#include <iostream>
#include <chrono>
using namespace std::literals;


auto firstDay = std::chrono::January / 1 / 2021;
auto lastDay = 31d / 12 / 2021;

for (auto d = firstDay; d <= lastDay; d += std::chrono::months{1}) {
  std::cout << d << ": Party\n";
}
```

Output:

```
2021-01-01: Party
2021-02-01: Party
2021-03-01: Party
2021-04-01: Party
2021-05-01: Party
2021-06-01: Party
2021-07-01: Party
2021-08-01: Party
2021-09-01: Party
2021-10-01: Party
2021-11-01: Party
2021-12-01: Party
```

firstDay is initialized as follows:

```
std::chrono::month / int
=> std::chrono::month_day / int
   => std::chrono::year_month_day
```

Let's Have a Party

C++20

```
#include <iostream>
#include <chrono>
using namespace std::literals;
#include <format>

auto firstDay = std::chrono::January / 1 / 2021;
auto lastDay = 31d / 12 / 2021;

std::cout << firstDay.year() << ":\n";
for (auto d = firstDay; d <= lastDay; d += std::chrono::months{1}) {
  std::cout << std::format(" {:20%a, %B %d:} Party\n", d));
}
```

Output:

```
2021:
 Fri, January 01:      Party
 Mon, February 01:     Party
 Mon, March 01:        Party
 Thu, April 01:        Party
 Sat, May 01:          Party
 Tue, June 01:         Party
 Thu, July 01:         Party
 Sun, August 01:       Party
 Wed, September 01:    Party
 Fri, October 01:      Party
 Mon, November 01:     Party
 Wed, December 01:     Party
```

firstDay is initialized as follows:

```
std::chrono::month / int
=> std::chrono::month_day / int
   => std::chrono::year_month_day
```


Let's Have a Party

C++20

```
#include <iostream>
#include <chrono>
using namespace std::literals;


auto firstDay = 2021y / 1 / 31;
auto lastDay = 31d / 12 / 2021;

for (auto d = firstDay; d <= lastDay; d += std::chrono::months{1}) {
  std::cout << d << ": Party\n";
}
```

Output:

```
2021-01-31: Party
2021-02-31 is not a valid date: Party
2021-03-31: Party
2021-04-31 is not a valid date: Party
2021-05-31: Party
2021-06-31 is not a valid date: Party
2021-07-31: Party
2021-08-31: Party
2021-09-31 is not a valid date: Party
2021-10-31: Party
2021-11-31 is not a valid date: Party
2021-12-31: Party
```

Let's Have a Party

C++20

```
#include <iostream>
#include <chrono>
using namespace std::literals;


auto firstDay = 2021y / 1 / 31;
auto lastDay = 31d / 12 / 2021;

for (auto d = firstDay; d <= lastDay; d += std::chrono::months{1}) {
  if (d.ok()) {
    std::cout << d << ": Party\n";
  }
  else {
    std::cout << d.year() / d.month() / 1 + std::chrono::months{1}
              << ": Party\n";
  }
}
```

Output:

```
2021-01-31: Party
2021-03-01: Party
2021-03-31: Party
2021-05-01: Party
2021-05-31: Party
2021-07-01: Party
2021-07-31: Party
2021-08-31: Party
2021-10-01: Party
2021-10-31: Party
2021-12-01: Party
2021-12-31: Party
```


Let's Have a Party

C++20

```
#include <iostream>
#include <chrono>
using namespace std::literals;


auto firstDay = 2021y / 1 / std::chrono::last;
auto lastDay = 12 / 31d / 2021;

for (auto d = firstDay; d <= lastDay; d += std::chrono::months{1}) {
  std::cout << d << ": Party\n";
}
```

Output:

```
2021/Jan/last: Party
2021/Feb/last: Party
2021/Mar/last: Party
2021/Apr/last: Party
2021/May/last: Party
2021/Jun/last: Party
2021/Jul/last: Party
2021/Aug/last: Party
2021/Sep/last: Party
2021/Oct/last: Party
2021/Nov/last: Party
2021/Dec/last: Party
```

`firstDay` has type `std::chrono::year_month_day_last`

Let's Have a Party

C++20

```
#include <iostream>
#include <chrono>
using namespace std::literals;


auto firstDay = 2021y / 1 / std::chrono::last;
auto lastDay = 12 / 31d / 2021;

for (auto d = firstDay; d <= lastDay; d += std::chrono::months{1}) {
  std::cout << std::format("{:13%B %d}: Party\n", d);
}
```

Output:

```
January 31    : Party
February 28   : Party
March 31      : Party
April 30      : Party
May 31        : Party
June 30       : Party
July 31       : Party
August 31     : Party
September 30  : Party
October 31    : Party
November 30   : Party
December 31   : Party
```


Let's Have a Party

C++20

```
#include <iostream>
#include <chrono>
using namespace std::literals;


auto firstDay = 2021y / 1 / std::chrono::last;
auto lastDay = 12 / 31d / 2021;

for (auto d = firstDay; d <= lastDay; d += std::chrono::months{1}) {
  if (std::chrono::weekday(d) != std::chrono::Sunday) {
    std::cout << std::format("{:%a, %b %d} is Party\n", d);
  }
}
```

Output:

```
Wed, Mar 31 is Party
Fri, Apr 30 is Party
Mon, May 31 is Party
Wed, Jun 30 is Party
Sat, Jul 31 is Party
Tue, Aug 31 is Party
Thu, Sep 30 is Party
Tue, Nov 30 is Party
Fri, Dec 31 is Party
```

Let's Have a Party in Multiple Time Zones

C++20

```
#include <iostream>
#include <chrono>
using namespace std::literals;


auto firstDay = 2021y / 1 / std::chrono::last;
auto lastDay = 12 / 31d / 2021;

for (auto d = firstDay; d <= lastDay; d += std::chrono::months{1}) {
  auto tp{std::chrono::local_days{d} + 18h + 30min};          // @ 18:30 local time
  std::chrono::zoned_time timeLocal{std::chrono::current_zone(), tp};
  std::cout << std::format("{0:13%B %d:} Local {0:%R %Z}\n", timeLocal);

  std::chrono::zoned_time timeLA{"America/Los_Angeles", timeLocal};
  std::cout << std::format("{:>13} LA    {:%R %Z}\n", "", timeLA);
}
```

Possible Output:

```
January 31:   Local 18:30 CET
                 LA 09:30 PST
February 28:  Local 18:30 CET
                 LA 09:30 PST
March 31:     Local 18:30 CEST
                 LA 09:30 PDT
...
September 30: Local 18:30 CEST
                 LA 09:30 PDT
October 31:   Local 18:30 CET
                 LA 10:30 PDT
November 30:  Local 18:30 CET
                 LA 09:30 PST
...
```

Note: Needs *IANA Time Zone DB* support

- Support not required for every platform
  - Visual C++ has limitations (uses ICU time zone support)

Let's Have a Party in Multiple Time Zones

C++20

```
#include <iostream>
#include <chrono>
using namespace std::literals;

try {
  auto firstDay = 2021y / 1 / std::chrono::last;
  auto lastDay = 12 / 31d / 2021;

  for (auto d = firstDay; d <= lastDay; d += std::chrono::months{1}) {
    auto tp{std::chrono::local_days{d} + 18h + 30min};          // @ 18:30 local time
    std::chrono::zoned_time timeLocal{std::chrono::current_zone(), tp};
    std::cout << std::format("{0:13%B %d:} Local {0:%R %Z}\n", timeLocal);

    std::chrono::zoned_time timeLA{"America/Los_Angeles", timeLocal};
    std::cout << std::format("{:>13} LA    {:%R %Z}\n", "", timeLA);
  }
}
catch (const std::exception& e) {
  std::cerr << "EXCEPTION: " << e.what() << '\n';
}
```

Possible Output:

```
January 31:   Local 18:30 CET
                 LA 09:30 PST
February 28:  Local 18:30 CET
                 LA 09:30 PST
March 31:     Local 18:30 CEST
                 LA 09:30 PDT
...
September 30: Local 18:30 CEST
                 LA 09:30 PDT
October 31:   Local 18:30 CET
                 LA 10:30 PDT
November 30:  Local 18:30 CET
                 LA 09:30 PST
...
```

Throws `std::runtime_error` if there is not time zone database

C++20

Calendars and Time Zones in Detail

Chrono Date and Duration Types

C++20

Calendrical Type	Literal Suffix	Output Format
day	d	05
month		Feb
year	y	1999
month_day		Feb/05
month_day_last		Feb/last
year_month		1999/Feb
year_month_day		1999-02-05
year_month_day_last		1999/Feb/last
weekday		Mon
weekday_indexed		Mon[2]
weekday_last		Mon[last]
month_weekday		Feb/Mon[2]
month_weekday_last		Feb/Mon[last]
year_month_weekday		1999/Feb/Mon[2]
year_month_weekday_last		1999/Feb/Mon[last]

Duration Type	Literal Suffix	Output Format
days		1d
weeks		1[604800]s
months		1[2629746]s
years		1[31556952]s
hours	h	1h
minutes	min	1min
seconds	s	1s
milliseconds	ms	1ms
microseconds	us	1µs or 1us
nanoseconds	ns	1ns

365.2425 days

Format Specifiers for Chrono Types

C++20

Spec.	Example	Meaning
%c	Sun Jun 9 17:33:16 2019	Date and time representation
Dates:		
%x	06/09/19	Date representation
%F	2019-06-09	*year-month-day* with four and two digits
%D	06/09/19	*month/day/year* with two digits
%e	9	Day with leading space if single digit
%d	09	Day as two digits
%b	Jun	Abbreviated month name
%h	Jun	ditto
%B	June	Standard or locale's month name
%m	06	Month with two digits
%Y	2019	Year with four digits
%y	19	Year without century as two digits
%G	2019	ISO-week-based year as four digits
%g	19	ISO-week-based year as two digits
%C	20	Century as two digits
Weekdays and weeks:		
%a	Sun	Abbreviated weekday name
%A	Sunday	Weekday name
%w	0	Weekday number (0 - 6 for Sunday - Saturday)
%u	7	Weekday number (1 – 7 for Monday - Sunday)
%W	22	Week of year (week 01 has first Monday)
%U	23	Week of year (week 01 has first Sunday)
%V	23	ISO week of year (week 01 has Jan. 4th)

Spec.	Example	Meaning
Times:		
%X	17:33:16	Time representation
%r	05:33:16 PM	12-hour clock time
%T	17:33:16.850	*hour:minutes:seconds* (+subsec.)
%R	17:33	*hour:minutes* with two digits
%H	17	24-hour as two digits
%I	05	12-hour as two digits
%p	PM	AM or PM
%M	33	Minute with two digits
%S	16.850	Seconds (+subsec.) as dec. value
Other:		
%Z	CEST	Timezone abbreviation
%z	+0200	Offset from UTC
%Oz	+02:00	Offset from UTC with :
%Ez	+02:00	ditto
%j	160	Day of the year (Jan. 1st is 0)
%q	ms	Unit suffix
%Q	63196850	Duration value
%n	\n	Newline character
%t	\t	Tabulator character
%%	%	% character

Calendrical Types vs. Canonical Timepoint Types

C++20

```
#include <iostream>
#include <chrono>
using namespace std::literals;
namespace chr = std::chrono;

auto first = 2021y / 1 / 1;
auto last = 2022y / 1 / 1;

for (chr::year_month_day d = first; d < last; d += chr::months{1}) // OK
{
  std::cout << d << '\n';
}

// calendrical types do not support direct day arithmetic (not the best performance):
for (chr::year_month_day d = first; d < last; d += chr::days{1})   // ERROR

// use the canonical calendar (system days since 1970-1-1):
for (chr::sys_days d = first; d < last; d += chr::days{1})         // OK
{
  std::cout << d << '\n';
}

// canonical types do not support direct month/year arithmetic (not the best performance):
for (chr::sys_days d = first; d < last; d += chr::months{1})       // ERROR
```

Output:

```
2021-01-01
2021-02-01
2021-03-01
...
2021-01-01
2021-01-02
...
2021-01-31
2021-02-01
...
```

Calendrical date type
(data structure with `year`, `month`, `day`)

Canonical timepoint type
(`time_point<system_clock, days>`)

Calendrical Types vs. Canonical Timepoint Types

C++20

```
#include <iostream>
#include <chrono>
using namespace std::literals;

auto begDay = 2021y / 1 / std::chrono::Monday[std::chrono::last];
```

Output:

```
2021/Jan/Mon[last]
2021/Feb/Mon[last]
2021/Mar/Mon[last]
...
2021/Dec/Mon[last]

Mon, Jan 25
Mon, Feb 22
Mon, Mar 22
...
Mon, Nov 01
Mon, Nov 29
Mon, Dec 27
```

```
// Compile-time ERRORS:
// - Can't compare year_month_weekday_last with year_month_weekday_last:
auto endDay = 2022y / 1 / std::chrono::Monday[std::chrono::last];
for (auto d = begDay; d < endDay; d += std::chrono::months{1}) ...

// - Can't compare sys_days with same year_month_weekday_last:
for (std::chrono::sys_days d = begDay; d < endDay; d += std::chrono::months{1}) ...

// - Can't add months/days to sys_days:
for (std::chrono::sys_days d = begDay; d < 2022y/1/1; d += std::chrono::months{1}) ...

// - Can't add weeks/days to year_month_weekday_last:
for (auto d = begDay; d.year() == begDay.year(); d += std::chrono::weeks{4}) ...
```

```
for (auto d = begDay; d.year() == begDay.year(); d += std::chrono::months{1}) {      // OK
  std::cout << d << '\n';
}

for (std::chrono::sys_days d = begDay; d < 2022y/1/1; d += std::chrono::weeks{4}) { // OK
  std::cout << std::format("{:%a, %b %d}\n", d);
}
```


Dealing with Date and Time

C++20

```
try {
  // NOW as universal timepoint:
  auto now = std::chrono::system_clock::now();                // type time_point<>
  std::cout << "now:      " << now << '\n';                   // 2023-09-22 14:40:41.640617000
  std::cout << std::format("{:%A, %D at %R}\n", now);         // Friday, 09/22/23 at 14:40

  // derive calendar date from now:
  std::chrono::year_month_day today{std::chrono::floor<std::chrono::days>(now)};
  std::cout << "today:    " << today << '\n';                 // 2023-09-22

  // derive hours, minutes, (sub)seconds from now (using duration from midnight):
  std::chrono::hh_mm_ss time{now - std::chrono::floor<std::chrono::days>(now)};
  std::cout << "time:     " << time << '\n';                  // 14:40:41.640617000
  std::cout << "hour:     " << time.hours() << '\n';          // 14h

  // using the timezone of Sydney:
  auto tz = std::chrono::locate_zone("Australia/Sydney");     // type const time_zone*
  std::cout << tz->name() << ":\n";                           // Australia/Sydney
  auto tzNow = tz->to_local(now);                             // type time_point<>
  std::cout << " tzNow:  " << tzNow << '\n';                  // 2023-09-23 00:40:41.640617000
  std::chrono::year_month_day tzDay{std::chrono::floor<std::chrono::days>(tzNow)};
  std::cout << " tzDay:  " << tzDay << '\n';                  // 2023-09-23
  std::chrono::hh_mm_ss tzTime{tzNow - std::chrono::floor<std::chrono::days>(tzNow)};
  std::cout << " tzTime: " << tzTime << '\n';                 // 00:40:41.640617000
}
catch (const std::exception& e) {
  std::cerr << "EXCEPTION: " << e.what() << '\n';
}
```

Dealing with Times

C++20

- Use `std::chrono::hh_mm_ss` to deal with time attributes
  - Initialize with a duration
  - Use members `hours()`, `minutes()`, `seconds()`, `subseconds()`
- `std::format()` provides corresponding specifiers

```
auto d = 0.5 * std::chrono::days{1};
std::cout << d << '\n';
std::cout << std::chrono::hh_mm_ss{d} << '\n';
std::cout << std::chrono::hh_mm_ss{d}.hours() << '\n';
std::cout << std::format("{:%T}\n", d);

auto secs = 10000s;
std::cout << secs << '\n';
std::cout << std::chrono::hh_mm_ss{secs} << '\n';

std::chrono::duration<int, std::ratio<1,3>> third{1};
std::cout << third << '\n';
std::cout << std::chrono::hh_mm_ss{third} << '\n';
std::cout << std::chrono::hh_mm_ss{third}.subseconds() << '\n';
std::cout << std::format("{:%R %S}\n", third);
```

Output:

```
0.5d
12:00:00
12h
12:00:00

10000s
02:46:40

1[1/3]s
00:00:00.333333
333333us
00:00 00.333333
```


C++20: Output Formats and Invalid Dates

C++20

```
auto ymd = std::chrono::year{2021}/1/31;
auto dmy = std::chrono::day{31}/1/2021;
auto mdy = std::chrono::January/31/2021;
std::cout << "ymd: " << ymd << '\n';        // "2021-01-31"
std::cout << "dmy: " << dmy << '\n';        // "2021-01-31"
std::cout << "mdy: " << mdy << '\n';        // "2021-01-31"

auto yml = std::chrono::year{2021}/1/std::chrono::last;
std::cout << "yml: " << yml << '\n';        // "2021/Jan/last"

auto ymd2 = std::chrono::year{2021}/1/31;
ymd2 += std::chrono::months{1};
std::cout << "ymd2: " << ymd2 << '\n';      // "2021-02-31 is not a valid date"
std::cout << std::format("ymd: {:%F}", ymd2) << '\n';          // "2021-02-31"
std::cout << std::format("ymd: {:%Y-%m-%d}", ymd2) << '\n';    // "2021-02-31"

if (!ymd2.ok()) {
  ymd2 = ymd2.year()/ymd2.month()/chr::last;          // round down to last of month
  std::cout << "ymd2a: " << ymd2 << '\n';             // "2021-02-28"
  ymd2 = ymd2.year()/ymd2.month()/1 + chr::months{1}; // round up to next 1st
  std::cout << "ymd2b: " << ymd2 << '\n';             // "2021-03-01"
}
```

All: `std::chrono::year_month_day`

only output format with `-` instead of `/` as separator

Chrono Clocks C++11 and C++20

C++11 / C++20

- **system_clock**
  - Maps to clock of the system (epoch usually at Jan 1, 1970)
    - Supports conversions to/from `time_t`
    - May be adjusted
  - All minutes have 60 seconds (a leap second is part of the previous second)
- **steady_clock**
  - Should not be adjusted (ideal for duration measurements and relative timers)

- **utc_clock** since C++20
  - Clock based on UTC/GMT (epoch at Jan 1, 1970)
  - Minutes might have 61 seconds (e.g. last minute of June 30, 1972)
- **gps_clock**
  - Clock based on the Global Positioning System (epoch at Jan 6, 1970)
  - All minutes have 60 seconds, but next minute might start earlier (currently, 18 seconds ahead)
- **tai_clock**
  - Clock based on the International Atomic Time (epoch at Jan 1, 1958)
  - All minutes have 60 seconds, but next minute might start earlier (currently, 37 seconds ahead)
- **file_clock**
  - for filesystem timepoints
- **local_t**
  - Pseudo clock for *local time*

C++20: Timepoints of Chrono Clocks

C++20

```
using namespace std::literals;
namespace chr = std::chrono;

auto tpSys = chr::sys_days{2017y/1/1} - 1000ms;       // last second of 2016
auto tpUtc = chr::clock_cast<chr::utc_clock>(tpSys); // we need a clock with leap seconds
for (auto tp = tpUtc; tp < tpUtc + 2500ms; tp += 200ms) {
  std::cout << std::format("{:%F %T} SYS ", chr::clock_cast<chr::system_clock>(tp));
  std::cout << std::format("{:%F %T %Z}  ", chr::clock_cast<chr::utc_clock>(tp));
  std::cout << std::format("{:%F %T %Z}  ", chr::clock_cast<chr::gps_clock>(tp));
  std::cout << std::format("{:%F %T %Z}\n", chr::clock_cast<chr::tai_clock>(tp));
}
```

Output:

```
2016-12-31 23:59:59.000 SYS 2016-12-31 23:59:59.000 UTC 2017-01-01 00:00:16.000 GPS 2017-01-01 00:00:35.000 TAI
2016-12-31 23:59:59.200 SYS 2016-12-31 23:59:59.200 UTC 2017-01-01 00:00:16.200 GPS 2017-01-01 00:00:35.200 TAI
2016-12-31 23:59:59.400 SYS 2016-12-31 23:59:59.400 UTC 2017-01-01 00:00:16.400 GPS 2017-01-01 00:00:35.400 TAI
2016-12-31 23:59:59.600 SYS 2016-12-31 23:59:59.600 UTC 2017-01-01 00:00:16.600 GPS 2017-01-01 00:00:35.600 TAI
2016-12-31 23:59:59.800 SYS 2016-12-31 23:59:59.800 UTC 2017-01-01 00:00:16.800 GPS 2017-01-01 00:00:35.800 TAI
2016-12-31 23:59:59.999 SYS 2016-12-31 23:59:60.000 UTC 2017-01-01 00:00:17.000 GPS 2017-01-01 00:00:36.000 TAI
2016-12-31 23:59:59.999 SYS 2016-12-31 23:59:60.200 UTC 2017-01-01 00:00:17.200 GPS 2017-01-01 00:00:36.200 TAI
2016-12-31 23:59:59.999 SYS 2016-12-31 23:59:60.400 UTC 2017-01-01 00:00:17.400 GPS 2017-01-01 00:00:36.400 TAI
2016-12-31 23:59:59.999 SYS 2016-12-31 23:59:60.600 UTC 2017-01-01 00:00:17.600 GPS 2017-01-01 00:00:36.600 TAI
2016-12-31 23:59:59.999 SYS 2016-12-31 23:59:60.800 UTC 2017-01-01 00:00:17.800 GPS 2017-01-01 00:00:36.800 TAI
2017-01-01 00:00:00.000 SYS 2017-01-01 00:00:00.000 UTC 2017-01-01 00:00:18.000 GPS 2017-01-01 00:00:37.000 TAI
2017-01-01 00:00:00.200 SYS 2017-01-01 00:00:00.200 UTC 2017-01-01 00:00:18.200 GPS 2017-01-01 00:00:37.200 TAI
2017-01-01 00:00:00.400 SYS 2017-01-01 00:00:00.400 UTC 2017-01-01 00:00:18.400 GPS 2017-01-01 00:00:37.400 TAI
```

C++20: Chrono Clocks and Seconds

C++20

```
template<typename ClockT>
void printSeconds(int year)
{
  namespace chr = std::chrono;
  chr::sys_days sysBeg = chr::year{year} / 1 / 1;
  chr::sys_days sysEnd = chr::year{year+1} / 1 / 1;
  auto beg = chr::clock_cast<ClockT>(sysBeg);
  auto end = chr::clock_cast<ClockT>(sysEnd);
  auto diff = chr::duration_cast<chr::seconds>(end - beg);
  std::cout << "  " << diff.count() << " seconds\n";
}
```

```
for (int y : {1900, 1972, 1982, 2016, 2020}) {
  std::cout << y << ":\n";
  printSeconds<chr::system_clock>(y);
  printSeconds<chr::utc_clock>(y);
  printSeconds<chr::gps_clock>(y);
  printSeconds<chr::tai_clock>(y);
}
```

Output:

```
1900:
  31536000 seconds
  31536000 seconds
  31536000 seconds
  31536000 seconds
1972:
  31622400 seconds
  31622402 seconds
  31622402 seconds
  31622402 seconds
1982:
  31536000 seconds
  31536001 seconds
  31536001 seconds
  31536001 seconds
2016:
  31622400 seconds
  31622401 seconds
  31622401 seconds
  31622401 seconds
2020:
  31622400 seconds
  31622400 seconds
  31622400 seconds
  31622400 seconds
```


C++20

std::jthread and Stop Tokens

Starting a Thread

C++11

- `std::async()` and `std::future<>`
  - **High-level API**
  - Function that requests to start a function/lambda in its own thread
    - Returns a handle to yield the outcome (return value or exception)
    - Request may be deferred until outcome is needed

- `std::thread`
  - **Low-level API**
  - Type of object that start a function/lambda in its own thread
    - Constructor with ***callable*** starts the thread
    - Has to `join()` for (or `detach()`) the thread before destructor is called
  - Threads can't return anything
  - Threads are not allowed to end with an exception

Use `std::jthread` since C++20

Using Class `std::thread`

C++11

```
#include <thread>

void foo()
{
  std::string name = "Kim";
  ...
  // start thread calling task() with two args:
  std::thread t{task, name, 42};
  ...
}
```

```
void task(const std::string& s,
          int val)
{
  ...
}
```

`s` refers to a **copy** of `name` (even as a reference)

Using Class `std::thread` and `std::cref()`

C++11

```
#include <thread>

void foo()
{
  std::string name = "Kim";
  …
  // start thread calling task() with two args:
  std::thread t{task, std::cref(name), 42};
  …
}
```

```
void task(const std::string& s,
          int val)
{
  …
}
```

s refers to name (as a reference)

Using Class `std::thread`

C++11

```
#include <thread>

void foo()
{
  std::string name = "Kim";
  …
  // start thread calling task() with two args:
  std::thread t{task, name, val};
  …
}
```

```
void task(const std::string& s,
          int val)
{
  …
}
```

On thread end with exception std::terminate() => std::abort()

Using Class `std::thread`

```
#include <thread>

void foo()
{
  std::string name = "Kim";
  ...
  // start thread calling task() with two args:
  std::thread t{task, name, val};
  ...




