

# Basic C++

C++20

Dr. Porkoláb Zoltán Károly

[gsd@inf.elte.hu](mailto:gsd@inf.elte.hu)

<http://gsd.web.elte.hu>

# Modules

- Using external code in our program
- Libraries
  - Java: `import org.apache.hadoop.*;`
  - Python: `import networkx as nx`
  - Fortran: `use opengl_gl`
  - C++: `#include <vector>`  
`#include <boost/program_options/cmdline.hpp>`  
`g++ a.o -lboost_program_options@1.71.0 -o a.out`

# C++ compilation model

- Separate translation
  - The compiler is always called on a single source file
- Several steps
  - Preprocessing
    - Locate include headers and transitively preprocessing ...
    - ... until no more preprocessor directives found
  - Translation unit / Input buffer is complete
  - Lexing → Parsing → Template instantiation ...
  - Code generation
  - Linking

# Token leak

```
// header.hpp
#define APP_DATE __DATE__ /* Build date */
// main.cpp
int main()
{
    std::cout << APP_DATE << '\n';
}

// lib.hpp
#define APP_DATE "2020.01.01" // Licensing date
// lib.cpp
const char *LicenseStartDate()
{
    return APP_DATE;
}

// main.cpp
#include "header.hpp"
int main()
{
    std::cout << LicenseStartDate() << APP_DATE << '\n';
}
```

# Name leak

```
// header1.hpp
namespace A {

    namespace {
        inline int detail() { return 1; }
    }
    class X { ... };
}
```

```
// header2.hpp
namespace A {

    namespace {
        inline int detail() { return 2; }
    }
    class Y { ... };
}
```

// What if we depend on both headers?

# Name leak

```
// client.cpp

struct B, D; // forward declaration to avoid parsing large headers

int f(const void * vp) { return 1; }
int f(const B* bp) { return 0; }

int test(D* dp)
{
    return f(dp); // f(const void*)
}
```

# Name leak

```
// d.h
```

```
struct D : B { ... };
```

```
// client.cpp  includes "d.h"
```

```
struct B, D;  // forward declaration to avoid parsing large headers
```

```
int f(const void * vp) { return 1; }
```

```
int f(const B* bp) { return 0; }
```

```
int test(D* dp)
{
    return f(dp);  // f(const B*)
}
```

```
// Google coding guidelines: never forward declare!
```

# Input buffer

- Names placed to the input buffer remains there forever
  - Issue with linking
  - Tool support
- Hiding
  - Static
  - Anonymous namespace
- Templates?



# Name leak

```
template <typename T>  
class X  
{  
private:  
    T foo(T t);  
};
```

X<int>{}.foo valid naming ?  
Can I call foo() ?

# Name leak

```
template <typename T>  
class X  
{  
private:  
    T foo(T t);  
};
```

X<int>{}.foo valid naming ?  
Can I call foo() ?

YES: X is defined, foo defined  
NO: foo is private

# Name leak

```
template <typename T>  
class X  
{  
private:  
    T foo(T t);  
};
```

X<int>{}.foo valid naming ?  
Can I call foo() ?

YES: X is defined, foo defined  
MAYBE: with some hack

# Name leak

```
static boost::regex rCppFiles{"\\.cpp\b"};

std::string sFilenameArgs{"/tmp/foo.cpp"};
boost::re_detail::matcher mBuf( sFilename.begin(), sFilename.end(),
    /* ... */
    &rCppFiles);

// no compiler error or warning using internal details
```

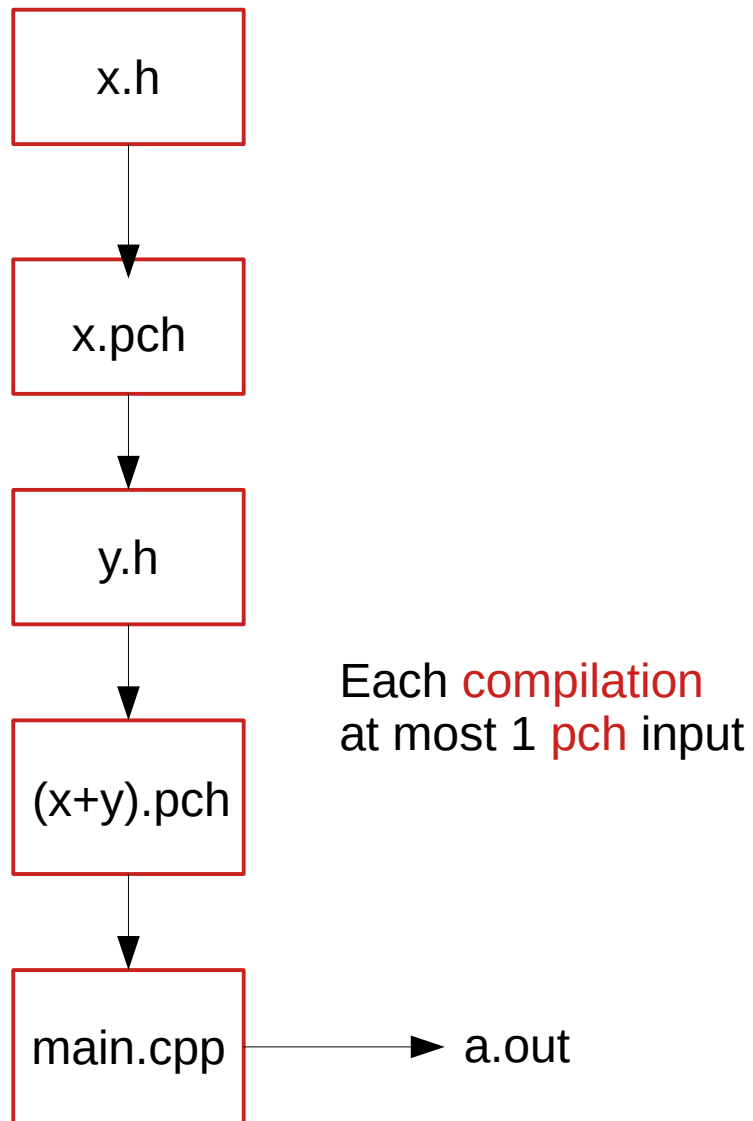
# Build times

- Include directives read from “storage”
  - Not defined by the standard, but usually from disk
- A large part of the input buffer is copy-paste
  - Usually 90-97% of the input buffer is coming from headers
- Unity build is usually possible only by manual work
  - Token leak
  - Name leak
- Templates everywhere
- Weak references are just waste resources