}
```

```
void task(const std::string& s,
          int val)
{
  try {
    ...
  }
  catch (...) {
    ...   // handle every exception
  }
}
```


Using Class `std::thread` C++11

```
#include <thread>

void foo()
{
  std::string name = "Kim";
  ...
  // start thread calling task() with two args:
  std::thread t{task, name, val};
  ...




}
```

There is a bug
(unless core dumps don't matter)

```
void task(const std::string& s,
          int val)
{
  try {
    ...
  }
  catch (...) {
    ...   // handle every exception
  }
}
```

Destructor of a `std::thread` will std::terminate() => std::abort() unless `join()` or `detach()` was called

Using Class `std::thread`

C++11

```
#include <thread>

void foo()
{
  std::string name = "Kim";
  ...
  // start thread calling task() with two args:
  std::thread t{task, name, val};
  ...

  // wait for thread to finish:
  t.join();
  ...
}
```

There is still a bug
(unless core dumps don't matter)

On exception, destructor will still std::terminate() => core dump because `join()` was not called

```
ask(const std::string& s,
    int val)
{
  try {
    ...
  }
  catch (...) {
    ...  // handle every exception
  }
}
```


Using Class `std::thread`

C++11

```
#include <thread>

void foo()
{
  std::string name = "Kim";
  ...
  // start thread calling task() with two args:
  std::thread t{task, name, val};
  try {
    ...
  }
  catch (...) {  // on exception
    ...
    t.join();    // wait for thread
    throw;       // rethrow exception
  }
  // wait for thread to finish:
  t.join();
  ...
}
```

```
void task(const std::string& s,
          int val)
{
  try {
    ...
  }
  catch (...) {
    ...  // handle every exception
  }
}
```

Using Class `std::thread` for Two Threads

C++11

```
...
std::thread t1{task1, name, val};

try {
  std::thread t2{task2, name, val};
  ...
}
catch (...) {   // on exception
  ...           // - signal stop



  t1.join();    // - wait for end of task1
  throw;
}
t1.join();
t2.join();      // ERROR: t2 is out of scope
...
```

Problem with two threads:
- Has to start `t2` in `try` scope because that might throw
- But need to join `t2` in same scope as `t1`

Using Class `std::thread` for Two Threads

C++11

```
...
std::thread t1{task1, name, val};
std::thread t2;
try {
  t2 = std::thread{task2, name, val};
  ...
}
catch (...) {   // on exception
  ...           // - signal stop
  if (t2.joinable()) { // if started
    t2.join(); // - wait for end of task2
  }
  t1.join();    // - wait for end of task1
  throw;
}
t1.join();
t2.join();      // OK
...
```

Using Class `std::thread` for Two Threads

C++11/C++20

```
...
std::thread t1{task1, name, val};
std::thread t2;
try {
  t2 = std::thread{task2, name, val};
  ...
}
catch (...) {   // on exception
  ...           // - signal stop
  if (t2.joinable()) {  // if started
    t2.join();  // - wait for end of task2
  }
  t1.join();    // - wait for end of task1
  throw;
}
t1.join();
t2.join();      // OK
...
```

```
// since C++20 just:
...
std::jthread t1{task1, name, val};
std::jthread t2{task2, name, val};
...
t1.join();
t2.join();
...
```

Prefer `std::jthread` over `std::thread` (same header, same API)

Using Class `std::jthread`

C++20

```
...
std::jthread t1{task1, name, val};
std::jthread t2{task2, name, val};
...
t1.join();
t2.join();
...
```

On exception:

```
if (t2.joinable()) {
  t2.request_stop();
  t2.join();
}
```
Destructor t2

```
if (t1.joinable()) {
  t1.request_stop();
  t1.join();
}
```
Destructor t1

```
void task1(const std::string& n, int val)
{
  try {
    ...     // stop request ignored
  }
  catch (...) {
    ...     // handle every exception
  }
}
```

Using Class `std::jthread` and Stop Tokens

C++20

```
...
std::jthread t1{task1, name, val};
std::jthread t2{task2, name, val};
...
```

On exception:

```
if (t2.joinable()) {
  t2.request_stop();
  t2.join();
}
```
Destructor t2

```
if (t1.joinable()) {
  t1.request_stop();
  t1.join();
}
```
Destructor t1

```
t1.join();
t2.join();
...
```

optional 1st parameter

```
void task1(std::stop_token st,
           const std::string& n, int val)
{
  try {
    ...
    if (st.stop_requested()) {
      return;
    }
    ...
    while (!st.stop_requested()) {
      ...
    }
  }
  catch (...) {
    ...     // handle every exception
  }
}
```


Using Class `std::jthread` and Stop Tokens

C++20

```
...
std::jthread t1{task1, name, val};
std::jthread t2{task2, name, val};
...



if (noLongerNeedTask1) {
  // signal first thread to stop:
  t1.request_stop();
}
...

t1.join();
t2.join();
...
```

optional 1st parameter

```
void task1(std::stop_token st,
           const std::string& n, int val)
{
  try {
    ...
    if (st.stop_requested()) {
      return;
    }
    ...
    while (!st.stop_requested()) {
      ...
    }
  }
  catch (...) {
    ...     // handle every exception
  }
}
```

Using Class `std::jthread` and Stop Callbacks

C++20

```
...
std::jthread t1{task1, name, val};
std::jthread t2{task2, name, val};
...



if (noLongerNeedTask1) {
  // signal first thread to stop:
  t1.request_stop();
}
...


t1.join();
t2.join();
...
```

optional 1st parameter

```
void task1(std::stop_token st,
           const std::string& n, int val)
{
  try {
    ...
    // register callback to be called on stop request:
    std::stop_callback cb{st,
                          []{ ... }};
    ...
    while (!st.stop_requested())
      ...
    }
  }
  catch (...) {
    ...     // handle every exception
  }
}
```

callback called within the thread requesting the stop

Stopping Multiple `std::jthreads`

C++20

```
...
std::jthread t1{task1, name, val};
std::jthread t2{task2, name, val};
try {
  ...
}
catch (...) {
  // stop all threads before joining:
  t2.request_stop();
  t1.request_stop();
  throw; // rethrow exception
}
t1.join();
t2.join();
...
```

With multiple threads, on exception, better stop all threads before first `join()`

optional 1st parameter

```
void task1(std::stop_token st,
           const std::string& n, int val)
{
  try {
    ...
    // register callback to be called on stop request:
    std::stop_callback cb{st,
                          []{ ... }};
    ...
    while (!st.stop_requested()) {
      ...
    }
  }
  catch (...) {
    ...     // handle every exception
  }
}
```

Stop Tokens and Condition Variables

C++20

```
...
std::jthread t1{task1, name, val};
std::jthread t2{task2, name, val};
...
...
t1.join();
t2.join();
...
```

```
void task1(std::stop_token st,
           const std::string& n, int val)
{
  try {
    ...
    // interruptible wait:
    std::unique_lock lg{readyMx};
    if (!readyCV.wait(lg, st, ...)) {
      // stop requested:
      return;
    }
    ...
  }
  catch (...) {
    ...     // handle every exception
  }
}
```

Condition variable `readyCV` has to have type `std::condition_variable_any`

Definition of `std::thread` and `std::jthread`

C++11/C++20

```
namespace std {
 class jthread {
  public:
   template <typename F, typename... Types>
     explicit jthread(F&& f, Types&&... args); // start thread calling f() with copies of args
   jthread() noexcept;                         // initialize with no attached thread (yet)
   ~jthread();                                 // jthread: signal stop and wait if joinable

   bool joinable() const noexcept;             // thread attached (no join() or detach() yet) ?
   void join();                                // wait for the end of the thread
   void detach();                              // no longer refer to the thread (runs in background)

   class id;                                   // type of the actual thread ID
   id get_id() const noexcept;                 // - note: IDs of terminated threads may be reused

   ...  // stop token API for jthread: request_stop(),get_stop_source(),get_stop_token()

   jthread(const jthread&) = delete;
   jthread(jthread&&) noexcept;
   jthread& operator=(const jthread&) = delete;
   jthread& operator=(jthread&&) noexcept;
   void swap(jthread&) noexcept;

   using native_handle_type = implementation-defined;
   native_handle_type native_handle(); // interface to non-portable operations
 };
}
```

- **Move-only type**
- **Non-portable API for**
  - Policies
  - Priorities
  - ...

C++20: Request for a General Cancellation Facility

C++20

- **Basic types:**
  - `std::stop_source`
    - to request a stop
  - `std::stop_token`
    - to react on a requested stop (checking state from time to time)
  - `std::stop_callback`
    - to react on a requested stop (immediately in a registered function)
- **Header file `<stop_token>`**
- **Could be used by synchronous or asynchronous operations**
  - Threads, coroutines, executors, networking, I/O, ...
  - For example:
    - Cancelling/interrupting a timed async schedule operation on an executor
    - Cancelling/interrupting a lock operation on an async_mutex
    - Cancelling/interrupting async I/O operations

Using Stop Tokens (since C++20)

C++20

```
#include <stop_token>
...

// create stop_source and stop_token:
std::stop_source ssrc;
std::stop_token stok{ssrc.get_token()};

// register callbacks:
std::stop_callback scb1{stok, []{
  std::cout << " 1st stop cb\n";
}};

std::stop_callback scb2{stok, []{
  std::cout << " 2nd stop cb\n";
}};

process(stok);

// call in the background other function:
callThread(func, ssrc);
...

std::stop_callback{stok, []{
  std::cout << "last cb\n";
}};
```

Callback immediately called if stop already requested

```
void process(std::stop_token st)
{
  // register temporary callback:
  std::stop_callback scb{st, []{
    std::cout << "process() cb\n";
  }};
  std::cout << "run process()\n";
  ...
  std::cout << "end process()\n";
} // unregisters callback
```

```
void func(std::stop_source stopper)
{
  ...
  std::cout << "request stop\n";
  stopper.request_stop();
  ...
}
```

- Calls callbacks in arbitrary order
- Callbacks called by/in ***this*** thread

Possible Output:

```
run process()
end process()
request stop
 2nd stop cb
 1st stop cb
last cb
```

C++20: Stop Tokens and Multiple Threads in Practice

C++20

```
...
// start two threads:
std::jthread t1{task1, name, val};
std::jthread t2{task2, name, val};
try {
  ...
}
catch(...) {
  // request to stop all threads before starting joins:
  t1.request_stop();
  t2.request_stop();
  throw;   // rethrow exception
}
t1.join();
t2.join();
...
```

```
void task1(std::stop_token st,
           std::string s, int v)
{
  ...
}

void task2(std::stop_token st,
           std::string s, int v)
{
  ...
}
```


C++20: Stop Tokens and Multiple Threads in Practice

C++20

```
...
// start two threads, stopping t1 stops t2:
std::jthread t1{task1, name, val};
std::jthread t2{task2, t1.get_stop_token(),
                       name, val};
try {
  ...
}
catch(...) {
  // request to stop all threads before starting joins:
  t1.request_stop();
  throw;   // rethrow exception
}
t1.join();
t2.join();
...
```

```
void task1(std::stop_token st,
           std::string s, int v)
{
  ...
}

void task2(std::stop_token st,
           std::string s, int v)
{
  ...
}
```

C++20: Stop Tokens and Multiple Threads in Practice

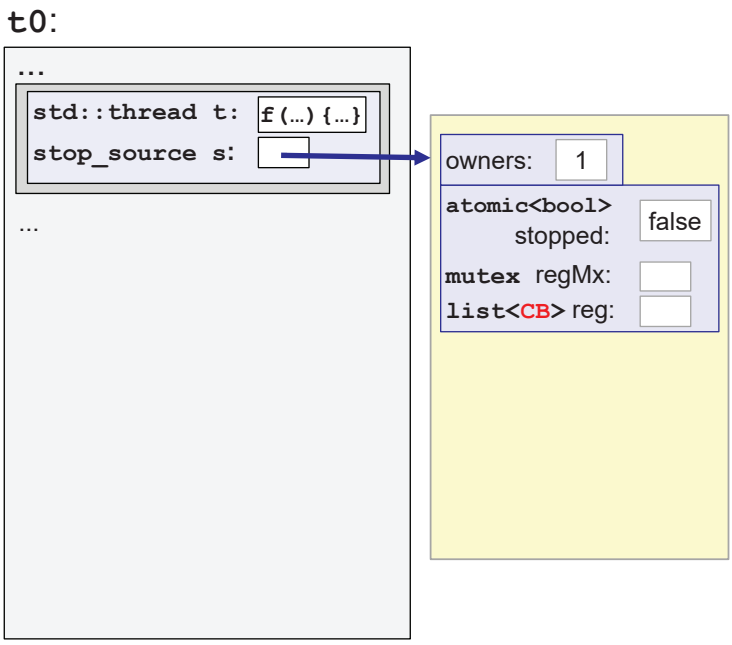
C++20

```
...
// init stop source that two stop multiple threads:
std::stop_source ssrc;
std::stop_token stok{ssrc.get_token()};
// start two threads that can be stopped by ssrc:
std::jthread t1{task1, stok, name, val};
std::jthread t2{task2, stok, name, val};
try {
  ...
}
catch(...) {
  // request to stop all threads before starting joins:
  ssrc.request_stop();
  throw;   // rethrow exception
}
t1.join();
t2.join();
...
```

```
void task1(std::stop_token st,
           std::string s, int v)
{
  ...
}

void task2(std::stop_token st,
           std::string s, int v)
{
  ...
}
```

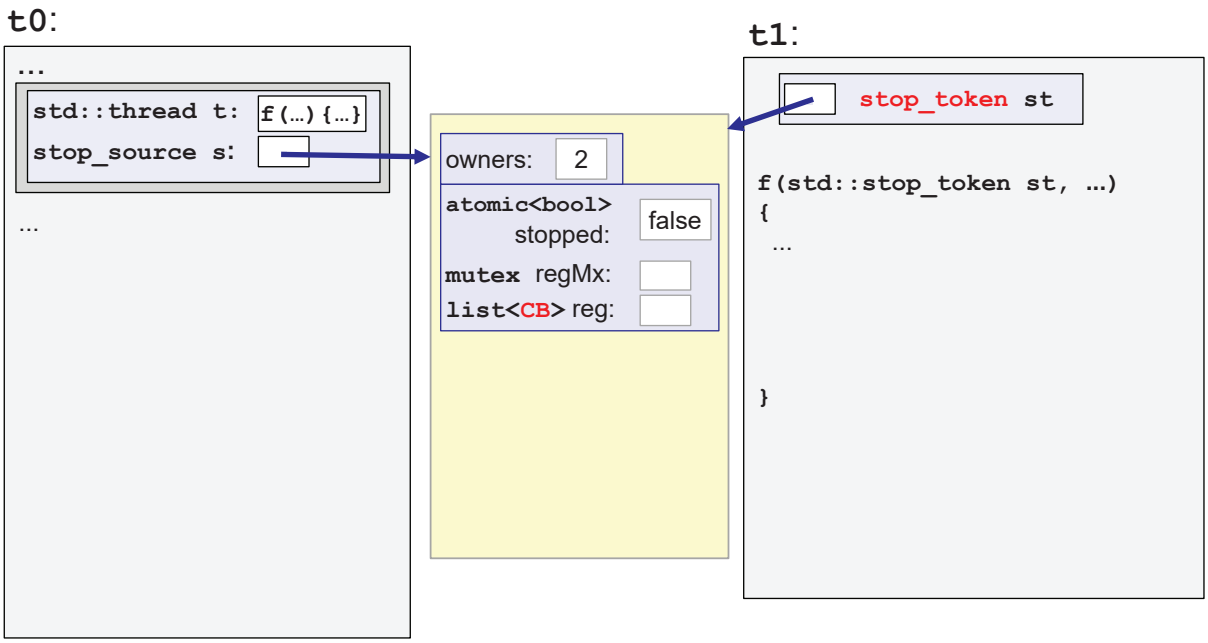

Using Stop Callbacks

C++20



Using Stop Callbacks

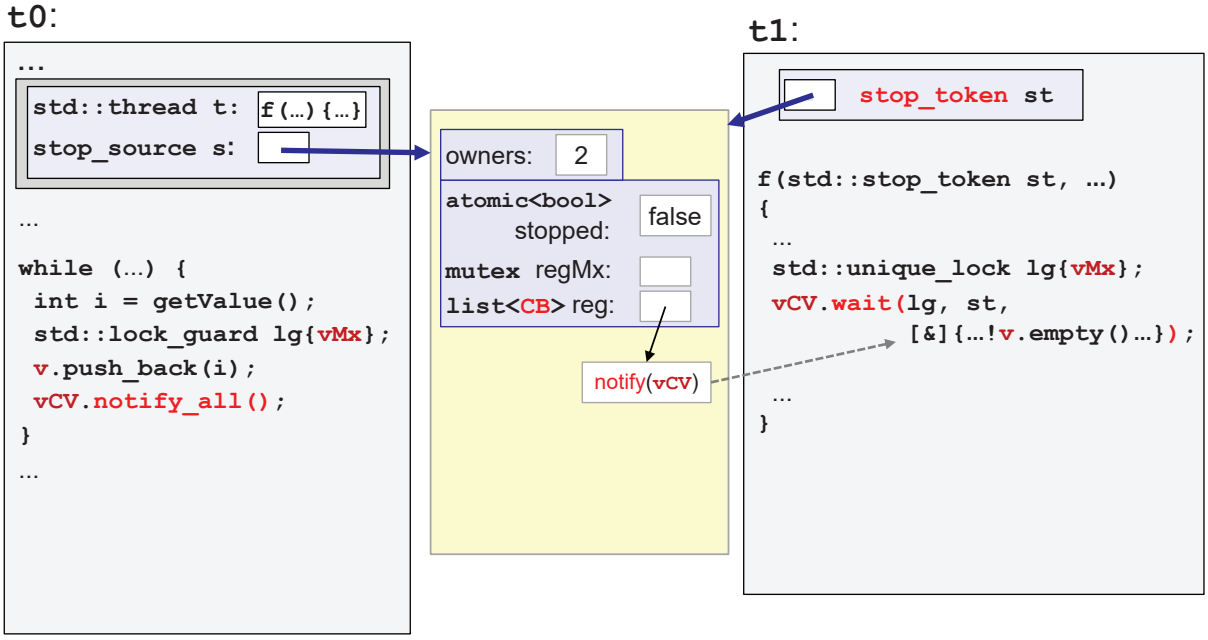
C++20



Using Stop Callbacks with Condition Variables

C++20

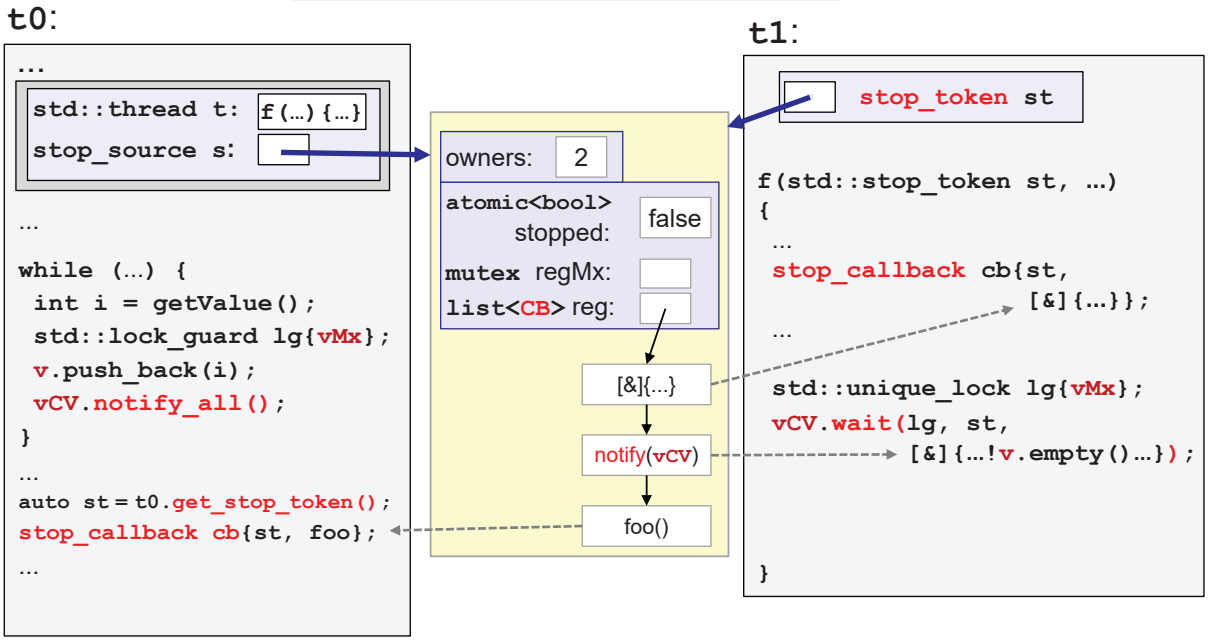
```
std::vector<int>          v;
std::mutex                vMx;
std::condition_variable   vCV;
```



Using Stop Callbacks with Condition Variables

C++20

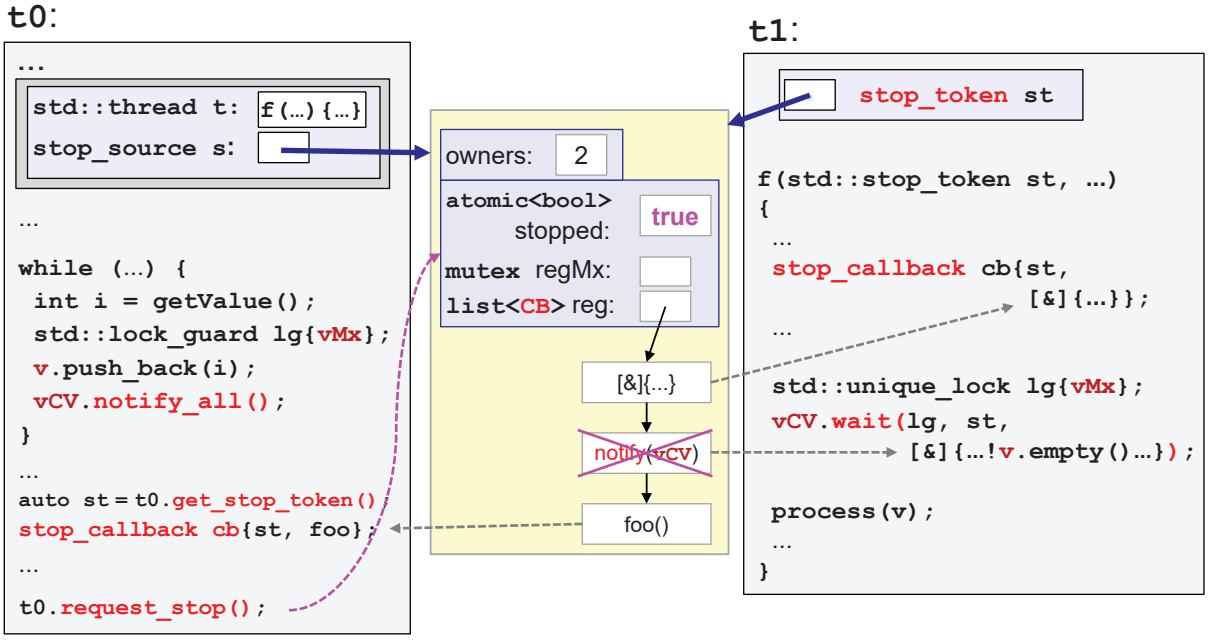
```
std::vector<int>          v;
std::mutex                vMx;
std::condition_variable   vCV;
```



Using Stop Callbacks with Condition Variables

C++20

```
std::vector<int>          v;
std::mutex                vMx;
std::condition_variable   vCV;
```

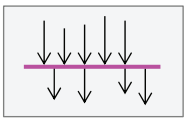


C++20

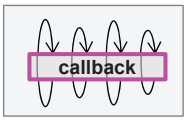
Concurrency Features

C++20 Thread Synchronization Mechanisms

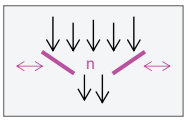
- **Latches**
  - Single-time synchronization points
    - Multiple threads can wait once for multiple threads
    - Multiple threads can wait to be called if all done



- **Barriers**
  - Multiple-time synchronization points
    - Multiple threads can loop to signal done and wait for others
    - Can call callback each time all threads are done



- **Semaphores**
  - Lightweight synchronization primitives
  - Limits and sync multiple resources for multiple threads
  - Two types: counting or binary

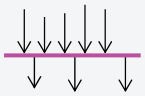


- **Output Sync Streams**
  - Synchronize output of different threads

C++20: std::latch

C++20

One-time thread synchronization

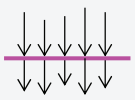


```
#include <latch>
...

std::array vals{'.', '?', '8', '-', '+'};   // values to process in multiple threads
std::latch allLoopsDone{vals.size()};       // init countdown (to wait until all loops are done)

std::vector<std::jthread> tasks;
for (char c : vals) {
  tasks.push_back(std::jthread{[c, &allLoopsDone]{        // for each value start thread
    for (int i = 0; i < c; ++i) {
      std::cout.put(c).flush();
      std::this_thread::sleep_for(100ms);
    }
    allLoopsDone.count_down();   // signal loop is done
    ...
  }});
}

std::cout << "waiting until all tasks are done" << std::endl;
allLoopsDone.wait();                        // wait until all loops done (countdown is 0)
std::cout << "all loops done" << std::endl; // note: threads might still run
```


C++20: std::latch

C++20

One-time thread synchronization



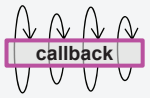
```
#include <latch>
...

std::array vals{'.', '?', '8', '-', '+'};   // values we have to loop over in a thread
std::latch allReady{vals.size()};           // synchronize start of processing of all threads

std::vector<std::jthread> tasks;
for (char c : vals) {
  tasks.push_back(std::jthread{[c, &allReady]{          // start thread for each task
    ...
    allReady.arrive_and_wait();  // sync all threads here
    for (int i = 0; i < c; ++i) {
      std::cout.put(c).flush();
      std::this_thread::sleep_for(100ms);
    }
    ...
  }});
}
...
```

Combination of:
`allReady.count_down();`
`allReady.wait();`

Thanks to Anthony Williams for this example

C++20: std::barrier<> C++20

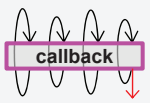
Multi-time thread synchronization




```
...
#include <barrier>

void compute()
{
  std::array vals{1.0, 2.0, 3.0, 4.0, 5.0};   // initial values for square roots
  std::barrier allRdy{vals.size(),             // barrier to synchronize threads
                      [&]()noexcept {          // callback (has to be noexcept)
                        print(vals);           // - print values each time all are processed
                      }};
  std::vector<std::jthread> threads;
  for (std::size_t idx = 0; idx < vals.size(); ++idx) {
    threads.push_back(std::jthread{[idx, &vals, &allRdy] {
                                     while(true) {
                                       vals[idx] = std::sqrt(vals[idx]);
                                       ...
                                       allRdy.arrive_and_wait(); // sync and print()
                                       ...
                                     }
                                   }});
  }
  ...
}
```


C++20: std::barrier<> C++20

Multi-time thread synchronization



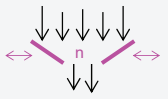

```
...
#include <barrier>

void compute()
{
  std::array vals{1.0, 2.0, 3.0, 4.0, 5.0};   // initial values for square roots
  std::barrier allRdy{vals.size(),             // barrier to synchronize threads
                      [&]()noexcept {          // callback (has to be noexcept)
                        print(vals);           // - print values each time all are processed
                      }};
  std::vector<std::jthread> threads;
  for (std::size_t idx = 0; idx < vals.size(); ++idx) {
    threads.push_back(std::jthread{[idx, &vals, &allRdy](std::stop_token st) {
                                     while(!st.stop_requested()) {
                                       vals[idx] = std::sqrt(vals[idx]);
                                       ...
                                       allRdy.arrive_and_wait(); // sync and print()
                                       ...
                                     }
                                     // ensure others don't wait for this thread anymore:
                                     allRdy.arrive_and_drop();
                                   }});
  }
  ...
} // destructor signals stop to all threads
```

also on exceptions

C++20: std::counting_semaphore<>

C++20

Limit and sync access to multiple resources




```
#include <semaphore>
...

std::queue<int> values;    // queue to read values from
std::mutex valuesMx;
constexpr int numThreads = 10;
std::counting_semaphore<numThreads> maxParallel{0};   // control no of parallel threads
std::vector<std::jthread> threads;
for (int i = 0; i < numThreads; ++i) {
  threads.push_back(std::jthread{[&](std::stop_token st){
    ...
    while(!st.stop_requested()) {
      maxParallel.acquire();    // OK to act?
      ...    // read/select next value
      ...    // process next value
      maxParallel.release();    // done
    }
  }});
}
...
maxParallel.release(3);    // enable 3 (more) threads in parallel
...
maxParallel.release();     // enable 1 (more) thread in parallel
...
```

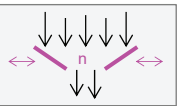
at most 10 threads enabled
initially 0 threads enabled

- **Threads scheduling is not fair**
  - Some threads might wait forever

C++20: std::counting_semaphore<>

C++20

Limit and sync access to multiple resources




```
#include <semaphore>
...

std::queue<int> values;    // queue to read values from
std::mutex valuesMx;
constexpr int numThreads = 10;
std::counting_semaphore<numThreads> maxParallel{0};   // control no of parallel threads
std::vector<std::jthread> threads;
for (int i = 0; i < numThreads; ++i) {
  threads.push_back(std::jthread{[&](std::stop_token st){
    ...
    while(!st.stop_requested()) {
      ...    // read next value
      maxParallel.acquire();    // OK to run?
      ...    // process next value
      maxParallel.release();    // others can run
    }
  }});
}
...
maxParallel.release(3);    // enable 3 (more) threads in
...
maxParallel.release();     // enable 1 (more) thread in p
...
```

at most 10 threads enabled
initially 0 threads enabled

Bad code
- **Thread schedulers are not fair**
  - Other threads might run forever and this value never processed

If parallel threads are limited
- **It should not matter which threads run**
  - Threads should do the same
  - Threads should not perform state-specific pre-processing

Signaling with `std::binary_semaphore`

C++20

```
int inData;                                   // std::counting_semaphore<1>
std::binary_semaphore inReady{0};             // signal there is data to process
std::binary_semaphore doneReady{0};           // signal processing is done

// start threads to read and process values by value:
std::jthread process{[&] (std::stop_token st) {
                       while(!st.stop_requested()) {   // block with timeout to check for stop requests
                         if (inReady.try_acquire_for(1s)) {   // wait for next value
                           int data = inData;                 // read next value
                           ...                                // process it
                           doneReady.release();               // signal processing done
                         }
                       }
                     }};

// provide values to process:
for (int i = 0; i < 10; ++i) {
  inData = i;                 // store next value
  inReady.release();          // signal to start processing
  doneReady.acquire();        // and wait until processing done
  ...
}
```


C++20: Changes/Extensions for Atomic Types

C++20

- **Atomic reference `std::atomic_ref<>`**
  - `std::atomic<>` for references
    - with specializations for raw pointers and integral values
  - provides temporary atomic interface to trivially copyable type
  - E.g., for (re-)initialization without atomic access
- **Atomic shared pointer `std::atomic<std::shared_ptr<>>`**
  - Replaces atomic operations for `std::shared_ptr<>`
  - Type ensures that all value access is atomic
    - instead of each access has to ensure it on its own
- **Specializations of atomics for floating-point types**
  - Adds atomic support to add/subtract values
- **Thread synchronization for all atomic types**
  - Using `wait()`, `notify_one()`, and `notify_all()`

std::atomic_ref<> C++20

```
#include <atomic>
...

// create and initialize an array of integers with the value 100:
std::array<int, 1000> values;
std::fill_n(values.begin(), values.size(), 100);

// start multiple threads concurrently decrementing the value:
std::vector<std::jthread> threads;
for (int i = 0; i < numThreads; ++i) {
  threads.push_back(std::jthread{[&values] (std::stop_token st) {
    while (!st.stop_requested()) {
      int idx = randomIndex();
      // enable atomic access to the value with the index:
      std::atomic_ref val{values[idx]};
      // and use it:
      --val;
      if (val.load() <= 0) {
        std::cout << "index " << idx << " is zero\n";
      }
      ...
    }
  });
}
...
```

no atomic access

atomic access

Atomic Waits and Notifications for Fair Scheduling C++20

Main thread

```
// limit the availability of threads with a fair ticket system:
std::atomic<int> maxTicket{0};   // maximum requested ticket no
std::atomic<int> actTicket{0};   // current allowed max ticket no

...  // start threads

// enable 5 threads (first 5 tickets):
actTicket += 5;
actTicket.notify_all();
```

Processing threads

```
... // set-up processing

// request next ticket:
int myTicket{++maxTicket};

// wait until our ticket is called:
int act = actTicket.load();
while (act < myTicket) {   // is it our turn?
  actTicket.wait(act);     // if not, block and reload
  act = actTicket.load();
}

...  // process

// done, so enable next ticket:
++actTicket;
actTicket.notify_all();
```

Specializations of std::atomic<> C++11

```
template<typename T>
struct atomic
{
 atomic() noexcept = default;
 constexpr atomic(T) noexcept;

 atomic(const atomic&) = delete;
 atomic& operator=(const atomic&)
    = delete;

 void store(T) noexcept;
 T load() const noexcept;
 T operator=(T) noexcept;
 operator T() const noexcept;

 ... // other members
};
```

Primary template

```
template<typename T>
struct atomic<T*>
{
 atomic() noexcept = default;
 constexpr atomic(T*) noexcept;

 atomic(const atomic&) = delete;
 atomic& operator=(const atomic&)
    = delete;

 void store(T*) noexcept;
 T* load() const noexcept;
 T* operator=(T*) noexcept;
 operator T*() const noexcept;

 T* operator++() noexcept;
 T* operator++(int) noexcept;
 T* operator--() noexcept;
 T* operator--(int) noexcept;
 T* operator+=(ptrdiff_t) noexcept;
 T* operator-=(ptrdiff_t) noexcept;

 ... // other members
};
```

Partial specialization for pointers

```
template<>
struct atomic<intT>
{
 atomic() noexcept = default;
 constexpr atomic(intT) noexcept;

 atomic(const atomic&) = delete;
 atomic& operator=(const atomic&)
    = delete;

 void store(intT) noexcept;
 intT load() const noexcept;
 intT operator=(intT) noexcept;
 operator intT() const noexcept;

 intT operator++() noexcept;
 intT operator++(int) noexcept;
 intT operator--() noexcept;
 intT operator--(int) noexcept;
 intT operator+=(intT) noexcept;
 intT operator-=(intT) noexcept;
 intT operator&=(intT) noexcept;
 intT operator|=(intT) noexcept;
 intT operator^=(intT) noexcept;

 ... // other members
};
```

Full specializations for all integral types

Specializations of std::atomic<> C++11 / C++20

```
template<typename T>
struct atomic
{
 atomic() noexcept = default;
 constexpr atomic(T) noexcept;

 atomic(const atomic&) = delete;
 atomic& operator=(const atomic&
    = delete;

 void store(T) noexcept;
 T load() const noexcept;
 T operator=(T) noexcept;
 operator T() const noexcept;

 ... // other members
};
```

Primary template

```
template<typename T>
struct atomic<T*>
{
 atomic() noexcept = default;
 constexpr atomic(T*) noexcept;
```

```
template<>                    C++20
struct atomic<floatT>
{
 atomic() noexcept = default;
 constexpr atomic(floatT) noexcept;

 atomic(const atomic&) = delete;
 atomic& operator=(const atomic&)
    = delete;

 void store(floatT) noexcept;
 floatT load() const noexcept;
 floatT operator=(floatT) noexcept;
 operator floatT() const noexcept;

 floatT operator+=(floatT) noexcept;
 floatT operator-=(floatT) noexcept;

 ... // other members
};
```

Full specializations for all floating-point types

```
template<>
struct atomic<intT>
{
 atomic() noexcept = default;
 constexpr atomic(intT) noexcept;

 atomic(const atomic&) = delete;
 atomic& operator=(const atomic&)
    = delete;

 void store(intT) noexcept;
 intT load() const noexcept;
 intT operator=(intT) noexcept;
 operator intT() const noexcept;

 intT operator++() noexcept;
 intT operator++(int) noexcept;
 intT operator--() noexcept;
 intT operator--(int) noexcept;
 intT operator+=(intT) noexcept;
 intT operator-=(intT) noexcept;
 intT operator&=(intT) noexcept;
 intT operator|=(intT) noexcept;
 intT operator^=(intT) noexcept;

 ... // other members
};
```

Full specializations for all integral types

Concurrent Output to Streams

C++11/C++20

- **Interleaved characters** for standard streams like `std::cout`
- **Undefined behavior** (*data race*) for streams in general

```
#include <iostream>
#include <cmath>
#include <thread>

void squareRoots(int num)
{
  for (int i = 0; i < num; ++i) {
    std::cout << "squareroot of "
              << i << " is "
              << std::sqrt(i) << '\n';
  }
}

int main()
{
  std::jthread t1(squareRoots, 5);
  std::jthread t2(squareRoots, 5);
  std::jthread t3(squareRoots, 5);
  ...
}
```

Undefined behavior if file stream are used

Possible Output:

```
squareroot of squareroot of 0 is 0 is 0
0squareroot of squareroot of
01squareroot of  is  is 101 is

1squareroot of squareroot of
12squareroot of  is  is 21 is 1.41421

1.41421squareroot of squareroot of
23squareroot of  is  is 31.41421 is 1.73205

1.73205squareroot of squareroot of
34squareroot of  is  is 41.73205 is 2

2squareroot of
4 is 2
```

Even with `std::flush` or `std::endl` (maybe less likely, though)

C++20: Synchronized Output Streams

C++20

```
#include <iostream>
#include <cmath>
#include <thread>
#include <syncstream>

void squareRoots(int num)
{
  std::osyncstream coutSync{std::cout};
  for (int i = 0; i < num; ++i) {
    coutSync << "squareroot of "
             << i << " is "
             << std::sqrt(i) << '\n'
             << std::flush_emit;
  }
}

int main()
{
  std::jthread t1(squareRoots, 5);
  std::jthread t2(squareRoots, 5);
  std::jthread t3(squareRoots, 5);
  ...
}
```

OK even for file streams

Possible Output:

```
squareroot of 0 is 0
squareroot of 0 is 0
squareroot of 1 is 1
squareroot of 0 is 0
squareroot of 1 is 1
squareroot of 2 is 1.41421
squareroot of 1 is 1
squareroot of 2 is 1.41421
squareroot of 3 is 1.73205
squareroot of 2 is 1.41421
squareroot of 3 is 1.73205
squareroot of 4 is 2
squareroot of 3 is 1.73205
squareroot of 4 is 2
squareroot of 4 is 2
```

C++20: Synchronized Output Streams

C++20

- **`std::osyncstream`**
  - allows to write concurrently to a stream
  - The output is written
    - with the manipulator **`std::flush_emit`**
    - with the destructor

```
...
#include <syncstream>

void squareRoots(int num)
{
  for (int i = 0; i < num; ++i) {
    std::osyncstream{std::cout} << "squareroot of "
                                << i << " is "
                                << std::sqrt(i) << '\n';
  }
}

std::jthread t1(squareRoots, 5);
std::jthread t2(squareRoots, 5);
std::jthread t3(squareRoots, 5);
```


C++20: Synchronized Output Streams to Files

C++20

```
...
#include <syncstream>

void squareRoots(std::ostream& strm, int num)
{
  for (int i = 0; i < num; ++i) {
    strm << "squareroot of " << i << " is "
         << std::sqrt(i) << '\n'
         << std::flush_emit;
  }
}

int main()
{
  std::ofstream fs{"tmp.out"};
  std::osyncstream syncStrm1{fs};
  std::jthread t1(squareRoots, std::ref(syncStrm1), 5);

  std::osyncstream syncStrm2{fs};
  std::jthread t2(squareRoots, std::ref(syncStrm2), 5);

  std::osyncstream syncStrm3{fs};
  std::jthread t3(squareRoots, std::ref(syncStrm3), 5);
}
```

Possible output to tmp.out:

```
squareroot of 0 is 0
squareroot of 0 is 0
squareroot of 1 is 1
squareroot of 0 is 0
squareroot of 1 is 1
squareroot of 2 is 1.41421
squareroot of 1 is 1
squareroot of 2 is 1.41421
squareroot of 3 is 1.73205
squareroot of 2 is 1.41421
squareroot of 3 is 1.73205
squareroot of 4 is 2
squareroot of 3 is 1.73205
squareroot of 4 is 2
squareroot of 4 is 2
```

Ensure each thread has its own **`osyncstream`**

C++20: Thread-safe "Printing Debugger"

C++20

```
#include <iostream>    // for std::cout
#include <syncstream>  // for std::osyncstream

auto coutSync() {
  return std::osyncstream{std::cout};
}

constexpr bool debug = true;
auto coutDbug() {
  if constexpr (debug) {
    return std::osyncstream{std::cout};
  }
  else {
    struct devnullbuf : public std::streambuf {
      int_type overflow (int_type c) { return c; }
    };
    static devnullbuf devnull;
    return std::ostream{&devnull};
  }
}

coutSync() << "foo(" << s << ") in " << std::this_thread::get_id() << "\n";
coutDbug() << "foo(" << s << ") in " << std::this_thread::get_id() << "\n";
```


C++17: Parallel `for_each()`

C++17

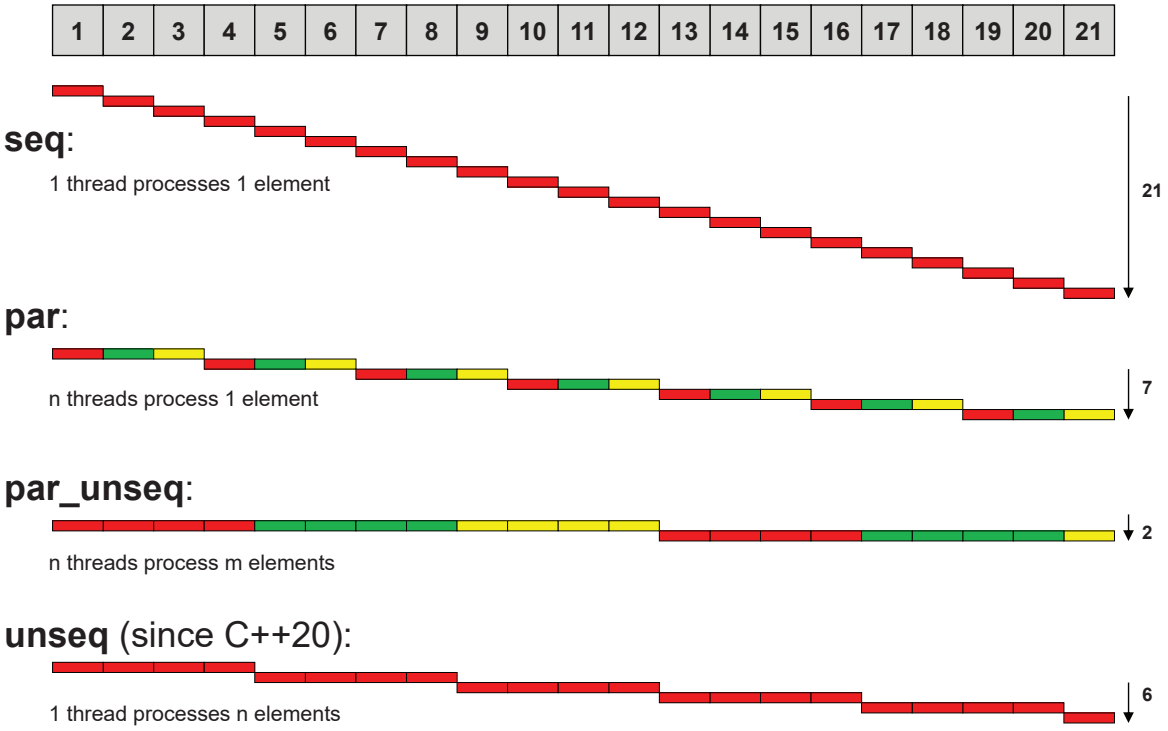
```
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>
#include <execution>

int main (int argc, char** argv)
{
  int numElems = 1000;
  if (argc > 1) {
    numElems = std::atoi(argv[1]);
  }