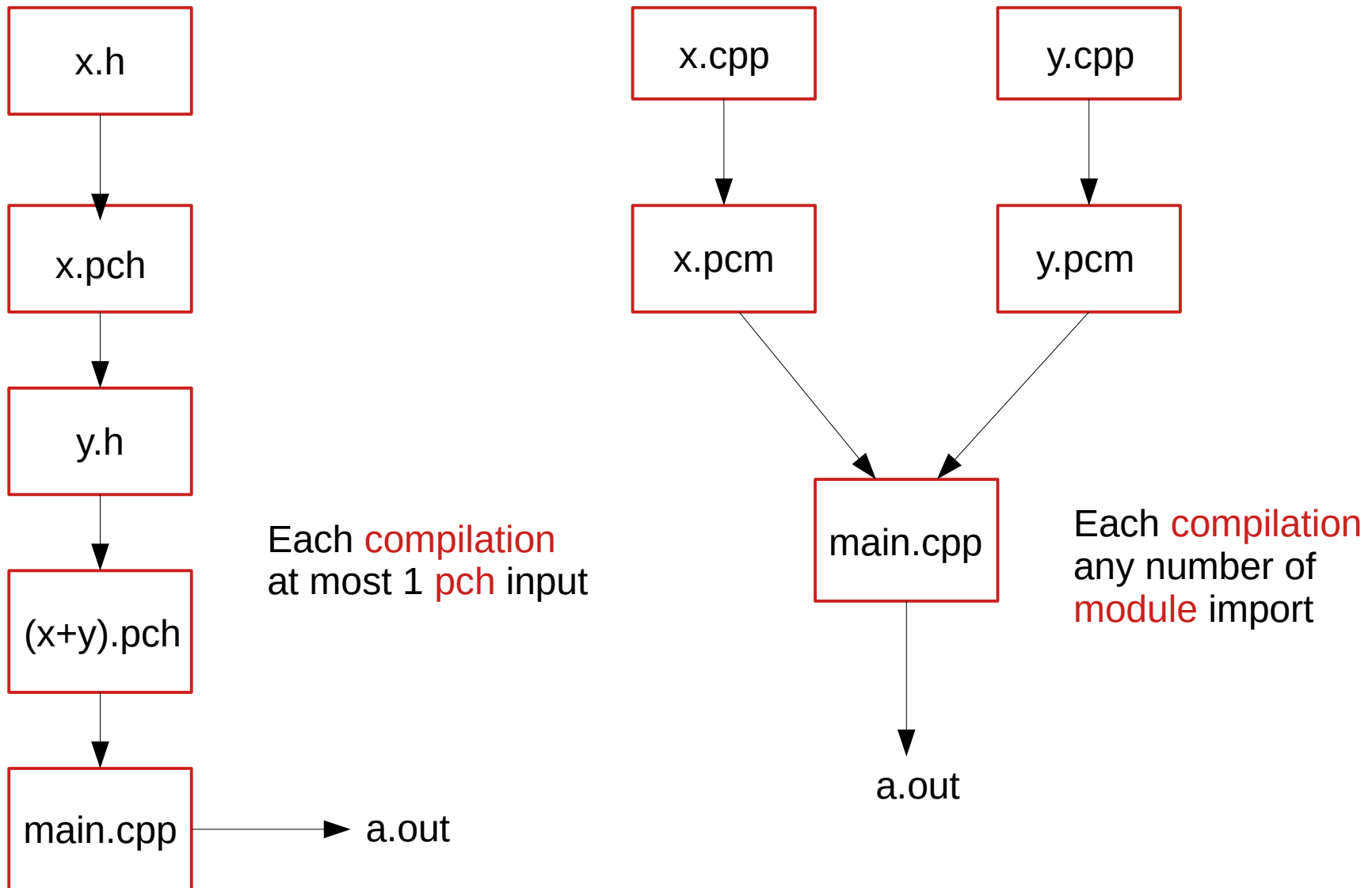
# Build times

- Precompiled headers (PCH)
  - Safe and reuse AST
  - PCH require build system support
- HP aCC compiler
  - Automatic generation of PCH

# PCH vs Modules



# PCH vs Modules





# Modules

- Originally proposed in 2004
- Part of C++20
- Header mechanism -> logical packaging
- One module is (still) a (few) translation units
- No preprocessor effect between TUs
- Name of the module is NOT part of fully qualified name
  - `import M; M::std::vector`
- Standard wording is flexible for optimizations

# Modules

- Module unit
  - Contains the module declaration
  - Module interface: export
  - Module implementation. No export keyword
- Module partition
  - Contains the “: <module-name>”
  - Can be interface or implementation partition
- Primary module interface unit (export + non-partition)  
`[export] module module-name [ : partition-name] [attrib-seq];`

# Modules

```
// speech.cpp
export module speech;

export const char* get_phrase_en() { return "Hello, world!"; }
export const char* get_phrase_hu() { return "Szia vilag!"; }
```

// possible partitions:

```
// speech.cpp
export module speech;
export import :english;
export import :hungarian;
```

```
// speech_e.cpp
export module speech:english;
export const char* get_phrase_en() { return "Hello, world!"; }
```

```
// speech_h.cpp
export module speech:hungarian;
export const char* get_phrase_hu() { return "Szia vilag!"; }
```

# Modules

```
// main.cpp
import <iostream>;
import speech;
int main() {
    std::cout << get_phrase_en() << get_phrase_hu() << '\n';
}

// possible partitions:

// speech.cpp
export module speech;
export import :english;
export import :hungarian;

// speech_e.cpp
export module speech:english;
export const char* get_phrase_en() { return "Hello, world!"; }

// speech_h.cpp
export module speech:hungarian;
export const char* get_phrase_hu() { return "Szia vilag!"; }
```

# Modules

```
// main.cpp      == client imports the primary module interface unit
import <iostream>;
import speech;
int main() {
    std::cout << get_phrase_en() << get_phrase_hu() << '\n';
}
```

// possible partitions:

```
// speech.cpp    == primary module interface unit
export module speech;
export import :english;      // the primary interface unit must export all
export import :hungarian;    // interface partitions, otherwise ill-formed
```

```
// speech_e.cpp  == module interface partition
export module speech:english;
export const char* get_phrase_en() { return "Hello, world!"; }
```

```
// speech_h.cpp  == module interface partition
export module speech:hungarian;
export const char* get_phrase_hu() { return "Szia vilag!"; }
```