  // init vector of different double values:
  std::vector<double> coll;
  coll.reserve(numElems);
  for (int i=0; i<numElems; ++i) {
    coll.push_back(i * 4.37);
  }

  // use parallel algorithm to compute square roots:
  std::for_each(std::execution::par,          // execution policy
                coll.begin(), coll.end(),
                [](auto& val) {
                  val = std::sqrt(val);
                });
}
```

numElems	seq	par	
1,000	1.7	11.3	✗
10,000	9.8	29.5	✗
100,000	106.7	79.3	✓
1,000,000	1297.5	868,1	✓
10,000,000	11063.1	7373.5	✓

With VC++ on my laptop (Intel i7 2 cores + HT)

Execution Policies

C++17/C++20

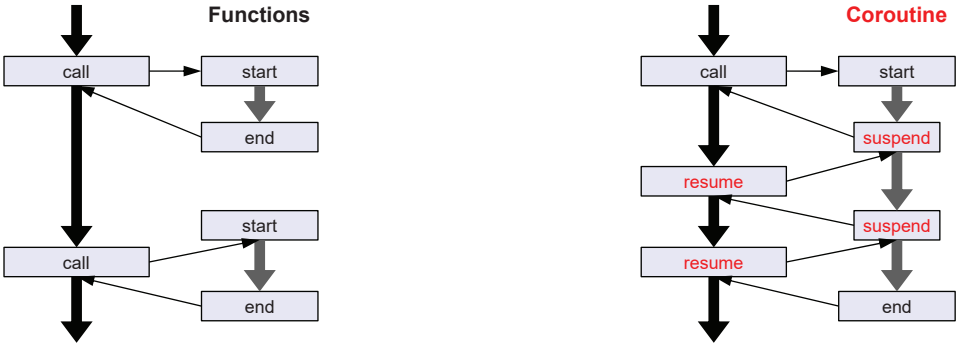


C++20

Coroutines

C++20: Coroutines

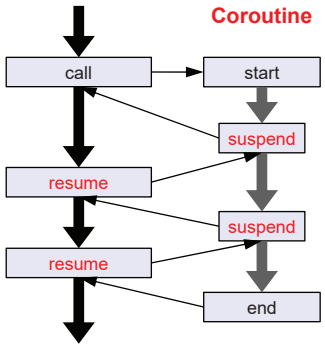
- **Function that can be suspended and resumed**
- **Term by Melvin Conway in 1958**



C++20: Coroutines

- **Function that can be suspended and resumed**
- **Term by Melvin Conway in 1958**

- **C++20: Low-level features for coroutines**
  - **Keywords** to define coroutines and signal suspension or end:
    - `co_await`, `co_yield`, `co_return`
  - **Framework** with type constraints
  - **Utilities** for basic handling
    - More to come in the C++23/C++26 library
- **Stackless approach**
  - Can only suspend a coroutine as a whole
  - Cheap and fast for billions of coroutines



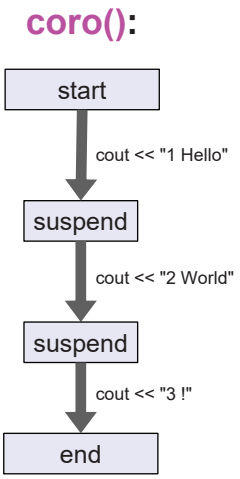
C++20: Coroutines

C++20

```
// ordinary function:
void func()
{
  int i = 0;
  std::cout << ++i << " Hello\n";
  std::cout << ++i << " World\n";
  std::cout << ++i << " !\n";
}
```

- `co_await` makes it a coroutine
- `CoroIF` is the *coroutine interface*
  - User-defined API to deal with the coroutine
  - Has member type `promise_type` as customization point

```
// coroutine:
CoroIF coro()
{
  int i = 0;
  std::cout << ++i << " Hello\n";
  co_await std::suspend_always{};   // suspend
  std::cout << ++i << " World\n";
  co_await std::suspend_always{};   // suspend
  std::cout << ++i << " !\n";
}
```



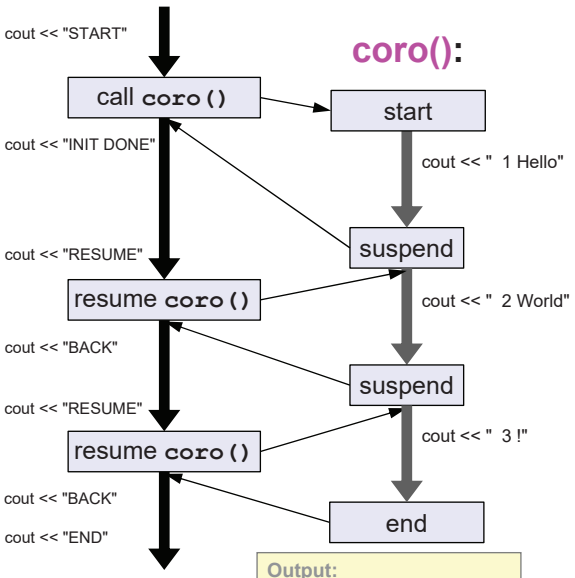
C++20: Basic Example of Using Eager Coroutines

C++20

`CoroIF` makes it an *eager* coroutine

```
CoroIF coro()
{
  int i = 0;
  std::cout << "  " << ++i << " Hello\n";
  co_await std::suspend_always{};   // SUSPEND
  std::cout << "  " << ++i << " World\n";
  co_await std::suspend_always{};   // SUSPEND
  std::cout << "  " << ++i << " !\n";
}
```

```
int main()
{
  std::cout << "START\n";
  // call coroutine:
  CoroIF coroTask = coro();
  std::cout << "INIT DONE\n";

  // loop to resume the coroutine until it is done:
  while (coroTask.isResumable()) {
    std::cout << "RESUME\n";
    coroTask.resume();              // RESUME
    std::cout << "BACK\n";
  }
  std::cout << "END\n";
}
```



Output:

```
START
    1 Hello
INIT DONE
RESUME
    2 World
BACK
RESUME
    3 !
BACK
END
```

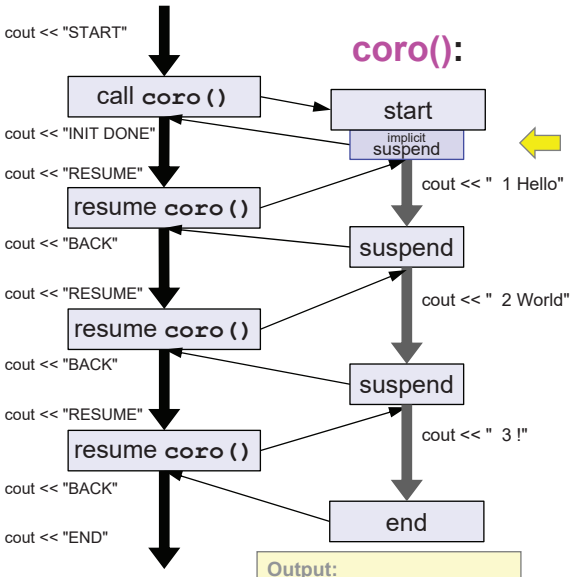
C++20: Basic Example of Using Lazy Coroutines

C++20

```
CoroIF coro()
{
  int i = 0;
  std::cout << "  " << ++i << " Hello\n";
  co_await std::suspend_always{};   // SUSPEND
  std::cout << "  " << ++i << " World\n";
  co_await std::suspend_always{};   // SUSPEND
  std::cout << "  " << ++i << " !\n";
}
```

CoroIF can make it a *lazy* coroutine

```
int main()
{
  std::cout << "START\n";
  // call coroutine:
  CoroIF coroTask = coro();
  std::cout << "INIT DONE\n";

  // loop to resume the coroutine until it is done:
  while (coroTask.isResumable()) {
    std::cout << "RESUME\n";
    coroTask.resume();                // RESUME
    std::cout << "BACK\n";
  }
  std::cout << "END\n";
}
```



Output:

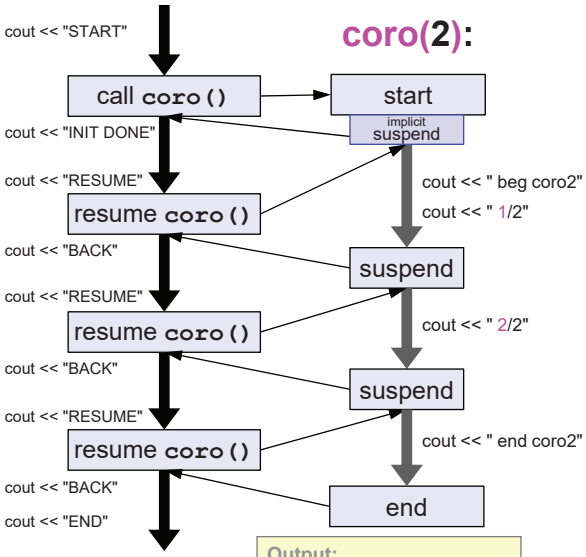
```
START
INIT DONE
RESUME
    1 Hello
BACK
 ESUME
    2 World
BACK
RESUME
    3 !
BACK
END
```


Coroutine Example with co_await

C++20

```
CoroIF coro(int max)
{
  std::cout << " beg coro" << max << "\n";
  // loop from 1 to max:
  for (int v = 1; v <= max; ++v) {
    std::cout << ' ' << v << '/' << max << '\n';
    co_await std::suspend_always(); // SUSPEND
  }
  std::cout << " end coro" << max << "\n";
}
```

```
int main()
{
  std::cout << "START\n";
  // call coroutine:
  CoroIF coroTask = coro(2);
  std::cout << "INIT DONE\n";

  // loop to resume the coroutine until it is done:
  while (coroTask.isResumable()) {
    std::cout << "RESUME\n";
    coroTask.resume();                // RESUME
    std::cout << "BACK\n";
  }
  std::cout << "END\n";
}
```



Output:

```
START
INIT DONE
RESUME
    beg coro2
    1/2
BACK
RESUME
    2/2
BACK
RESUME
    end coro2
BACK
END
```

C++20: Using Coroutines Multiple Times

C++20

```
CoroIF coro(int max)
{
  std::cout << " beg coro" << max << "\n";
  // loop from 1 to max:
  for (int v = 1; v <= max; ++v) {
    std::cout << ' ' << v << '/' << max << '\n';
    co_await std::suspend_always(); // SUSPEND
  }
  std::cout << " end coro" << max << "\n";
}

int main()
{
  std::cout << "START\n";
  CoroIF coroTask8 = coro(8);          // assume we start coroutines lazily
  CoroIF coroTask2 = coro(2);
  std::cout << "INIT DONE\n";

  std::cout << "RESUME8:\n";
  coroTask8.resume();                  // RESUME coro(8)

  while (coroTask2.isResumable()) {
    std::cout << " RESUME2:\n";
    coroTask2.resume();                // RESUME coro(2)
  }

  std::cout << "RESUME8:\n";
  coroTask8.resume();                  // RESUME coro(8)
  std::cout << "END\n";
}
```

Output:

```
START
INIT DONE
RESUME8:
    beg coro8
    1/8
RESUME2:
    beg coro2
    1/2
RESUME2:
    2/2
RESUME2:
    end coro2
RESUME8:
    2/8
END
```


C++20: Implementing Coroutine Interfaces

C++20

```
CoroIF coro(int max)
{
  std::cout << " beg coro" << max << "\n";
  // loop from 1 to max:
  for (int v = 1; v <= max; ++v) {
    std::cout << ' ' << v << '/' << max << '\n';
    co_await std::suspend_always(); // SUSPEND
  }
  std::cout << " end coro" << max << "\n";
}

int main()
{
  std::cout << "START\n";
  // call coroutine:
  CoroIF coroTask = coro(2);
  std::cout << "INIT DONE\n";

  // loop to resume the coroutine until it is done:
  while (coroTask.isResumable()) {
    std::cout << "RESUME\n";
    coroTask.resume();                 // RESUME
    std::cout << "BACK\n";
  }
  std::cout << "END\n";
}
```

```
#include <coroutine>
...
class [[nodiscard]] CoroIF {
 public:
  // required type for customization:
  struct promise_type {
    ...
  };

  // native coroutine handle with state:
  using CoroT = std::coroutine_handle<promise_type>;
  CoroT hdl;   // THE handle

  // constructor and destructor:
  CoroIF(CoroT h) : hdl{h} {
  }
  ~CoroIF() {
    if (hdl) hdl.destroy();
  }

  // API:
  bool isResumable() const {
    return hdl && !hdl.done();
  }
  void resume() {
    if (isResumable()) {
      hdl.resume(); // do RESUME (also: hdl())
    }
  }
};
```

Also deal with copy and move

C++20: Implementing Coroutine Interfaces

C++20

```
CoroIF coro(int max)
{
  std::cout << " beg coro" << max << "\n";
  // loop from 1 to max:
  for (int v = 1; v <= max; ++v) {
    std::cout << v << "/" << max << '\n';
    co_await std::suspend_always{};
  }
  std::cout << " end coro" << max << "\n";
}

int main()
{
  std::cout << "START\n";
  // call coroutine:
  CoroIF coroTask = coro(2);
  std::cout << "INIT DONE\n";

  // loop to resume the coroutine until it is done:
  while (coroTask.isResumable()) {
    std::cout << "RESUME\n";
    coroTask.resume();
    std::cout << "BACK\n";
  }
  std::cout << "END\n";
}
```

[[nodiscard]]: Don't ignore the returned interface

Interface to the compiler
init, eager/lazy, exceptions, ...

The state of the coroutine
(here bound to the interface)

Interface for the user to deal with the started coroutine

not standardized (yet)

```
#include <coroutine>
...
class [[nodiscard]] CoroIF {
 public:
  // required type for customization:
  struct promise_type {
    ...
  };

  // native coroutine handle with state:
  using CoroT = std::coroutine_handle<promise_type>;
  CoroT hdl;   // THE handle

  // constructor and destructor:
  CoroIF(CoroT h) : hdl{h} {
  }
  ~CoroIF() {
    if (hdl) hdl.destroy();
  }

  // API:
  bool isResumable() const {
    return hdl && !hdl.done();
  }
  void resume() {
    if (isResumable()) {
      hdl.resume(); // do RESUME (also: hdl())
    }
  }
};
```

Also deal with copy and move

C++20: Coroutine Promise Type

C++20

```
#include <coroutine>
...
class [[nodiscard]] CoroIF {
 public:
  // required type for customization:
  struct promise_type {
    auto get_return_object() {          // yield value returned by coro call
      return CoroIF{CoroT::from_promise(*this)};
    }
    auto initial_suspend() {            // eager or lazy?
      return std::suspend_always{};     // - suspend immediately
    }
    void unhandled_exception() {        // deal with exceptions
      std::terminate();                 // - terminate the program
    }
    void return_void() {                // on end or co_return;
    }
    auto final_suspend() noexcept {     // clean-ups / postprocessing
      return std::suspend_always{};     // - suspend at the end
    }
  };

  using CoroT = std::coroutine_handle<promise_type>;
  CoroT hdl;   // THE handle
  ...
};
```

```
try {
  co_await prm.initial_suspend();
  std::cout << "beg coro" << max << "\n";
  for (int v = 1; v <= max; ++v) {
    std::cout << v << "/" << max << '\n';
    co_await std::suspend_always();
  }
  std::cout << "end coro" << max << "\n";
}
catch (...) {
  prm.unhandled_exception();
}
co_await prm.final_suspend();
```

Coroutine Example with co_yield

C++20

```
template <typename T>
Gen<std::ranges::range_value_t<T>>
nextElem(T coll)
{
  for (auto elem : coll) {
    co_yield elem;   // yield value and SUSPEND
  }
}
```

```
int main()
{
  // define generator over elements of a vector:
  std::vector<int> coll{0, 8, 15, 33, 42, 7};
  auto gen = nextElem(coll);

  // start loop to process elem by elem:
  while (gen.isResumable()) {
    gen.resume()) {                // RESUME
    std::cout << gen.getValue() << '\n';
    ...
  }
}
```

```
template<typename T>
class [[nodiscard]] Gen {
 public:
  // helper type for customization:
  struct promise_type {
    ...
    T yieldValue;
    auto yield_value(T value) {
      yieldValue = value;
      return std::suspend_always{};
    }
  };

  using CoroT = std::coroutine_handle<promise_type>;
  CoroT hdl;   // native coroutine handle

  Gen(CoroT h) : hdl{h} {
  }
  ~Gen() {
    if (hdl) hdl.destroy();
  }

  void resume() {
    hdl.resume();
  }
  T getValue() const {
    return hdl.promise().yieldValue;
  }
};
```


Coroutine Example with co_return

C++20

```
StrRetGen foo()
{
  std::string ret = "Hello";
  co_await std::suspend_always{};   // SUSPEND
  ret += " World";
  co_await std::suspend_always{};   // SUSPEND
  ret += "!";
  co_return ret;
}
```

```
int main()
{
  using namespace std::literals;

  // define generator over elements of a vector:
  StrRetGen gen = foo();

  // start loop to resume foo():
  while (gen.isResumable()) {
    gen.resume();                   // RESUME
    std::this_thread::sleep_for(500ms);
  }
  std::cout << gen.returnValue() << '\n';
}
```

```
class [[nodiscard]] StrRetGen {
 public:
  // helper type for customization:
  struct promise_type {
    ...
    std::string retValue;
    void return_value(const auto& v) {
      retValue = v;
    }
  };

  // native coroutine handle with state:
  using CoroT = std::coroutine_handle<promise_type>;
  CoroT hdl;

  StrRetGen(CoroT h) : hdl{h} {
  }
  ~StrRetGen() {
    if (hdl) hdl.destroy();
  }

  bool resume() {
    hdl.resume();
  }
  std::string returnValue() const {
    return hdl.promise().retValue;
  }
};
```

Coroutine versus Class

C++20

```
CoroIF process(int max)
{
  prepare();
  co_await …;     // SUSPEND

  for (int val = 1; val <= max; ++val) {
    std::cout << val * val << '\n';
    co_await …;   // SUSPEND
  }
}

int main()
{
  auto gen = process(7);
  while (gen.nextValue()) {
    …
  }
}
```

```
class process {
 private:
  int max;
  enum State {init, loop, done};
  State state;
  int val;
 public:
  process(int m)
   : max{m}, state{init} {
  }

  bool nextValue() {
    if (state == init) {
      prepare();
      val = 1;
      state = loop;
    }
    else if (state == loop) {
      std::cout << val * val << '\n';
      ++val;
      if (val > max) state = done;
    }
    return state != done;
  }
};
```


C++20: Coroutines and Call-by-Reference

C++20

- **Be very careful with call-by-reference in coroutines**
  - Referenced object has to exist as long as the coroutine is used

```
template <typename T>
Gen<std::ranges::range_value_t<T>> nextElem(const T& coll)
{
  for (auto elem : coll) {
    co_yield elem;   // yield value and suspend
  }
}

std::vector<int> getTemporaryColl();   // fwd. decl.

int main()
{
  auto gen = nextElem(getTemporaryColl());
  // OOPS, returned vector destroyed here
  while (gen.resume()) {                // runtime ERROR
    std::cout << gen.getValue() << '\n';
  }
}
```

Don't do this
- **Always pass by value**
- Caller could use `std::views::all()` or `std::ref()` to pass by reference

C++20: Coroutines and Range-Based `for` Loop

C++20

- **Be very careful with call-by-reference in coroutines**
  - Referenced object has to exist as long as the coroutine is used

```
void func(const int& num)
{
  for(int i = 0; i < num; ++i) {
    foo(i);
  }
};

func(10);                          // OK
```

```
std::generator<int> coro(const int& num)   // std::generator<> comes with C++23
{
  for(int i = 0; i < num; ++i) {
    co_yield i;
  }
};

for (auto val : coro(10)) {   // runtime ERROR: undefined behavior
  foo(val);
}
```

Don't use references to temporaries in the range-based `for` loop
- See http://wg21.link/p2012

Fixed with C++23

C++20: Issues with Coroutines

C++20

- **No coroutine interface types yet**
  - Problem: Many many different use cases
  - C++23 will probably provide **`std::generator<>`**
    - See http://wg21.link/p2168
    - Example: http://godbolt.org/z/bjq6PrPqo
  - Other examples:
    - http://github.com/lewissbaker/cppcoro
    - https://github.com/mpusz/mp-coro
- **Memory Management**

Coroutine Interface `std::generator<>` C++23

- https://wg21.link/p2502
- 26.8 Range generators [coro.generator]

```
#include <generator>
…

std::generator<int> intValues(int val = 0)
{
  while(true) {
    co_yield val++;
  }
}

int main()
{
  for (auto i : intValues() | std::views::take(3)) {
    std::cout << i << ' ';    // prints 0 1 2
  }
}
```


C++20

Coroutines In Detail

C++20: Coroutine Frames

C++20

- **When coroutines are started**
  - A **coroutine frame** is created
    - Usually on the heap (except for simple cases)
  - All parameters are copied into the frame
    - For references, the references are copied not their values
  - A promise is created to control its behavior and state
- **While the coroutine runs**
  - The promise is used as customization point to
    - return the coroutine interface on its start
    - define whether to suspend immediately at the beginning
    - define whether to suspend immediately at the end
    - deal with exceptions
    - deal with yield and return values

Compilers may not use the heap if
- the lifetime of the coroutine is within the lifetime of the caller
- the compiler can see everything to compute the size of the frame

For the current state, see http://www.godbolt.org/z/5rvxq6ja5

C++20: Coroutine and its Promise API

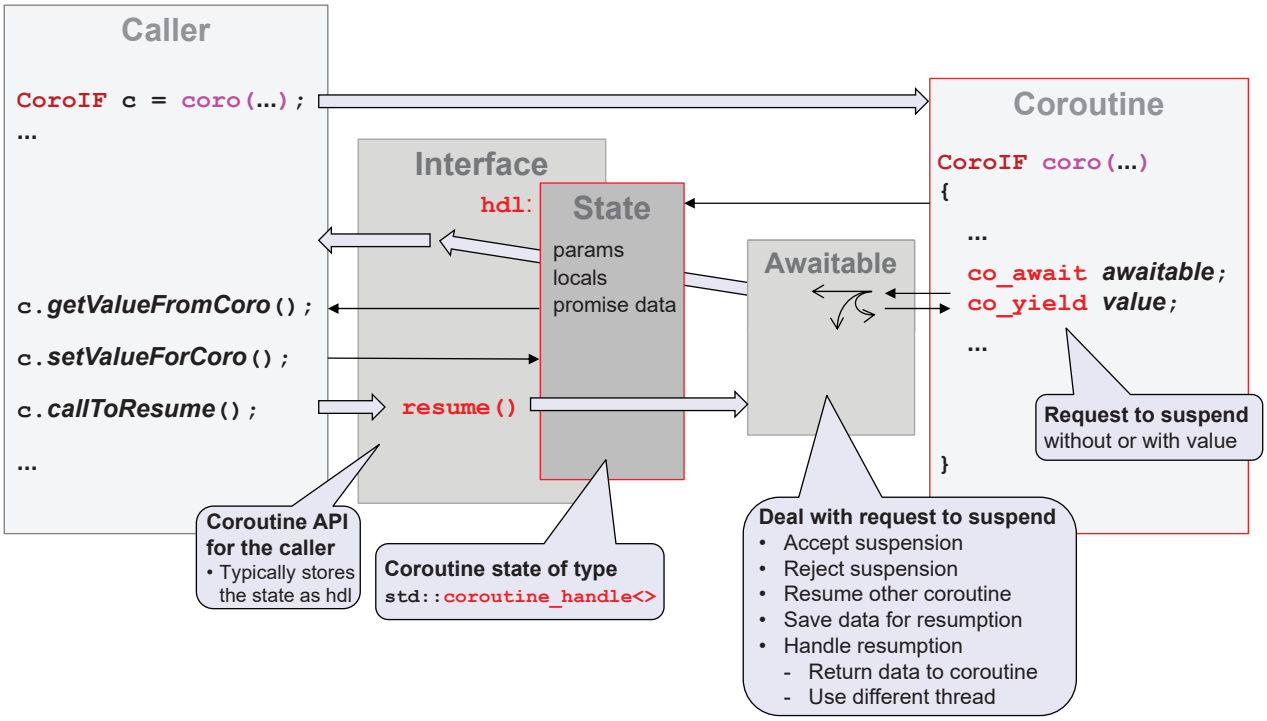
C++20

References are copied as references

```
Parameters params{...};    // copy of coroutine parameters
PromiseType prm{...};      // promise type of the coroutine

// to return from the coroutine call:
return prm.get_return_object();

// to execute the coroutine body:
try {
  co_await prm.initial_suspend();
  std::cout << "coro" << max << '\n';
  for (int v = 1; v <= max; ++v) {
    std::cout << v << '/' << max << '\n';
    co_await std::suspend_always{}; // suspend
  }
  std::cout << "end coro" << max << '\n';
}
catch (...) {
  if (!initial-await-resume-called) throw;
  prm.unhandled_exception();
}
co_await prm.final_suspend();

// on co_yield value;
prm.yield_value(Type);

// on end or co_return;
prm.return_void();

// on co_return value;
prm.return_value(Type);
```

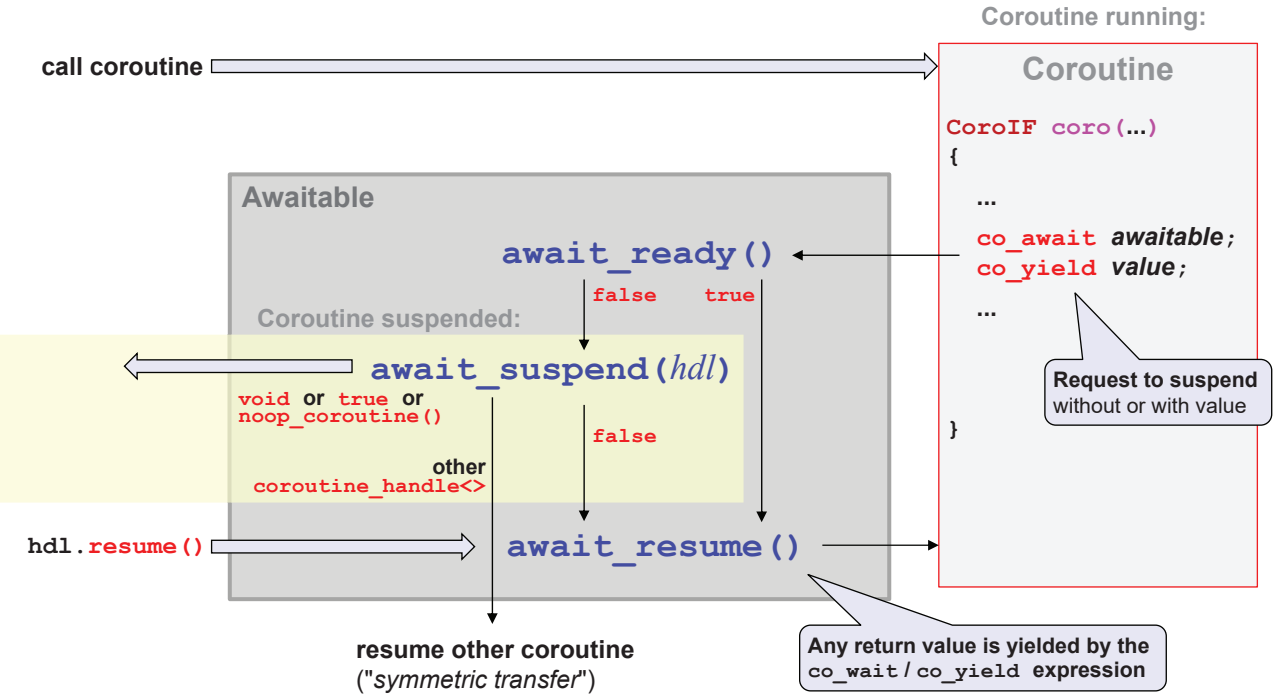
Control Flow with Coroutines

C++20



Control Flow within Awaitables

C++20



C++20: Awaitables and Awaiters

C++20

- ***Awaitables***
  - Objects that handle suspensions (and the following resumption)
  - Basic API to deal with `co_await` and (indirectly) `co_yield`
- ***Awaiters***
  - The typical way to implement an ***awaitable***
  - Member functions:
    - `await_ready()` to temporarily disable suspensions
    - `await_suspend()` to react after suspension
    - `await_resume()` to handle a resumption

- reject suspensions
- start other coroutine on suspension
- return value from caller on resumption
- change thread

- **Standard awaiters:**
  - `std::suspend_always{}` : always accept request to suspend
  - `std::suspend_never{}` : never accept request to suspend

C++20: API of Awaiters

C++20

bool `await_ready()`

- to temporarily disable suspensions
  - The coroutine is **not suspended** yet
- Return value `false` **to accept the suspension**

Type `await_suspend(`*coroHandle*`)`

- to react after suspension of the coroutine (passed as *coroHandle*)
  - The coroutine is already **suspended**
- Return value can be
  - `void` or `true` or `noop_coroutine()` to give **control back to the caller**
  - `false` **to resume** the coroutine again
  - *otherHdl* **to resume another** coroutine instead

Type `await_resume()`

- to handle a resumption (or non-suspension)
  - The coroutine is **not or no longer suspended**
- Return value is what `co_await`/`co_yield` yields back to the coroutine

C++20: A Simple Awaiter

C++20

```
class Awaiter {
 public:
  bool await_ready() const noexcept {
    std::cout << "   await_ready()\n";
    return false;  // true: do NOT (try to) suspend
  }
  void await_suspend(auto hdl) const noexcept {
    std::cout << "   await_suspend()\n";
  }
  void await_resume() const noexcept {
    std::cout << "   await_resume()\n";
  }
};

CoroIF coro(int max)
{
  std::cout << " beg coro" << max << '\n';
  for (int v = 1; v <= max; ++v) {
    std::cout << ' ' << v << '/' << max
              << '\n';
    co_await Awaiter{};  // SUSPEND
  }
  std::cout << " end coro" << max << '\n';
}
```

```
auto coroIF = coro(2);
std::cout << "START\n";

std::cout << "LOOP:\n";
while (coroIF.resume()) {   // RESUME
  std::cout << "INLOOP\n";
}
std::cout << "END\n";
```

Output:

```
START
LOOP:
  beg coro2
  1/2
    await_ready()
    await_suspend()
INLOOP
    await_resume()
  2/2
    await_ready()
    await_suspend()
INLOOP
    await_resume()
  end coro2
END
```


Coroutines: Sending Values Back on Resumption

C++20

```
CoroBackInt coroLoop()
{
 while (true) {
   // suspend for next value:
   int v = co_await BackAwaiter{};
   std::cout << "  coro: " << v << '\n';
   ...  // process next value
 }
}
```

```
// start coroutine (start eager to resume with value):
auto loopCoro = coroLoop();

for (auto val : {0, 8, 15, 47, 11}){
  ...
  // let coroutine process the next value:
  std::cout << "resume with " << val << '\n';
  loopCoro.process(val);
  ...
}
```

Output:

```
resume with 0
  coro: 0
resume with 8
  coro: 8
resume with 15
  coro: 15
...
```

```
class [[nodiscard]] CoroBackInt {
  ...
  struct promise_type {
    auto initial_suspend() {
      return std::suspend_never{};
    }
    ...
    int resumeVal; // from caller on resumption
  };

  void process(int val) {
    hdl.promise().resumeVal = val;
    hdl.resume();
  }
};
```

```
class BackAwaiter {
  using HdlT = CoroBackInt::CoroT;
  HdlT hdl = nullptr;
 public:
  bool await_ready() const noexcept {
    return false;  // do suspend
  }
  void await_suspend(HdlT h) noexcept {
    hdl = h;         // save handle for resumption
  }
  auto await_resume() const noexcept {
    // pass value from resumption back to coroutine:
    return hdl.promise().resumeVal;
  }
};
```

Coroutines: Sending Values Back and Forth

C++20

```
GenBack<double> coroLoopSqrt()
{
  double val = 0;       // value for square root
  while (true) {
    // yield square root and suspend for next value:
    val = co_yield std::sqrt(val);
  }
}
```

```
// start coroutine (should start eager to ignore first sqrt):
auto sqrtLoop = coroLoopSqrt();

for (double d : {1, 2, 3, 4, 5, 6, 7}) {
  // let coroutine process the next square root:
  double res = sqrtLoop.process(d);
  std::cout << d << " => " << res << '\n';
}
```

Output:

```
1 => 1
2 => 1.41421
3 => 1.73205
4 => 2
5 => 2.23607
6 => 2.44949
7 => 2.64575
```

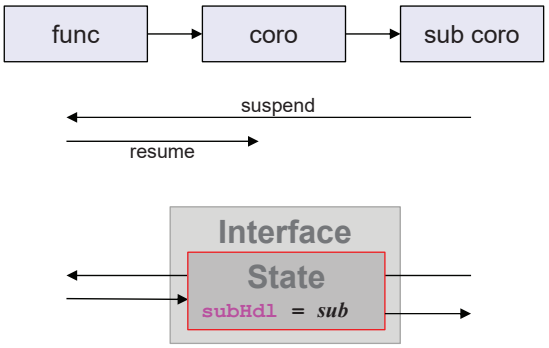
```
template<typename T>
class [[nodiscard]] GenBack {
  ...
  struct promise_type {
    ...
    T suspValue{};   // to caller on suspension
    T resmValue{};   // from caller on resumption
    auto yield_value(T value) {
      suspValue = value;
      return BackAwaiter<CoroT>{};
    }
  };
  T process(T val) const {
    hdl.promise().resmValue = val;
    hdl.resume();
    return hdl.promise().suspValue;
  }
};
```

```
template<typename HdlT>
class BackAwaiter {
  HdlT hdl = nullptr;
 public:
  bool await_ready() const noexcept {
    return false;  // do suspend
  }
  void await_suspend(HdlT h) noexcept {
    hdl = h;       // save handle for resumption
  }
  auto await_resume() const noexcept {
    // pass value from resumption back to coro:
    return hdl.promise().resmValue;
  }
};
```


Nested Coroutines

C++20

- Coroutines are stackless
- Only the top-level coroutine can be suspended
- Use awaiters to continue sub-coroutines




```
class [[nodiscard]] CoroSubIF {
 public:
  ...
  HdlT hdl;
  ...
  struct promise_type {
    ...
    CoroT subHdl = nullptr;       // sub-coro (if any)
    CoroT upHdl = nullptr;        // to remove sub-coro
  };

  bool await_ready() const noexcept {
    return false;  // do suspend
  }
  void await_suspend(auto uh) noexcept {
    uh.promise().subHdl = hdl;  // store sub-coro
    hdl.promise().upHdl = uh;
  }
  void await_resume() const noexcept {
    hdl.promise().upHdl.promise().subHdl = nullptr;
  }

  void resume() {
    // find deepest sub-coroutine not done yet:
    HdlT h = hdl;
    while (h.promise().subHdl &&
           !h.promise().subHdl.done()) {
      h = h.promise().subHdl;
    }
    h.resume();      // and RESUME
  }
};
```

Nested Coroutines

C++20

```
CoroSubIF bar()
{
  std::cout << "  bar(): PART1/2\n";
  co_await std::suspend_always(); // SUSPEND
  std::cout << "  bar(): PART2/2\n";
}

CoroSubIF foo()
{
  std::cout << "foo(): PART1/3\n";
  co_await std::suspend_always(); // SUSPEND
  std::cout << "foo(): PART2/3\n";
  co_await bar();                 // call bar()
  std::cout << "foo(): PART3/3\n";
}

CoroSubIF coro()
{
  co_await foo();                 // call foo()
  co_await std::suspend_always(); // SUSPEND
  co_await foo();                 // call foo()
}

int main()
{
  auto coroTask = coro();         // init
  while (coroTask.isResumable()) {
    coroTask.resume();            // RESUME
  }
}
```

```
class [[nodiscard]] CoroSubIF {
 public:
  ...
  HdlT hdl;
  ...
  struct promise_type {
    ...
    CoroT subHdl = nullptr;       // sub-coro (if any)
    CoroT upHdl = nullptr;        // to remove sub-coro
  };

  bool await_ready() const noexcept {
    return false;  // do suspend
  }
  void await_suspend(auto uh) noexcept {
    uh.promise().subHdl = hdl;  // store sub-coro
    hdl.promise().upHdl = uh;
  }
  void await_resume() const noexcept {
    hdl.promise().upHdl.promise().subHdl = nullptr;
  }

  void resume() {
    // find deepest sub-coroutine not done yet:
    HdlT h = hdl;
    while (h.promise().subHdl &&
           !h.promise().subHdl.done()) {
      h = h.promise().subHdl;
    }
    h.resume();     // and RESUME
  }
};
```


Coroutines: Memory Management

C++20

- To avoid that coroutines use heap memory, define:

```
void* operator new(std::size_t sz)
void  operator delete(void* ptr, std::size_t sz)
```

```
class [[nodiscard]] CoroIF {
  // provide memory as polymorphic memory resource (C++17) for all coroutines:
  inline static std::pmr::monotonic_buffer_resource
    monobuf{beginOfMemory, sizeOfMemory, std::pmr::null_memory_resource()};
  inline static std::pmr::synchronized_pool_resource mempool{&monobuf};
  ...
 public:
  struct promise_type {
    ...
    // define how the coroutine allocated memory for the state:
    void* operator new(std::size_t sz) {
      return mempool.allocate(sz);
    }
    void operator delete(void* ptr, std::size_t sz) {
      mempool.deallocate(ptr, sz);
    }
  };
  ...
};
```

Trick:
`operator new(...) = delete`
to see whether memory is allocated on the heap

C++20

Modules

C++20: Modules

- **Why:**
  - Scaling C++ with components
    - scaling beyond 1,000,000,000 LoC
  - Note: Due to templates almost everything is in header files
- **Key approach:**
  - Combine multiple translation units (header and source files) into one component
- **Benefits:**
  - Significant compile-time saving
    - many code parsed and compiled only once (if not inlined)
  - No runtime overhead
  - Clear abstraction and information hiding

Header Files vs. Modules

C++20

```
#ifndef MYCLASS_HPP
#define MYCLASS_HPP

#define SAFE_MODE
#include <iostream>

class Type {
  …
};

template<typename T>
Type toType(const T&) {
  …
}

namespace myclassintern {
  inline void init(Type&) {
    …
  }
}

#endif // MYCLASS_HPP
```

#include makes
- **everything** visible
- code compiled **multiple times**

```
#include "MyClass.hpp"

Type x = toType(42);
```

module interface unit (translation unit with module declaration):

```
module;
```

global module fragment (not exported)

```
#define SAFE_MODE
#include <iostream>

export module Type;   // declare module "Type"

export class Type {
  …
};

export template<typename T>
Type toType(const T&) {
  …
}

void init(Type&) {
  …
}
```

symbols visible with **import**

Type.gcm, Type.ifc, or Type.bmi

Pre-compiled code (non-portable format)

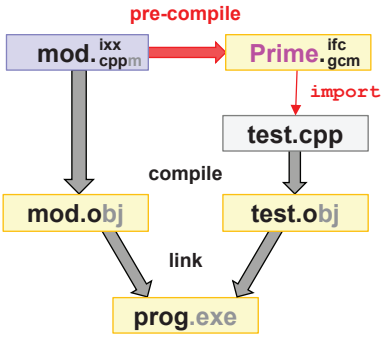
import makes
- only **exports** visible

```
import Type;

Type x = toType(42);
```


C++20 Modules: Primary Interface and its Use

C++20

```
export module Prime; // primary interface of Prime

export bool isPrime(int value)
{
  for (int i = 2; i <= value/2; ++i) {
    if (value % i == 0) {
      return false;
    }
  }
  return value > 1;   // 0 and 1 are no prime numbers
}
```




```
#include <iostream>

import Prime;

int main()
{
  int count = 20;
  std::cout << "first " << count << " prime numbers:\n";
  int num = 0;
  for (int val = 1; num < count; ++val) {
    if (isPrime(val)) {
      std::cout << "  " << val << '\n';
      ++num;
    }
  }
}
```

A Bigger Modules Example

C++20

```
module;

#include <vector>

export module Prime;  // primary interface of Prime

export bool isPrime(int value);

class PrimeColl {
  std::vector<int> vals;
 public:
  // init collection of num prime numbers:
  PrimeColl(int num) {
    for (int v=2; std::ssize(vals)<num; ++v) {
      if (isPrime(v)) {
        vals.push_back(v);
      }
    }
  }

  // provide iterator API:
  using iterator = decltype(vals)::const_iterator;
  auto begin() const { return vals.begin(); }
  auto end() const { return vals.end(); }
};

export PrimeColl primeColl(int num) {
  return PrimeColl{num};
}
```

Global module fragment
- defines and includes
- not exported

Type `PrimeColl` is
- not visible
- but reachable

```
module Prime;  // implementation unit of Prime

bool isPrime(int value)
{
  for (int i = 2; i <= value/2; ++i) {
    if (value % i == 0) {
      return false;
    }
  }
  return value > 1; // 0 and 1 are no primes
}
```

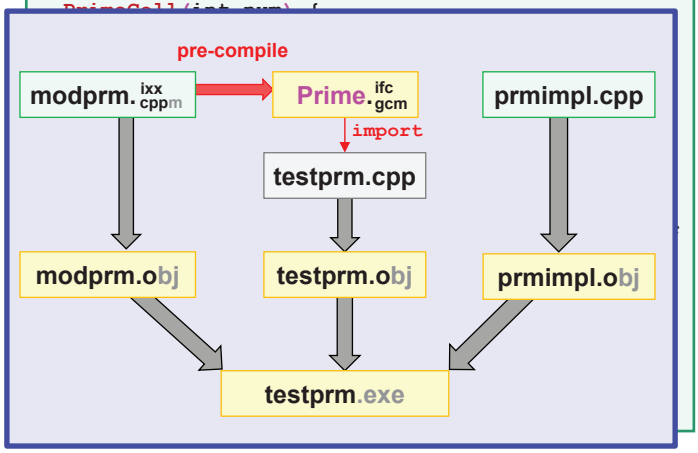
```
#include <iostream>
#include <list>
import Prime;

int main()
{
 std::list coll{0, 8, 15, 47, 11, 9};
  for (int val : coll) {
    if (isPrime(val)) {
      std::cout << val << '\n';
    }
  }

  auto primes = primeColl(10);
  for (auto p : primes) {
    std::cout << p << ' ';
  }
  std::cout << '\n';
}
```


A Bigger Modules Example

C++20

```
module;

#include <vector>

export module Prime;  // primary interface of Prime

export bool isPrime(int value);

class PrimeColl {
  std::vector<int> vals;
 public:
  // init collection of num prime numbers:
```

modprm.cppm




```
module Prime;  // implementation unit of Prime

bool isPrime(int value)
{
  for (int i = 2; i <= value/2; ++i) {
    if (value % i == 0) {
      return false;
    }
  }
  return value > 1; // 0 and 1 are no primes
}
```

prmimpl.cpp

```
#include <iostream>
#include <list>
import Prime;

int main()
{
 std::list coll{0, 8, 15, 47, 11, 9};
  for (int val : coll) {
    if (isPrime(val)) {
      std::cout << val << '\n';
    }
  }

  auto primes = primeColl(10);
  for (auto p : primes) {
    std::cout << p << ' ';
  }
  std::cout << '\n';
}
```

testprm.cpp

C++20: Module Names

C++20

- **Modules can have arbitrary names**
  - Including . with no special meaning
  - Names can still be used inside for something else
- **Modules don't belong to or create a namespace**
  - Exported scope is imported scope
  - Convention: Module name is exported namespace

```
export module MyMod;   // primary interface

namespace MyMod {
  int helper(int i);
  export class Customer {
    …
  };
  export template<typename T>
  Customer toCustomer(const T& x) {
    …
  }
  int helper(int i) {   // not exported
    …
  }
}
```

or

```
export module MyMod;   // primary interface

int helper(int i);
export namespace MyMod {
  class Customer {
    …
  };
  template<typename T>
  Customer toCustomer(const T& x) {
    …
  }
}
int helper(int i) {   // not exported
  …
}
```


Modules: Primary Interface

C++20

Primary interface: *The* API (with *all* exports)

```
module;                  // global module fragment

#include <string>
#include <vector>

export module Mod1;      // primary interface of Mod1

struct Order {
  int count;
  std::string name;
  double price;
  Order(int c, std::string n, double p)
   : count{c}, name{n}, price{p} {
  }
};

export class Customer {
 private:
  std::string name;
  std::vector<Order> orders;
 public:
  Customer(std::string n) : name{n} {
  }
  void buy(int num, std::string n, double p) {
    orders.push_back(Order{num, n, p});
  }
  double sumPrice() const;
  double averagePrice() const;
  void print() const;
};
```

Modules: Implementation Units

C++20

```
module;                          // global module fragment
#include <string>
#include <vector>
export module Mod1;              // primary interface of Mod1
struct Order {
  int count;
  std::string name;
  double price;
  Order(int c, std::string n, double p)
   : count{c}, name{n}, price{p} {
  }
};

export class Customer {
 private:
  std::string name;
  std::vector<Order> orders;
 public:
  Customer(std::string n) : name{n} {
  }
  void buy(int num, std::string n, double p) {
    orders.push_back(Order{num, n, p});
  }
  double sumPrice() const;
  double averagePrice() const;
  void print() const;
};
```

Implementation unit: Definitions to compile

```
module Mod1;    // implementation unit of Mod1

double Customer::sumPrice() const
{
  double sum = 0.0;
  for (const Order& od : orders) {
    sum += od.count * od.price;
  }
  return sum;
}

double Customer::averagePrice() const
{
  ...
}
```

```
module;                          // global module fragment

#include <iostream>
#include <format>

module Mod1;    // implementation unit of Mod1

void Customer::print() const
{
  ...
}
```


Modules: Implementation Units

C++20

```
module;                          // global module fragment
#include <string>
#include <vector>
export module Mod1;              // primary interface of Mod1
struct Order {
  int count;
  std::string name;
  double price;
  Order(int c, std::string n, double p)
   : count{c}, name{n}, price{p} {
  }
};

export class Customer {
 private:
  std::string name;
  std::vector<Order> orders;
 public:
  Customer(std::string n) : name{n} {
  }
  void buy(int num, std::string n, double p) {
    orders.push_back(Order{num, n, p});
  }
  double sumPrice() const;
  double averagePrice() const;
  void print() const;
};
```

Organization of module code does not impact any import

```
module Mod1;    // implementation unit of Mod1

double Customer::sumPrice() const
{
  double sum = 0.0;
  for (const Order& od : orders) {
    sum += od.count * od.price;
  }
  return sum;
}

double Customer::averagePrice() const
{
  ...
}
```

```
module;

#include <ios
#include <for

module Mod1;

void Customer
{
  ...
}
```

```
#include <iostream>

import Mod1;

int main()
{
  Customer c{"Kim"};

  c.buy("table", 59.90);
  c.buy(4, "chair", 9.20);

  c.print();
  auto avg = c.averagePrice();
  std::cout << avg << '\n';
}
```

Cannot use `Order` here

Modules: Internal Partitions

C++20

```
module;                          // global module fragment
#include <string>
#include <vector>
export module Mod2;              // primary interface of Mod2
import :Order;                   // import internal partition Order
export class Customer {
 private:
  std::string name;
  std::vector<Order> orders;
 public:
  Customer(std::string n) : name{n} {
  }
  void buy(int num, std::string n, double p) {
    orde
  }
  double
  double
  void p
};
```

```
module;                          // global module fragment
#include <string>
module Mod2:Order;               // internal partition

struct Order {
 int count;
 std::string name;
 double price;
 Order(int c, std::string n, double p)
  : count{c}, name{n}, price{p} {
 }
};
```

Internal partition: Declarations for the module

```
module Mod2;    // implementation unit of Mod2

double Customer::sumPrice() const
{
  double sum = 0.0;
  for (const Order& od : orders) {
    sum += od.count * od.price;
  }
  return sum;
}

double Customer::averagePrice() const
{
  ...
}
```

```
module;

#include <ios
#include <for

module Mod2;

void Customer
{
  ...
}
```

```
#include <iostream>

import Mod2;

int main()
{
  Customer c{"Kim"};

  c.buy("table", 59.90);
  c.buy(4, "chair", 9.20);

  c.print();
  auto avg = c.averagePrice();
  std::cout << avg << '\n';
}
```


Modules: Interface Partitions

C++20

```
export module Mod3;              // primary interface of Mod3

export import :Customer;         // interface partition
export import :Finance;          // interface partition
...
```

```
module;                          // global module fragment
#include <string>
#include <vector>
export module Mod3:Customer;     // interface partition
import :Order;                   // import internal partition Order
export class Customer {
 private:
  std::string name;
  std::vector<Order> orders;
 public:
  Customer(std::string n) : name{n} {
  }
  void buy(int num, std::string n, double p) {
    orders.push_back(Order{num, n, p});
  }
  double sumPrice() const;
  double averagePrice() const;
  void print() const;
};
```

Interface partition: Additional exports for the user

```
module Mod3;    // implementation unit of Mod3

double Customer::sumPrice() const
{
  double sum = 0.0;
  for (const Order& od : orders) {
    sum += od.count * od.price;
  }
  return sum;
}

double Customer::averagePrice() const
{
  ...
}
```

```
module;

#include <ios
#include <for

module Mod3;

void Customer
{
  ...
}
```

```
#include <iostream>

import Mod3;

int main()
{
  Customer c{"Kim"};

  c.buy("table", 59.90);
  c.buy(4, "chair", 9.20);

  c.print();
  auto avg = c.averagePrice();
  std::cout << avg << '\n';
}
```

Modules: All Module Units

C++20

```
export module Mod3;          // primary interface of Mod3

export import :Customer;     // interface partition
export import :Finance;      // interface partition
...
```

```
module;                      // global module fragment
#include <string>
#include <vector>
export module Mod3:Customer; // interface partition
import :Order;               // import internal partition Order

export class Customer {
 private
  std::s
  std::v
 public:
  Custom
  }
  void b
    orde
  }
  double
  double
  void p
};
```

```
module;                      // global module fragment
#include <string>
module Mod3:Order;           // internal partition

struct Order {
 int count;
 std::string name;
 double price;
 Order(int c, std::string n, double p)
  : count{c}, name{n}, price{p} {
 }
};
```

```
module Mod3;   // implementation unit of Mod3

double Customer::sumPrice() const
{
  double sum = 0.0;
  for (const Order& od : orders) {
    sum += od.count * od.price;
  }
  return sum;
}

double Customer::averagePrice() const
{
  ...
}
```

```
module;

#include <ios
#include <for

module Mod3;

void Customer
{
  ...
}
```

```
#include <iostream>

import Mod3;

int main()
{
  Customer c{"Kim"};

  c.buy("table", 59.90);
  c.buy(4, "chair", 9.20);

  c.print();
  auto avg = c.averagePrice();
  std::cout << avg << '\n';
}
```


C++20: Module Units

C++20

- **Primary Module Interface Unit** — `export module name;`
  - *The API* of the module (exactly once)
  - Pre-compile and compile (*.cppm or *.ixx or *.cpp)
- **Module implementation Unit** — `module name;`
  - Internal definitions of the module
  - Compile only (*.cpp)
- **Internal Partition Unit** — `module name:part;`
  - Internal declarations and definitions of the module
    - Like internal headers
  - Pre-compile and compile (*.cppp or *.ixx or *.cpp)
- **Interface Partition Unit** — `export module name:part;`
  - Part of the exported API in its own file
  - Pre-compile and compile (*.cppm or *.ixx or *.cpp)

C++20: Umbrella Modules

C++20

- **Modules can wrap other modules**
  - and export them as a whole
  - and export parts of the with using

```
export module ModWrapper;

// export another module as a whole:
export import OtherModule;

// export parts of another module:
import LogModule

// class Logger in namespace LogModule in LogModule as ::Logger
export using LogModule::Logger;   // export as ::Logger
export namespace LogModule {
  using LogModule::Logger;        // export as LogModule::Logger
}

export using ::globalLogger;      // export global object globalLogger in LogModule

export using ::log;               // global log() function in LogModule
```


C++20: Module Migration: Global Module Fragment

C++20

- **Module units may start with a global module fragment**
  - Only for `#include` and other preprocessor commands
    - Included content is not available outside the module unit

Global Module Fragment:

```
module;

#include <string>    // make std::string usable here
#include <cstdlib>   // make std::system() usable here

export module System.Cmd;

namespace Stdlib {
  export int system(const char* cmd) {
    return std::system(cmd);
  }
  export int system(const std::string& cmd) {
    return std::system(cmd.c_str());
  }
}
```

Module usage:

```
#include <string>

import System.Cmd;

int main()
{
  Stdlib::system("cd");
  …
}
```

C++20: How to Deal with Header Files

C++20

- **`#include <header>`**
  - In **global module fragment**
  - Everything not used from this will be discarded
  - Everything used will has module linkage
  - Useful for **`#define`** that impacts the header

```
module;

#define NDEBUG
#include <cassert>

export module MyMod;

import <string>;
…
```

- **`import <header>;`**
  - Not in global module fragment
  - Shortcut for declaring a module with everything exported
    - Even imported macros are visible by the importer
      - With all other imports they are ***not***
  - Any **`#define`** before has ***no*** effect on the imported header
    - Header is pre-compiled once for all imports
  - Only guaranteed to work standard C++ headers
    - No guaranteed support for std C headers (e.g., <cassert>) or third party headers
      - Use **`#include`** in global module fragment

Migrating to Modules

C++20

person.hpp

```
#ifndef PERSON_HPP
#define PERSON_HPP

#include <iostream>
#include <string>

class Person {
  std::string name;
 public:
  Person(const std::string& n)
   : name{n} {
  }
  std::string getName() const {
    return name;
  }
};

inline void printPerson(const Person& p) {
  std::cout << "Person " << p.getName() << '\n';
}

#endif // PERSON_HPP
```

```
module;

#include "Person.hpp"

export module Person;

export using ::Person;
export using ::printPerson;
```

```
#include "person.hpp"

int main()
{
  Person p{"TestPerson"};
  printPerson(p);
}
```

```
import Person;

int main()
{
  Person p{"TestPerson"};
  printPerson(p);
}
```

Not yet working with gcc (bug 109679)

C++20: Modules: Open Issues

C++20

- **New artifacts to compile/built module code:**
  - All implementation defined
    - Unclear naming scheme for source files (e.g. primary interface)
      - `.ixx` or `.cppm` or `.cpp`
    - Unclear naming scheme pre-compiled binary/built modules files
      - `.cgm` or `.ifx` or `.bmi` or `.cmi`
    - Unclear locations, non-portable formats
  - Some compilers need build system support (Visual C++)
    - No easy migration of small components to small *module for everybody*
  - Study group looking for recommendations

Modules are not easy and portably usable yet

- **No modularization of the standard library (yet)**
  - C++23 will probably provide (see P2465):
    - `import std;`
      - All in namespace std
    - `import std.compat;`
      - Plus C stuff in global namespace

No macros are re-exported:
- Use `<version>` for feature test macros
- Use `<cassert>` for `assert()`

Importable Header Files

C++23

- **C++23 supports importing C++ header files**
  - Either by importing them individually
    - `import <vector>;`
      - Not guaranteed to work for C header files
      - No macros are re-exported:
        - Use `#include <cassert>` for `assert()`
        - Use `#include <version>` for feature test macros
  - Or by importing them all together:
    - `import std;`
      - All in namespace std
    - `import std.compat;`
      - Plus C stuff in global namespace

see P2465R3

C++20 Modules: gcc/g++

C++20

- http://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Modules.html
  - http://splichal.eu/gccsphinx-final/html/gcc/gcc-command-options/c%2B%2B-modules.html
- **No special file extensions necessary/supported**
- **Compile and pre-compile with:**

```
g++ --std=c++20 -fmodules-ts -c file.cpp
g++ --std=c++20 -fmodules-ts -c -xc++ file.cppm
```

  - creates *file*`.o` and `gcm-cache/`*Mod*`.gcm`

C++20 Modules: Visual C++

C++20

- http://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Modules.html
  - http://splichal.eu/gccsphinx-final/html/gcc/gcc-command-options/c%2B%2B-modules.html
- **Special file extensions supported** (and by default required)
- **Compile and pre-compile module interface:**

```
cl /std:c++20 /c file.ixx
cl /std:c++20 /c /interface /Tpfile.cppm
```

  - creates *file*`.obj` and *Mod*`.ifc`
- **Compile and pre-compile module internal partition:**

```
cl /std:c++20 /c file.???
cl /std:c++20 /c /internalPartition /Tpfile.cppp
```

  - creates *file*`.obj` and *Mod*`.ifc`

No support for `/interface` and `/internalPartition` on one command line
=> Use **clmod.py** from https://github.com/josuttis/cppmodules

C++20

Other Core Features

C++20: Range-Based `for` Loops with Initialization


```
// print elements with leading position:
for (int i = 1; const auto& elem : coll) {
  std::cout << std::format("{:3}: {}\n", ++i, elem);
}

// print elements of coll while it is locked:
for (std::lock_guard lg{collMx}; const auto& elem : coll) {
  std::cout << elem: << elem << '\n';
}

// initialize a string and iterator over its characters:
for (std::string str{getStrings()[0]}; char c : str) {      // OK
  std::cout << c << '\n';
}
```

C++20: `using` for `enum`

C++20

- **`using` directive for `enum class` values**

```
enum class Status {open, progress, done = 9};
```

```
void print(Status s)
{
  switch (s) {
   using enum Status;
   case open:
    std::cout << "open";
    break;
   case progress:
    std::cout << "in progress";
    break;
   case done:
    std::cout << "done";
    break;
  }
}
```

Or:

```
using Status::open, Status::progress,
      Status::done;
```

```
namespace Task {
  using enum Status;
  …
}
…
auto x = Task::open;      // OK
```

```
using enum Status;
…
bool open = false;   // ERROR
```


C++20: Aggregate Initialization with `( )`

C++20

- **Parentheses can be used to initialize aggregates**
  - Enables generic code to support aggregates with `T(`*obj*`)`
  - Does not work for nested initialization

```
struct Trip {
  std::string from;
  std::string to;
};

Trip t0 = {"Rome", "Rio"};          // OK (aggregate initialization)
Trip t1{"Rome", "Rio"};             // OK since C++11
Trip t2("Rome", "Rio");             // OK since C++20 (ERROR before C++20)
Trip t3 = ("Rome", "Rio");          // still ERROR

auto p1 = new Trip{"Rome", "Rio"};  // OK since C++11
auto p2 = new Trip("Rome", "Rio");  // OK since C++20 (ERROR before C++20)

std::vector<Trip> trips;
trips.emplace_back("Rome", "Rio");  // OK since C++20 (uses: new T(...))
```

C++20: Designated Initializers

C++20

- **Initialize aggregates by name of members**
- **Not as flexible as in C**
  - Order must be preserved
    - To match the order of initialization (destruction in opposite order)
  - Can also be used for **`union`**'s

```
struct Value {
  double amount = 0;
  int precision = 2;
  std::string unit = "Dollar";
};

Value v1{100};                              // OK
Value v2{.amount = 100, .unit = "Euro"};    // OK
Value v3{.precision = 8};                   // OK
Value v4{100, .unit = "Euro"};              // ERROR: all or none designated
Value v5{.unit = "$", .amount = 20};        // ERROR: invalid order
Value v6(.amount = 29.9, .unit = "Euro");   // ERROR: only with {}
```


C++20: Lambdas

C++20

- **Using explicit template parameters**
- **Having a default constructor (if no captures)**
- **Lambdas as non-type template parameters**
- **consteval lambdas**
- **New capturing rules**
  - members, **`this`**, and **`*this`**
  - structured bindings
  - init-capturing parameter packs

C++20: Lambdas

- **Apply `consteval` to lambdas:**
  - Ensures that lambdas can only be used at compile-time

```
auto square = [] (auto x) consteval {
                return x * x;
              };
```

- **Template parameters for generic lambdas:**

```
[]<typename T>(std::vector<T> vec) {
  T tmp;
  ...
}
```

- **`[=]` to capture `this` is deprecated**
  - Use e.g. `[=, this]`
- **Capture with initialization can now be variadic**

```
[args = std::move(args)...] { ... }
```


C++20: Example of `consteval` Lambdas with NTTP


```
auto isPrime = [](int value) constexpr {
                 for (int i = 2; i <= value/2; ++i) {
                   if (value % i == 0) {
                     return false;
                   }
                 }
                 return value > 1;   // 0 and 1 are no prime numbers
               };

// local compile-time computation of Num prime numbers:
auto primeNumbers = [isPrime] <int Num> () consteval {
                      std::array<int, Num> primes;
                      int idx = 0;
                      for (int val = 1; idx < Num; ++val) {
                        if (isPrime(val)) {
                          primes[idx++] = val;
                        }
                      }
                      return primes;
                    };

// init array with prime numbers:
std::array primes = primeNumbers.operator()<100>();
```

C++20: New Attributes

C++20

`[[ nodiscard("reason") ]]`

- Optional attribute to explain why result should be used

`[[ likely ]]` and `[[ unlikely ]]`

- Branch prediction

```
if (n <= 0) [[unlikely]] {
  // this branch is considered to be arbitrarily unlikely
  ...
}
```

`[[ no_unique_address ]]`

- Signal to deal with members that provide no data (only functionality)
  - Allocators, function objects for predicates or hashers
- Allows to optimize code not to waste unnecessary space
  - Alternative to EBCO

C++20: Attributes for Branch Prediction

C++20

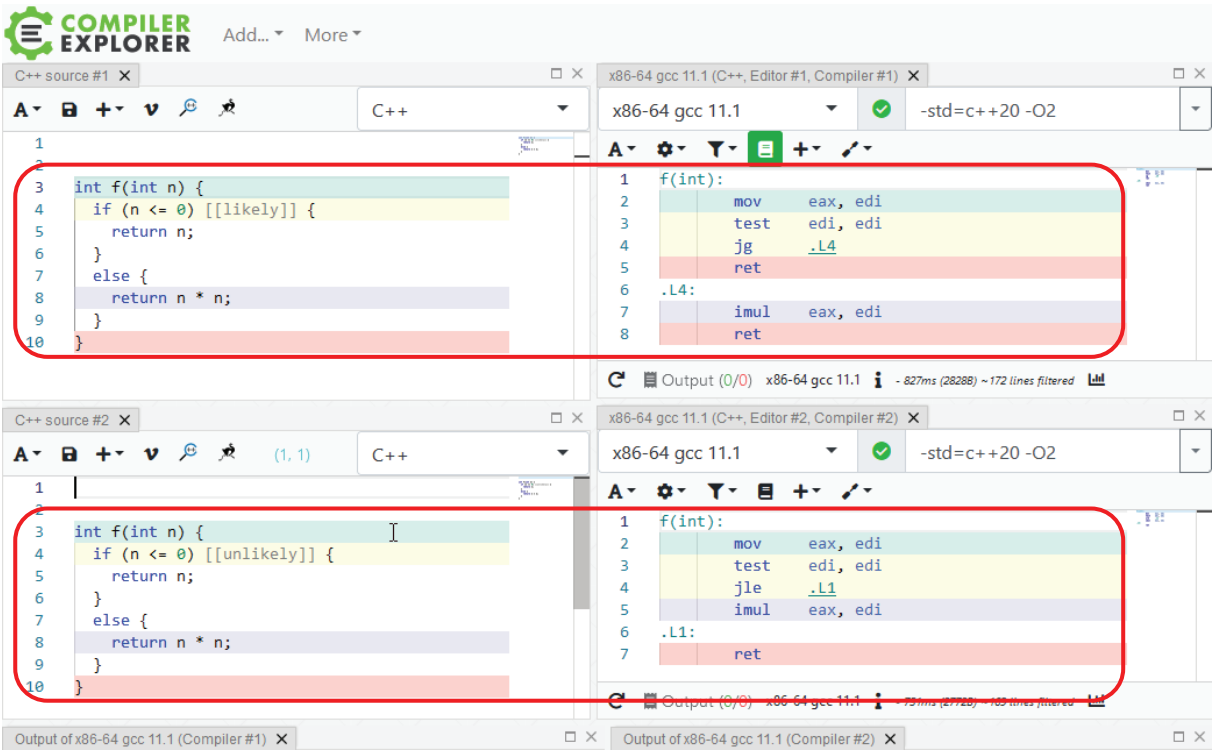
- `[[likely]]` and `[[unlikely]]`
  - Hint to impact compiler-specific optimizations
  - Might have no effect or even be counter-productive

```
if (n <= 0) [[unlikely]] {    // n <= 0 is considered to be arbitrarily unlikely
  return n;
}
else {
  return n * n;
}

switch (n) {
  case 1:
    ...
    break;
  [[likely]] case 2:          // n == 2 is considered to be arbitrarily most likely
    ...
    break;
 default:
    ...
    break;
}
```

Use with care and double-check

C++20: Attributes for Branch Prediction

C++20



C++20: Attribute for Members Without State

C++20

- `[[no_unique_address]]`
  - Avoid memory for members that have no state (provide functionality only)

```
struct Empty {};              // empty class: sizeof(Empty) is 1

struct I {                    // sizeof yields e.g. 4 (which is sizeof(int))
  int i;
};

struct EandI {                // sizeof yields e.g. 8 (sum of sizes of members with alignment)
  Empty e;
  int i;
};

struct EbasedI : Empty {      // sizeof yields e.g. 4
  int i;
};

struct EattrI {               // sizeof yields e.g. 4 (like with EBCO)
  [[no_unique_address]] Empty e;
  int i;
};

struct IattrE {               // sizeof yields e.g. 4 (like with EBCO)
  int i;
  [[no_unique_address]] Empty e;
};
```

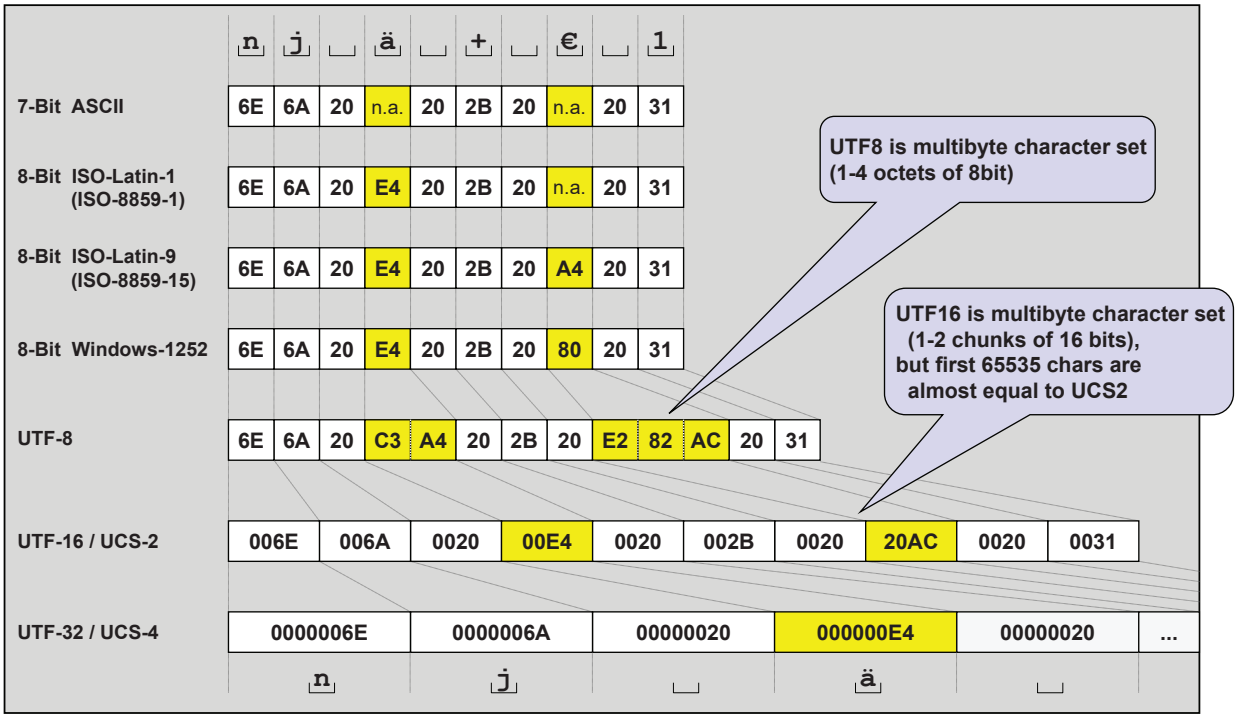
Empty Base Class Optimization
- Clumsy and does not always work

Consequence:
Same address for members `e` and `i`

Visual C++ (also) needs: `[[msvc::no_unique_address]]`

Different Characters Sets and Encodings

C++

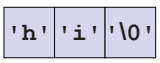


Type of String Literals

C++11/C++17/C++20

- **String literals**
  - are arrays of constant characters with a null terminator at the end

```
"hi"    // type: const char[3]
```



  - Support binary/special characters
  - Support Unicode characters (since C++11)
  - Can have different encodings
  - Can have different types (specific UTF-8 character type supported since C++20)
  - Can be specified by using a raw string syntax (since C++11)

```
auto s1 = L"hello";          // wide character sequence (using wchar_t, all versions)
auto s2 = u"hello";          // UTF-16 characters (using char16_t, since C++11)
auto s3 = U"hello";          // UTF-32 characters (using char32_t, since C++11)
auto s4 = u8"K\u00F6ln";     // UTF-8 characters (using char since C++11, char8_t since C++20)
auto c = u8'@';              // UTF-8 character byte (char in C++17, char8_t since C++20 )
auto s4 = R"(LB: "\n")";     // raw string, same as "LB: \"\\n\"" (since C++11)
```

C++20: UTF-8 Character and String Types

C++20

- **UTF-8 character and string types:**
  - `char8_t`
    - One byte of a UTF-8 character
  - `std::u8string`, `std::u8string_view`
    - Sequence of `char8_t` characters
    - Used now for functions Using/returning UTF-8 strings

```
char8_t c = u8'@';                  // character with UTF-8 encoding for character @
const char8_t* s = u8"K\u00F6ln";   // character with UTF-8 encoding for Köln
auto s1 = u8"K\u00F6ln";            // s1 is const char8_t* since C++20 (const char* before)

using namespace std::literals;
auto s2 = u8"K\u00F6ln"s;           // s2 is std::u8string since C++20 (std::string before)
auto s3 = u8"K\u00F6ln"sv;          // s3 is std::u8string_view since C++20
```


C++20: Unicode and UTF-8

C++20

	Unicode	Name	String	Enc.	Code Units	Type
ä	U+00E4	LATIN SMALL LETTER A WITH DIAERESIS	`u8"\u00E4"`	UTF-8	0xC3 0xA4 0x00	`const char[3]`
				since C++20:		`const char8_t[3]`
			`u"\u00E4"`	UTF-16	0x00E4 0x0000	`const char16_t[2]`
			`U"\u00E4"`	UTF-32	0x000000E4 0x00000000	`const char32_t[2]`
€	U+20AC	EURO SIGN	`u8"\u20AC"`	UTF-8	0xE2 0x82 0xAC 0x00	`const char[4]`
				since C++20:		`const char8_t[4]`
			`u"\u20AC"`	UTF-16	0x20AC 0x0000	`const char16_t[2]`
			`U"\u20AC"`	UTF-32	0x000020AC 0x00000000	`const char32_t[2]`

```
char c1 = '\u0061';                 // since C++98
char c2 = u8'\u0061';               // since C++17
char8_t c3 = u8'\u0061';            // since C++20
auto c4 = u8'\u00E4';               // ERROR (all versions): more than one code unit

auto s = u8"K\u00F6ln";             // const char* until C++17, const char8_t* since C++20
std::cout << "s: " << s << '\n';    // deleted since C++20
```

See http://wg21.link/p1423 and http://wg21.link/p2513 for hints about compatibility

C++20: Dealing with char8_t

C++20

```
std::string s0 = u8"text";   // OK in C++17, ERROR since C++20

auto s = u8"K\u00F6ln";      // s is const char* until C++17, but const char8_t* since C++20
const char* s2 = s;          // OK in C++17, ERROR since C++20
#ifndef __cpp_char8_t
std::cout << s;              // OK in C++17, ERROR since C++20
#else
std::cout << reinterpret_cast<const char*>(s);  // OK
#endif

auto c = u8'c';              // c1 is char in C++17, but char8_t since C++20
char c2 = c;                 // OK (even if char8_t)
char* cp = &c;               // OK in C++17, ERROR since C++20
#ifndef __cpp_char8_t
std::cout << c;              // OK in C++17, ERROR since C++20
#else
std::cout << static_cast<char>(c);     // OK
#endif

for (const auto& file : fs::directory_iterator(path)) {
  auto name = file.path().u8string(); // std::string in C++17, std::u8string since C++20
  ...
}
```

See http://wg21.link/p1423 for other ways to deal with compatibility issues

__cplusplus

- **The preprocessor constant `__cplusplus` signals C++ and which version of it is supported**
  - Due to soft migrations it is difficult to rely on its value (vendors might change it very early or very late)
- **Values:**
  - For C++98 and C++03: `199711L`
  - For C++11: `201103L`
  - For C++14: `201402L`
  - For C++17: `201703L`
  - For C++20: `202002L`

Visual Studio 2017
- still has `199711` in all modes
- Use `/Zc:__cplusplus` for correct value

Formally introduced with C++20

- **Feature test macros allow more portable checks**
  - http://isocpp.org/std/standing-documents/sd-6-sg10-feature-test-recommendations

C++20: Using Feature Test Macros

C++20

```
class MyType {
 private:
  int value;
 public:
  MyType(int i)      // implicit constructor from int
   : value{i} {
  }
  bool operator==(const MyType& rhs) const {   // enables MyType == int
    return value == rhs.value;
  }
};

#ifndef __cpp_impl_three_way_comparison
bool operator==(int i, const MyType& t) {
  return t == i;    // calls member function
}
#endif

MyType x = 42;
if (x == 0) ...       // OK until C++17, since C++20: endless recursion
if (0 == x) ...       // OK
```

Feature test macros:

- Macros defined for each language and library feature
- May have different versions

http://stackoverflow.com/questions/65648897/c20-behaviour-breaking-existing-code-with-equality-operator

Feature Test Macros

C++20

- **To be able to use new features only if supported**
  - For language and library features
  - Might have different values for different versions of behavior
  - http://isocpp.org/std/standing-documents/sd-6-sg10-feature-test-recommendations

__cpp_generic_lambdas	201304	[N3649] Generic (Polymorphic) Lambda Expressions (Revision 3)
	201707	[P0428R2] Familiar template syntax for generic lambdas

```
#ifndef __cpp_lib_as_const
template<typename T>
const T& asConst(T& t) {
  return t;
}
#endif

#ifdef __cpp_lib_as_const
  auto printColl = [&coll = std::as_const(coll)] {
#else
  auto printColl = [&coll = asConst(coll)] {
#endif
                     ...
                   };
```

C++20

Other Library Features

C++20: std::source_location

C++20

- std::source_location
  - Typed API for locations in the source code
  - Replaces the preprocessor macros
    __FILE__, __LINE__, __func__
  - std::source_location::current() yields
    - Location where it is called in normal code
    - Location where a function is called if used as default argument

```
#include <source_location>

int main()
{
  std::source_location loc = std::source_location::current();
  std::cout << loc.file_name() << ": " << loc.line() << '\n';
  std::cout << loc.function_name()
            << " column: " << loc.column() << '\n';
}
```

Possible output:

```
prog.cpp: 6
int main() column: 59
```

C++20: std::source_location

C++20

```
#include <source_location>

void foo()
{
  auto sl = std::source_location::current();
  …
  std::cout << "file:     " << sl.file_name() << '\n';
  std::cout << "function: " << sl.function_name() << '\n';
  std::cout << "line/col: " << sl.line() << '/' << sl.column() << '\n';
}

void bar(std::source_location sl = std::source_location::current())
{
  std::cout << "file:     " << sl.file_name() << '\n';
  std::cout << "function: " << sl.function_name() << '\n';
  std::cout << "line/col: " << sl.line() << '/' << sl.column() << '\n';
}

int main()
{
  foo();
  bar();
}
```

Output (gcc):

```
file:     sourceloc.cpp
function: void foo()
line/col: 8/42

file:     sourceloc.cpp
function: int main()
line/col: 34/6
```

Output (VC++):

```
file:     sourceloc.cpp
function: foo
line/col: 8/35

file:     sourceloc.cpp
function: main
line/col: 34/3
```


String Updates

C++20

- `reserve()` can no longer shrink memory
  - Calling `reserve()` without any argument is no longer valid
- New member functions (also for string view's):
  - `starts_with()`
  - `ends_with()`
- Support for `operator<=>`

```
void foo(const std::string& s, std::string_view suffix)
{
  if (s.starts_with('.')) {
    …
  }
  if (s.ends_with(".tmp")) {
    …
  }
  if (s.ends_with(suffix)) {
    …
  }
}
```

ssize() C++20

- std::ssize() and std::ranges::ssize()
  - to avoid warning/errors because an unsigned size is used

```
std::vector<int> coll;
...

for (int i = 0; i < coll.size(); ++i) {
  ...
}
```

Warning:
comparison of integer expressions of different signedness: 'int' and 'std::vector<int>::size_type' {aka 'long unsigned int'}

```
for (int i = 0; i < std::ssize(coll); ++i) {   // usually no warning
  ...
}

for (int i = 0; i < ssize(coll); ++i) {   // OK due to argument-dependent lookup
  ...
}

std::latch l1{coll.size()};                // ERROR: latch needs ptrdiff_t
std::latch l2{ssize(coll)};                // OK
```


C++20: Safe Comparison of Integral Values C++20

- **Safe comparisons of two integral values of different types**
  - In header <utility>

Function	Effect
std::cmp_equal(x, y)	Yields whether x == y
std::cmp_not_equal(x, y)	Yields whether x != y
std::cmp_less(x, y)	Yields whether x < y
std::cmp_less_or_equal(x, y)	Yields whether x <= y
std::cmp_greater(x, y)	Yields whether x > y
std::cmp_greater_or_equal(x, y)	Yields whether x >= y
std::in_range<*Type*>(x)	Yields whether x is a valid value of *Type*

```
int x = -7;
unsigned y = 42;

if (x < y) ...                          // OOPS: false
if (x < static_cast<int>(y)) ...        // true
if (std::cmp_less(x, y)) ...            // true
if (std::in_range<unsigned>(x) ) ...    // false
```

converts x to unsigned long

C++20: Math Constants

C++20

- **Nearest representable floating-point values**
  - In header **<numbers>**

Constant	double Instantiation of
std::numbers::e	std::numbers::e_v<*type*>
std::numbers::pi	std::numbers::pi_v<*type*>
std::numbers::inv_pi	std::numbers::inv_pi_v<*type*>
std::numbers::inv_sqrtpi	std::numbers::inv_sqrtpi_v<*type*>
std::numbers::sqrt2	std::numbers::sqrt2_v<*type*>
std::numbers::sqrt3	std::numbers::sqrt3_v<*type*>
std::numbers::inv_sqrt3	std::numbers::inv_sqrt3_v<*type*>
std::numbers::log2e	std::numbers::log2e_v<*type*>
std::numbers::log10e	std::numbers::log10e_v<*type*>
std::numbers::ln2	std::numbers::ln2_v<*type*>
std::numbers::ln10	std::numbers::ln10_v<*type*>
std::numbers::egamma	std::numbers::egamma_v<*type*>
std::numbers::phi	std::numbers::phi_v<*type*>

```
double area1 = std::numbers::pi * rad * rad;
auto   area2 = std::numbers::pi_v<long double> * rad * rad;
```


C++20: Bit Cast

C++20

- **std::bit_cast<*type*>(*value*)**
  - Cast of a value to a type of same size
  - Safer than **reinterpret_cast<>()** and **union**'s
    - Number of bits has to fit
    - Only for standard layout types
    - Not for pointer types

```
#include <bit>
…
struct S { int val; };
S s{11};

int i;
i = static_cast<int>(s);            // ERROR (no static type relation)
i = std::bit_cast<int>(s);          // OK
i = *reinterpret_cast<int*>(&s);    // OK

long l;
l = std::bit_cast<long>(s);         // ERROR unless sizeof(int) == sizeof(long)
l = *reinterpret_cast<long*>(&s);   // compiles, but usually fatal runtime ERROR
```

C++20: Bit Operations

C++20

- **Bit Operations**
  - Low-level command for bit instructions
  - For unsigned integral types (not `int`, `char`, or `std::byte`)

`std::rotl(val, n)`	Yields *val* with *n* bits rotated to the left
`std::rotr(val, n)`	Yields *val* with *n* bits rotated to the right
`std::countl_zero(val)`	Yields number of leading (most significant) `0` bits
`std::countl_one(val)`	Yields number of leading (most significant) `1` bits
`std::countr_zero(val)`	Yields number of leading (most significant) `0` bits
`std::countr_one(val)`	Yields number of leading (most significant) `1` bits
`std::popcount(val)`	Yields number of `1` bits
`std::has_single_bit(val)`	Yields whether *val* is a power of 2 (has exactly 1 bit set)
`std::bit_floor(val)`	Yields previous power-of-2 value (e.g., 8 for 13)
`std::bit_ceil(val)`	Yields next power-of-2 value (e.g., 16 for 13)
`std::bit_width(val)`	Yields number of bits necessary to store the value

C++20: Bit Operations

C++20

- **Bit Operations**
  - Low-level command for bit instructions
  - For unsigned integral types (not `int`, `char`, or `std::byte`)

```
#include <bit>
...

  std::uint8_t i8 = 0b0000'1101;
  std::cout
    << std::format("{0:08b} {0:3}\n", i8)
    << std::format("{0:08b} {0:3}\n", std::rotl(i8, 2))
    << std::format("{0:08b} {0:3}\n", std::rotr(i8, 1))
    << std::format("{0:08b} {0:3}\n", std::rotr(i8, -1))
    << std::format("{}\n", std::countl_zero(i8))
    << std::format("{}\n", std::countr_one(i8))
    << std::format("{}\n", std::popcount(i8))
    << std::format("{}\n", std::has_single_bit(i8))
    << std::format("{0:08b} {0:3}\n", std::bit_floor(i8))
    << std::format("{0:08b} {0:3}\n", std::bit_ceil(i8))
    << std::format("{}\n", std::bit_width(i8));
```

Output:

```
00001101  13
00110100  52
10000110 134
00011010  26
4
1
3
false
00001000   8
00010000  16
4
```

C++20: `std::endian`

C++20

- **`std::endian`**
  - Enumeration type for the endianness of the execution environment
    - **`std::endian::big`** value for most significant byte placed first
    - **`std::endian::little`** (value for least significant byte placed first
    - **`std::endian::native`** (endianness of the execution environment)
      - Either **`std::endian::big`** or **`std::endian::little`** or other value

```
#include <bit>
...

if constexpr (std::endian::native == std::endian::big) {
  ...    // handle big-endian
}
else if constexpr (std::endian::native == std::endian::little) {
  ...    // handle little-endian
}
else {
  ...    // handle mixed endian
}
```


C++20: New Type Traits

C++20

Trait	Meaning
remove_cvref_t<T>	Type without reference, const, volatile
is_bounded_array_v<T>	true if array with specified size
is_unbounded_array_v<T>	true if array without specified size
unwrap_reference_t<T>	referenced type of **std::ref()**, **std::cref()** or T
unwrap_ref_decay_t<T>	referenced type of **std::ref()**, **std::cref()** or decayed type of T
is_nothrow_convertible_v<T1, T2>	true if T1 converts to T2 without exception
common_reference_t<T1, T2, ...>	common type all types can assign to (reference if no temporary involved)
type_identity_t<T>	Type T as it is
is_layout_compatible_v<T1, T2>	Safe conversion of pointers with reinterpret_cast
is_layout_pointer_interconvertible_base_of_v<T1, T2>	Safe conversion of pointers to base pointers with reinterpret_cast
iter_difference_t<T>	Difference type of a pointer/iterator or incrementable type
iter_value_t<T>	Non-**`const`** value/element type to where T refers

Plus helpers for safe conversion of class members:

- **is_pointer_interconvertible_with class()**
- **is_corresponding_member()**

C++20: <version>

- **<version>**
  - Header to understand properties of the used standard library
    - No functionality
    - Short and fast to load
  - Implementation-dependent information such as:
    - Library version number
    - Library release date
    - Copyright notice
    - ...
  - Contains also all feature test macros of the standard library
    - They are also available in their specific headers

C++20

Compile-Time Computing

constexpr Functions

C++11

- constexpr
  - *Enables* **compile-time** evaluation of functions
  - Restricts what can happen in the function body (version specific)

```
int runtimeFunc(int x) {
  return x * x;
}

constexpr int constexprFunc(int x) {
  return x * x;
}
```

```
constexpr int foo(int x) {
  static int i = 11;  // ERROR
  return x * i;
}
```

```
// runtime context:
int x = 42;

std::cout << runtimeFunc(x);     // OK
std::cout << constexprFunc(x);   // OK

std::cout << runtimeFunc(7);     // OK
std::cout << constexprFunc(7);   // OK

// compile-time context:
int arr1[runtimeFunc(7)];        // ERROR
int arr2[constexprFunc(7)];      // OK
```

may be called at compile time

C++20: consteval Functions

C++20

- consteval
  - *Requires* **compile-time** evaluation of functions
  - Restricts what can happen in the function body (version specific)

```
int runtimeFunc(int x) {
  return x * x;
}

constexpr int constexprFunc(int x) {
  return x * x;
}

consteval int constevalFunc(int x) {
  return x * x;
}

consteval int foo(int x) {
  static int i = 11;  // ERROR
  return x * i;
}
```

```
// runtime context:
int x = 42;

std::cout << runtimeFunc(x);     // OK
std::cout << constexprFunc(x);   // OK
std::cout << constevalFunc(x);   // ERROR

std::cout << runtimeFunc(7);     // OK
std::cout << constexprFunc(7);   // OK
std::cout << constevalFunc(7);   // OK

// compile-time context:
int arr1[runtimeFunc(7)];        // ERROR
int arr2[constexprFunc(7)];      // OK
int arr3[constevalFunc(7)];      // OK
```

for sure called at compile time

Conditional Compile-time Functions with `consteval` C++20

```
int next() {
  static int max = 0;
  return ++max;
}

constexpr int fooCR(bool truth) {
  if (truth) {
    return 42;
  }
  else {
    return next(); // checked when used
  }
}

consteval int fooC(bool truth) {
  if (truth) {
    return 42;
  }
  else {
    return next(); // checked when used
  }
}
```

```
int x = 42;
bool b = true;

std::cout << fooCR(true);     // OK
std::cout << fooCR(false);    // OK
std::cout << fooCR(b);        // OK

std::cout << fooC(true);      // OK
std::cout << fooC(false);     // ERROR
std::cout << fooC(b);         // ERROR

constexpr int i1 = fooCR(true);  // OK
constexpr int i2 = fooCR(false); // ERROR
constexpr int i3 = fooCR(b);     // ERROR

constexpr int i4 = fooC(true);   // OK
constexpr int i5 = fooC(false);  // ERROR
constexpr int i6 = fooC(b);      // ERROR
```


Conditional Compile-time Functions with `consteval` C++20

```
int next() {
  static int max = 0;
  return ++max;
}

constexpr int fooCR(bool truth) {
  if (truth) {
    return 42;
  }
  else {
    return next(); // checked when used
  }
}

consteval int fooC(bool truth) {
  if (truth) {
    return 42;
  }
  else {
    return next(); // checked when used
  }
}
```

```
int x = 42;
constexpr bool b = true;

std::cout << fooCR(true);     // OK
std::cout << fooCR(false);    // OK
std::cout << fooCR(b);        // OK

std::cout << fooC(true);      // OK
std::cout << fooC(false);     // ERROR
std::cout << fooC(b);         // ERROR

constexpr int i1 = fooCR(true);  // OK
constexpr int i2 = fooCR(false); // ERROR
constexpr int i3 = fooCR(b);     // OK

constexpr int i4 = fooC(true);   // OK
constexpr int i5 = fooC(false);  // ERROR
constexpr int i6 = fooC(b);      // OK
```

C++20: Using Conditional Compile-time Functions

C++20

```
void forceCompileTimeError()
{
}

consteval std::size_t hashed(const char* str)
{
  if (str[0] == '\0' || str[1] == '\0') {   // needs at least two characters
    forceCompileTimeError();
  }
  std::size_t hash = 5381;   // see http://www.cse.yorku.ca/~oz/hash.html (djb2)
  while (*str != '\0') {
    hash = hash * 33 ^ *str++;
  }
  return hash;
}
```

```
auto h1 = hashed("otto");    // OK yields 6382897381
std::cout << h1 << '\n';
auto h2 = hashed("toto");    // OK yields 6382929925
std::cout << h2 << '\n';
auto h3 = hashed("a");       // Compile-time ERROR
```


Runtime Context in Compile-time Functions

C++20

- **Compile-time functions still follow runtime rules**
  - A parameter does not become a compile-time value

```
constexpr int returnSize() {
  return 42;
}

consteval auto computeSize(int sz)
{
  std::array<int, sz> arr1{};     // ERROR

  auto sz1 = returnSize();
  std::array<int, sz1> arr2{};    // ERROR

  constexpr auto sz2 = returnSize();
  std::array<int, sz2> arr3{};    // OK

  return sz;
}

std::array<int, computeSize(10)> arr{};
```

C++20: `std::is_constant_evaluated()`

C++20

- **`std::is_constant_evaluated()` yields `true` if**
  - constant-expression or
  - in constant context (in **`if constexpr`**, **`consteval`**, constant initialization)
  - initializer of a variable usable at compile time

```
#include <type_traits>

constexpr int len(const char* s) {
  if (std::is_constant_evaluated()) {
    int idx = 0;
    while (s[idx] != '\0') {        // compile-time friendly code
      ++idx;
    }
    return idx;
  }
  else {
    return std::strlen(s);          // function called at runtime
  }
}

int l1 = len("hello");              // no required compile-time context => uses else branch
constexpr int l2 = len("hello");    // uses then branch
```

Within `if constexpr` and in `consteval` always true (http://wg21.link/p1938)

C++20: Example of `consteval` Lambdas with NTTP

C++20

```
auto isPrime = [](int value) constexpr {
                 for (int i = 2; i <= value/2; ++i) {
                   if (value % i == 0) {
                     return false;
                   }
                 }
                 return value > 1;   // 0 and 1 are no prime numbers
               };

// local compile-time computation of Num prime numbers:
auto primeNumbers = [isPrime] <int Num> () consteval {
                      std::array<int, Num> primes;
                      int idx = 0;
                      for (int val = 1; idx < Num; ++val) {
                        if (isPrime(val)) {
                          primes[idx++] = val;
                        }
                      }
                      return primes;
                    };

// init array with prime numbers:
std::array primes = primeNumbers.operator()<100>();
```

C++20: Compile-Time Computing

C++20

- `constexpr` (and `consteval`) now allowed for
  - virtual functions
  - use `dynamic_cast<>`
  - using `typeid`
  - have `try`/`catch` block (but no `throw`)
    - simply ignored
  - allocate memory with `std::allocator`
  - change active member of a `union`

- `std::vector<>` and `std::string` are `constexpr` now
  - Use a container and have names at compile-time
    - For reflection (list and names of members, attributes, ...) in C++23
  - Can't use compile-time vector/string at runtime (yet)

C++20: Compile-Time Vectors and Strings

C++20

- **Constraints for complex compile-time types:**
  - The objects may only live at compile time
    - Create and destroy at compile-time
    - The types need a `constexpr` destructor
  - Only allocators using standard memory handling can be used
    - Calling `std::allocator<T>::allocate()` and `std::allocator<T>::deallocate()`
- **Vectors and strings can now be used at compile time**
- **Note:**
  - `static_assert()` can't be called for compile-time strings
    - Requires string literals

```
constexpr std::string s1{"short"};                      // may compile with compile-time SSO
constexpr std::string s2{"a long string without SSO"}; // ERROR
constexpr auto sz = std::string{"hello"}.size();        // OK
```

Merge a List of Sizes

C++20

```
template<std::ranges::input_range C, std::integral... Types>
constexpr auto mergeSizes(const C& rg, Types&&... vals)
{
  // initialize vector with passed sizes:
  std::vector<std::ranges::range_value_t<C>> v{std::ranges::begin(rg),
                                               std::ranges::end(rg)};
  (... , v.push_back(std::forward<Types>(vals)));   // merge passed sizes

  std::ranges::sort(v);                             // sort sizes

  return v;                                         // return all sizes
}
```

```
constexpr std::array a{sizeof(int), sizeof(std::string), sizeof(std::set<int>)};
...
// initialize sorted collection of all sizes:
auto allSizes = mergeSizes(a, sizeof(long), sizeof(Coord));
for(const auto& i : allSizes) {
  std::cout << i << ' ';
}
```

Possible Output:

```
4 32 24 + 8 24
4 8 24 24 32
```

`mergeSizes()` still evaluated at runtime because the result is used at runtime

Merge a List of Sizes at Compile-Time

C++20

```
template<std::ranges::input_range C, std::integral... Types>
consteval auto mergeSizes(const C& rg, Types&&... vals)
{
  // initialize vector with passed sizes:
  std::vector<std::ranges::range_value_t<C>> v{std::ranges::begin(rg),
                                               std::ranges::end(rg)};
  (... , v.push_back(std::forward<Types>(vals)));   // merge passed sizes

  std::ranges::sort(v);                             // sort sizes

  return v;                                         // return all sizes
}
```

```
constexpr std::array a{sizeof(int), sizeof(std::string), sizeof(std::set<int>)};
...
// initialize sorted collection of all sizes:
auto allSizes = mergeSizes(a, sizeof(long), sizeof(Coord));   // ERROR
for(const auto& i : allSizes) {
  std::cout << i << ' ';
}
```

Cannot use a compile-time vector at runtime

Merge a List of Sizes at Compile-Time

C++20

```
template<std::ranges::input_range C, std::integral... Types>
consteval auto mergeSizes(const C& rg, Types&&... vals)
{
  // initialize vector with passed sizes:
  std::vector<std::ranges::range_value_t<C>> v{std::ranges::begin(rg),
                                               std::ranges::end(rg)};
  (... , v.push_back(std::forward<Types>(vals)));   // merge passed sizes

  std::ranges::sort(v);                             // sort sizes

  // return merges sizes as std::array<>:
  auto sz = std::ranges::size(rg) + sizeof...(vals);
  std::array<std::ranges::range_value_t<C>, sz> ret{};    // ERROR

}
```

"*runtime context*" for `sz` in `std::array<..., sz>`

```
constexpr std::array a{sizeof(int), sizeof(std::string), sizeof(std::set<int>)};
...
// initialize sorted collection of all sizes:
auto allSizes = mergeSizes(a, sizeof(long), sizeof(Coord));
for(const auto& i : allSizes) {
  std::cout << i << ' ';
}
```


Merge a List of Sizes at Compile-Time

C++20

```
template<std::ranges::input_range C, std::integral... Types>
consteval auto mergeSizes(const C& rg, Types&&... vals)
{
  // initialize vector with passed sizes:
  std::vector<std::ranges::range_value_t<C>> v{std::ranges::begin(rg),
                                               std::ranges::end(rg)};
  (... , v.push_back(std::forward<Types>(vals)));   // merge passed sizes

  std::ranges::sort(v);                             // sort sizes

  // return merges sizes as std::array<>:
  constexpr auto sz = std::ranges::size(rg) + sizeof...(vals);
  std::array<std::ranges::range_value_t<C>, sz> ret{};    // ERROR

}
```

"*runtime context*" for `size(rg)` because `rg` is passed by reference

```
constexpr std::array a{sizeof(int), sizeof(std::string), sizeof(std::set<int>)};
...
// initialize sorted collection of all sizes:
auto allSizes = mergeSizes(a, sizeof(long), sizeof(Coord));
for(const auto& i : allSizes) {
  std::cout << i << ' ';
}
```

Merge a List of Sizes at Compile-Time

C++20

```
template<std::ranges::input_range C, std::integral... Types>
consteval auto mergeSizes(C rg, Types&&... vals)
{
  // initialize vector with passed sizes:
  std::vector<std::ranges::range_value_t<C>> v{std::ranges::begin(rg),
                                               std::ranges::end(rg)};
  (... , v.push_back(std::forward<Types>(vals)));  // merge passed sizes

  std::ranges::sort(v);                            // sort sizes

  // return merges sizes as std::array<>:
  constexpr auto sz = std::ranges::size(rg) + sizeof...(vals);
  std::array<std::ranges::range_value_t<C>, sz> ret{};   // OK
  std::ranges::copy(v, ret.begin());
  return ret;
}

constexpr std::array a{sizeof(int), sizeof(std::string), sizeof(std::set<int>)};
...
// initialize sorted collection of all sizes:
auto allSizes = mergeSizes(a, sizeof(long), sizeof(Coord));   // OK
for(const auto& i : allSizes) {
  std::cout << i << ' ';
}
```

Possible Output:

```
4 8 24 24 32
```


Merge a List of Sizes at Compile-Time

C++20

```
template<std::ranges::input_range C, std::integral... Tyr
consteval auto mergeSizes(C rg, Types... vals)
{
  // initialize vector with passed sizes:
  std::vector<std::ranges::range_value_t<C>> v{std::ranges::begin(rg),
                                               std::ranges::end(rg)};
  (... , v.push_back(vals));  // merge passed sizes

  std::ranges::sort(v);       // sort sizes

  // return merges sizes as std::array<>:
  constexpr auto sz = std::ranges::size(rg) + sizeof...(vals);
  std::array<std::ranges::range_value_t<C>, sz> ret{};
  std::ranges::copy(v, ret.begin());
  return ret;
}

constexpr std::array a{sizeof(int), sizeof(std::string), sizeof(std::set<int>)};
...
// initialize sorted collection of all sizes:
auto allSizes = mergeSizes(a, sizeof(long), sizeof(Coord));   // OK
for(const auto& i : allSizes) {
  std::cout << i << ' ';
}
```

Pass compile-time parameters by value

Possible Output:

```
4 8 24 24 32
```

Merge a List of Sizes at Compile-Time

C++20

```
template<std::ranges::input_range C, std::integral... Types>
consteval auto mergeSizes(C rg, Types... vals)
{
  // initialize vector with passed sizes:
  std::vector<std::ranges::range_value_t<C>> v{std::ranges::begin(rg),
                                               std::ranges::end(rg)};
  (... , v.push_back(vals));   // merge passed sizes

  std::ranges::sort(v);        // sort sizes

  // return merges sizes as std::array<>:
  constexpr auto sz = std::ranges::size(rg) + sizeof...(vals);
  std::array<std::ranges::range_value_t<C>, sz> ret{};
  std::ranges::copy(v, ret.begin());
  return ret;
}
```

```
constexpr std::array a{sizeof(int), sizeof(std::string), sizeof(std::set<int>)};
...
// initialize sorted collection of all sizes:
auto allSizes = mergeSizes(a, sizeof(long), sizeof(Coord));    // OK
for(const auto& i : allSizes) {
  std::cout << i << ' ';
}
```

Possible Output:

```
4 8 24 24 32
```


Merge a List of Sizes at Compile-Time

C++20

```
template<std::ranges::input_range C, std::integral... Types>
consteval auto mergeSizes(C rg, Types... vals)
{
  // initialize vector with passed sizes:
  std::vector<std::ranges::range_value_t<C>> v{std::ranges::begin(rg),
                                               std::ranges::end(rg)};
  (... , v.push_back(vals));   // merge passed sizes

  std::ranges::sort(v);        // sort sizes

  // return merges sizes as std::array<> plus the computed number of elements:
  constexpr auto maxSz = std::ranges::size(rg) + sizeof...(vals);
  std::array<std::ranges::range_value_t<C>, maxSz> ret{};
  auto res = std::ranges::unique_copy(v, ret.begin());   // remove duplicates
  return std::pair{ret, res.out - ret.begin()};          // return array and size
}
```

Error if no {}

```
constexpr std::array a{sizeof(int), sizeof(std::string), sizeof(std::set<int>)};
...
// initialize sorted collection of all sizes:
auto [arr,sz] = mergeSizes(a, sizeof(long), sizeof(Coord));
auto allSizes = std::views::counted(arr.begin(), sz);
for(const auto& i : allSizes) {
  std::cout << i << ' ';
}
```

Possible Output:

```
4 8 24 32
```

C++20: constinit

C++20

- **constinit**
  - Requires compile-time initialization for global and static objects
    - Does **not** imply **const** (in contrast to **constexpr**)
    - Does **not** imply **inline** (in contrast to **constexpr**)

constinit = constexpr - const

```
constexpr int max = 42;          // initialized at compile-time, but const
constinit int act = max;         // initialized at compile-time, not const

void foo()
{
  std::cout << max << '\n';      // 42
  ++max;                         // ERROR
  std::cout << act << '\n';      // 42
  ++act;                         // OK
  std::cout << act << '\n';      // 43
}

class MyType {
  inline static thread_local constinit long max = sizeof(int) * 1000;
  ...
};
```

inline, static, static, and constinit may have any order

C++20: constinit

C++20

- **constinit**
  - Requires compile-time initialization for global and static objects
    - Does **not** imply **const** (in contrast to **constexpr**)
    - Does **not** imply **inline** (in contrast to **constexpr**)

```
constexpr std::array<int, 5> getTestColl() {
  return {1, 2, 3, 4, 5};
}
constinit auto globColl = getTestColl();   // OK

void foo() {
  std::cout << globColl[0] << '\n';        // prints 1 (first element for globalColl)
  globColl = {};                           // reset globalColl
  std::cout << globColl[0] << '\n';        // prints 0 (first element for globalColl)
}

int getNextId() {
  static constinit IdType maxId = 0;       // constructor has to be constexpr
  static constinit IdType val1 = maxId;    // ERROR: no static initializer
  constinit IdType val2 = 0;               // ERROR: not static or global
  return ++maxId;
}
```

`static` Initialization Order Fiasco

C++

truth.hpp:

```
struct Truth {
  int value;
  // initialize all objects with 42:
  Truth()
   : value{42} {
  }
};

// declare global object:
extern Truth theTruth;
```

truth.cpp:

```
#include "truth.hpp"

// define global object:
Truth theTruth;
```

main.cpp:

`val` may be initialized before `theTruth` is initialized

```
#include "truth.hpp"
#include <iostream>


static int val = theTruth.value;   // OOPS

int main()
{
  std::cout << val << '\n';   // may be 0 or 42
  ++val;
  std::cout << val << '\n';   // may be 1 or 43
}
```


`static` Initialization Order Fiasco and `constinit`

C++20

truth.hpp:

```
struct Truth {
  int value;
  // initialize all objects with 42:
  Truth()
   : value{42} {
  }
};

// declare global object:
extern Truth theTruth;
```

truth.cpp:

```
#include "truth.hpp"

// define global object:
Truth theTruth;
```

`constinit`
- Guarantees compile-time initialization
- Requires compile-time initializer

main.cpp:

```
#include "truth.hpp"
#include <iostream>

constinit
static int val = theTruth.value;   // ERROR

int main()
{
  std::cout << val << '\n';   //
  ++val;
  std::cout << val << '\n';   //
}
```

`static` Initialization Order Fiasco and `constinit` C++20

truth.hpp:

```
struct Truth {
  int value;
  // initialize all objects with 42:
  constexpr Truth()
   : value{42} {
  }
};

// declare global object:
constexpr Truth theTruth;
```

constinit
- Guarantees compile-time initialization
- Requires compile-time initializer

main.cpp:

```
#include "truth.hpp"
#include <iostream>

constinit
static int val = theTruth.value;   // OK

int main()
{
  std::cout << val << '\n';   // 42
  ++val;
  std::cout << val << '\n';   // 43
}
```


C++20

Other Generic Features

Template Parameters for Generic Lambdas (since C++20)

C++20

- **Generic lambdas can have template parameters**
  - to make only parts of a declaration generic
  - to have a name for the type
  - to have a default type for a default value

```
auto foo = [](const auto& vec) {                    // OOPS: for all containers
auto foo = [](const std::vector<auto>& vec) {       // ERROR: not allowed
auto foo = []<typename T>(const std::vector<T>& vec) {  // OK since C++20
             T tmp{};   // element type directly usable
             ...
           };
```

This is still a **function object** (not a function template)

```
// foo() callable only for vectors of arbitrary type:
foo(std::vector<int>{});            // OK
foo(std::vector<std::string>{});    // OK
foo(std::list<int>{});              // should not compile

auto print = []<typename T = int>(auto name, T val = {}) {
               ...
             };
```


Lambda Template Parameters vs. Variable Templates

C++14 C++20

- **Generic lambdas are function objects (functors) with a generic `operator()`:**

since C++20

```
auto primeNumbers = []<int Num>() {
                      std::array<int, Num> primes{};
                      ... // compute and assign first Num prime numbers
                      return primes;
                    };

// initialize array with the first 20 prime numbers:
auto primes20 = primeNumbers.operator()<20>();
```

- **Thinks about using a variable template instead:**

since C++14

```
template<int Num>
auto primeNumbers = []() {
                      std::array<int, Num> primes{};
                      ... // compute and assign first Num prime numbers
                      return primes;
                    };

// initialize array with the first 20 prime numbers:
auto primes20 = primeNumbers<20>();
```

C++20: Optional `typename` Qualification

C++20

- **Many needs for `typename` qualification are gone with C++20**
  - When declaring a return type (unless inside a function/block scope)
  - When declaring a data member
  - When declaring a parameter of a member function or **`requires`** expression
  - In a **`using`** declaration (alias definition)
  - …

```
template<typename T>
class MyClass {
  T values;
  typename T::size_type SizeT;
 public:
  using IteratorT = typename T::iterator;
  MyClass() = default;
  typename T::iterator begin() const;
  typename T::iterator end() const;
  void print(typename T::iterator);
};

template<typename T>
typename T::value_type foo(const T& rg, typename T::iterator pos) {
  typename T::value_type val;
  return val;
}
```


Keyword `typename`

C++98

a) Keyword for template declarations

- equivalent to **`class`**, but should be preferred

b) Specifies dependent names of template types as types

```
template<typename T>      // equivalent to:  template<class T>
class MyClass {
  …
  void foo() {
    T::value_type * a;    // if T::value_type is a type
                          // declaration of a as pointer to T::value_type
    …
    T::max * b;           // if T::max is not a type
                          // multiplication of T::max with b
  }
};
```

needs `typename`

Keyword `typename` C++98

a) Keyword for template declarations

– equivalent to `class`, but should be preferred

b) Specifies dependent names of template types as types

```
template<typename T>      // equivalent to:  template<class T>
class MyClass {
  ...
  void foo() {
    typename T::value_type * a;   // T::value_type is a type
                                  // => declaration of a as pointer to T::value_type
    ...
    T::max * b;                   // T::max is not a type
                                  // multiplication of T::max and b
  }
};

template<typename T>
void printElems(const T& v) {
  for (typename T::iterator pos = v.begin(); pos != v.end(); ++pos) {
      ...
  }
}
```

Typical usage

C++20: Conditional `explicit` C++20

- **Boolean compile-time condition for `explicit`**

```
template<typename T>
class Wrapper {
    T value;
  public:
    template<typename U>
    explicit(!std::is_convertible_v<U, T>) Wrapper(const U& val)
     :value{val} {
    }
};

// implicit conversion from string literal to string:
std::string s1{"hello"};
std::string s2 = "hello";                    // OK
Wrapper<std::string> ws1{"hello"};
Wrapper<std::string> ws2 = "hello";          // OK
printStringWrapper("hello");                 // OK

// only explicit conversion from size to vector<string>:
std::vector<std::string> v1{42u};            // OK
std::vector<std::string> v2 = 42u;           // ERROR: explicit
Wrapper<std::vector<std::string>> wv1{42u};
Wrapper<std::vector<std::string>> wv2 = 4u2; // ERROR: explicit
printVectorWrapper(42u);                     // ERROR: explicit
```

Support implicit conversions only from a type implicitly convertible to T

Remember: `explicit` disables
- parameter passing with implicit conversion
- copy initialization (with =)

C++20: Conditional `explicit`

C++20

- **Conditional `explicit` is used in the Standard Library**
  - **`std::pair`, `std::tuple`, `std::optional`, `std::span`, ...**

```
namespace std {
  template<typename T1, typename T2>
  struct pair {
    template <typename U1 = T1, typename U2 = T2>
    explicit(!std::is_convertible_v<U1, T1> || !std::is_convertible_v<U2, T2>)
    constexpr pair(U1&&, U2&&);
    ...
  };
}
```

Support implicit type conversions only for pairs of types that also have them

```
std::pair<int, int> p1{11, 11};

std::pair<long, long> p2{};
p2 = p1;                                                    // OK: implicit conversion from int to long

std::pair<std::chrono::day, std::chrono::month> p3{};
p3 = p1;                                                    // ERROR: only explicit from int to dates
p3 = std::pair<std::chrono::day, std::chrono::month>{p1}; // OK

std::map<std::string, std::string> coll1;
coll1.insert({"hi", "ho"});                                 // OK: uses implicit conversions to strings

std::map<std::string, std::chrono::month> coll2;
coll2.insert({"XI", 11});                                   // ERROR: no implicit conversion from int to month
coll2.insert({"XI", std::chrono::month{11}});               // OK (inserts element with string and month)
```


C++20: Conditional `explicit` for `std::span<>`

C++20

- **Conditional `explicit` is used in the Standard Library**
  - **`std::pair`, `std::tuple`, `std::optional`, `std::span`, ...**

```
namespace std {
  template<typename ElemType, size_t Extent = dynamic_extent>
  class span {
    ...
    template<typename It>
    constexpr explicit(extent != dynamic_extent) span(It beg, size_type count);
    template<typename It1, typename It2>
    constexpr explicit(extent != dynamic_extent) span(It1 beg, It2 end);
    ...
}
```

Support implicit type conversions only for spans with dynamic extent

```
void fooDyn(std::span<int>);
void fooFix(std::span<int, 3>);

std::vector coll{1, 2, 3, 4, 5};
fooDyn({coll.begin(), 3});                                  // OK
fooFix({coll.begin(), 3});                                  // ERROR: no implicit conversion supported

std::span<int> spDyn1{coll.begin(), 3};                     // OK: direct initialization
std::span<int> spDyn2 = {coll.begin(), 3};                  // OK: copy initialization with implicit conversion
std::span<int, 3> spFix1{coll.begin(), 3};                  // OK: direct initialization
std::span<int, 3> spFix2 = {coll.begin(), 3};               // ERROR: no implicit conversion supported
fooDyn(spFix1);                                             // OK
fooFix(spDyn1);                                             // ERROR: no implicit conversion supported
```

C++20

NTTP

Non-Type Template Parameter (NTTP) Types

C++98 / C++20

- **Supported types:**
  - Types for constant integral values (`int`, `long`, `enum`, ...)
  - `std::nullptr_t` (the type of `nullptr`)
  - Pointers to globally visible objects/functions/members
  - Lvalue references to objects or functions
- **Not supported are:**
  - String literals (directly)
  - Classes
- **Since C++20 supported are:**
  - Floating-point types (`float`, `double`, ...)
  - Data structures with public members
  - Lambdas

Non-Type Template Parameter Types since C++20

C++20

- **Non-type template parameters can now be values of**
  - Floating-point types
  - Data structures / classes (literal types with only public members)

```
template<double Val>
class MyDistribution {
  …
};

MyDistribution<0.0> md;          // OK since C++20


template<auto Val>
class MyClass {
  …
};

struct Data {
  int id;
  double value;
};
MyClass<Data{42, 7.7}> ms;       // OK since C++20

MyClass<"hi"> mlit;              // still ERROR in C++20
```


C++20: Non-Type Template Parameter `double`

C++20

```
template<double Val>
class MyClass {
  …
};
```

Could also use `template<auto Val>` (and support all NTTP types)

```
constexpr double third(double d) {
  return d / 3;
}

MyClass<41.7> obj;
std::is_same_v<MyClass<42.0>, MyClass<17.7>>              // false
std::is_same_v<MyClass<42.0>, MyClass<third(126.0)>>     // true or false (often true)
std::is_same_v<MyClass<42.7>, MyClass<third(128.1)>>     // true or false (often false)

std::is_same_v<MyClass<0.1 + 0.3 + 0.00001>,
               MyClass<0.3 + 0.1 + 0.00001>>             // true or false (often true)

std::is_same_v<MyClass<0.1 + 0.3 + 0.00001>,
               MyClass<0.00001 + 0.3 + 0.1>>             // true or false (often false)

std::is_same_v<MyClass<NAN>, MyClass<NAN>>       // true (although NAN==NAN yields false)
```

template-argument-equivalent if the values are **identical** floating-point values (operator == doesn't matter)

C++20: Structs as Non-Type Template Parameters

C++20

```
void print(const std::ranges::input_range auto& rg) {
  for (auto pos = std::ranges::begin(rg); pos != std::ranges::end(rg); ++pos) {
    std::cout << *pos << ' ';
  }
  std::cout << '\n';
}

template<auto Val>
struct EndValue {
  bool operator== (auto pos) const {
    return *pos == Val;   // Val is end value
  }
};

struct Coord {
  double x, y, z;
  auto operator<=> (const Coord&) const = default;   // enables all 6 comparison operators
  friend std::ostream& operator<< (std::ostream& strm, const Coord& c) {
    return strm << std::format("{}/{}/{}", c.x, c.y, c.z);
  }
};

std::array points{Coord{0,0,2}, Coord{0,2,0}, Coord{1,2,3}, Coord{4,5,6}};
print(points);
print(std::ranges::subrange{points.begin(), EndValue<Coord{4,5,6}>{}});
```

Literals types with public members
can be NTTP parameters

Output:

```
0/0/2 0/2/0 1/2/3 4/5/6
0/0/2 0/2/0 1/2/3
```


C++17: Non-Type Template Parameter `auto`

C++17

```
template<auto Val>
class MyClass {
  ...
};

constexpr int half(int i) {
  return i / 2;
}

MyClass<42> obj;
std::is_same_v<MyClass<42>, MyClass<17>>                    // false
std::is_same_v<MyClass<42>, MyClass<half(84)>>              // true

enum class Status { zero, one, two, none = zero};
bool operator==(Status, Status) {
  return true;
}

std::is_same_v<MyClass<Status::zero>, MyClass<Status::one>>   // false
std::is_same_v<MyClass<Status::zero>, MyClass<Status::none>>  // true
```

C++20: String Objects as Non-Type Template Parameters

C++20

- **Non-type template parameters can now be values of**
  - Floating-point types
  - Data structures / classes from string literals

```
template<auto Val>
class MyClass {
  …
};

template<std::size_t N>
struct Str {
  char chars[N];
  const char* value() const {
    return chars;
  }
  friend std::ostream& operator<< (std::ostream& strm, const Str& s) {
    return strm << s.chars;
  }
};
template<std::size_t N> Str(const char(&)[N]) -> Str<N>;   // deduction guide

MyClass<Str{"hi"}> msg;     // OK
```

e.g. for compile-time regex:
`Regex<Pattern{"[a-z][0-9a-z]*"}> rx;`

C++20: Lambdas as Non-Type Template Parameters

C++20

- **Non-type template parameters can now be values of**
  - Floating-point types
  - Data structures / classes with only public members
  - Lambdas

```
template<auto GetVat>
int addTax(int value)
{
  return static_cast<int>(std::round(value * (1 + GetVat())));
}

auto getTax = [] {
  return 0.19;
};

int main()
{
  std::cout << addTax<getTax>(100) << '\n';
  std::cout << addTax<getTax>(4199) << '\n';
}
```

Thank You!

Nicolai M. Josuttis

www.josuttis.com
nico@josuttis.com
@NicoJosuttis