# Modules - v2

```
// speech.cpp
export module speech;

import :english;
import :hungarian;    // interface partitions, otherwise ill-formed

export const char* get_phrase_en(); // must be declared to be part
export const char* get_phrase_hu(); // of the module interface


// speech_e.cpp == module implementation partition
module speech:english;
const char* get_phrase_en() { return "Hello, world!"; }

// speech_h.cpp == module implementation partition
module speech:hungarian;
const char* get_phrase_hu() { return "Szia vilag!"; }
```

# Modules - v3

```
// speech.cpp
export module speech;

export const char* get_phrase_en(); // must be declared to be part
export const char* get_phrase_hu(); // of the module interface


// speech_impl.cpp
module speech;

const char* get_phrase_en() { return "Hello, world!"; }
const char* get_phrase_hu() { return "Szia vilag!"; }
```

# Modules

```
// module.cpp
```

```
export module myModule;
```

```
    int four() { return 4; }  
export int five() { return four()+1; }    // but here four is available
```

```
// client.cpp
```

```
import MyModule;
```

```
int main()  
{  
    int i5 = five();  
    int i4 = four();    // compile error: no function named four()  
}
```

```
// Different from the private visibility: there is no name "four"
```



# Modules – templates

```
// template.cpp
export template <typename T>
struct foo
{
    T value;
    foo(T const v) : value(v) {}
};
export template <typename T>
foo<T> make_foo(T const value)
{
    return foo<T>(value);
}

// client.cpp
import <iostream>;
import <string>;
import foo;

int main()
{
    auto fi = make_foo(42);
    std::cout << fi.value << '\n';

    auto fs = make_foo(std::string("modules"));
    std::cout << fs.value << '\n';
}
```

# Modules

```
// module.cpp

export module myModule;

struct S // not exported
{
    S(int i) : m_(i) {}
    S(const S&) = delete;
    S(S&&) = default;

    int m_;
};

export S makeS() { return S{0}; } // factory method
```

# Modules

```
// module.cpp
```

```
export module myModule;
```

```
struct S // not exported
```

```
{
```

```
    S(int i) : m_(i) {}
```

```
    S(const S&) = delete;
```

```
    S(S&&) = default;
```

```
    int m_;
```

```
};
```

```
export S makeS() { return S{0}; } // factory method
```

```
// client.cpp
```

```
import MyModule;
```

```
int main()
```

```
{
```

```
}
```

# Modules

```
// module.cpp

export module myModule;

struct S                                // not exported
{
    S(int i)    : m_(i) {}
    S(const S&) = delete;
    S(S&& ) = default;

    int m_;
};

export S makeS() { return S{0}; } // factory method

// client.cpp

import MyModule;

int main()
{
    S    s1{1};                        // error: no type name S in current scope

}
```

# Modules

```
// module.cpp

export module myModule;

struct S // not exported
{
    S(int i) : m_(i) {}
    S(const S&) = delete;
    S(S&& ) = default;

    int m_;
};

export S makeS() { return S{0}; } // factory method

// client.cpp

import MyModule;

int main()
{
    S s1{1}; // error: no type name S in current scope
    auto s2 = makeS(); // ok
}
```

# Modules

```
// module.cpp
```

```
export module myModule;
```

```
struct S                                // not exported
{                                       // but reachable
    S(int i)    : m_(i) {}
    S(const S&) = delete;
    S(S&& ) = default;

```

```
    int m_;
};
export S makeS() { return S{0}; } // factory method
```

```
// client.cpp
```

```
import MyModule;
```

```
int main()
{
    S    s1{1};           // error: no type name S in current scope
    auto s2 = makeS();    // ok
    s2.m_ = 1;           // ok, works for anonymous types C++14
}
```

# Concepts

- Generic/Template: form of parametric polymorphism
- Constrained vs Unconstrained
- Java, ADA, Eiffel generics are constrained
  - Early error detection
  - Clear(er) error messages
- C++ templates are unconstrained (before C++20)
  - Duck typing (e.g. iterator adaptors)
  - Sometimes ugly error messages (but still in compile time)

# Concepts history

- 2000 First Workshop on C++ Template Programming
  - Jeremy Siek and Andrew Lumsdaine.  
[Concept Checking: Binding Parametric Polymorphism in C++](#) In Proceedings of the First Workshop on C++ Template Programming, Erfurt, Germany, 2000.
- Boost Concept Check Library
  - Jeremy Siek, Andrew Lumsdaine, David Abrahams.  
[https://www.boost.org/doc/libs/1\\_82\\_0/libs/concept\\_check/concept\\_check.htm](https://www.boost.org/doc/libs/1_82_0/libs/concept_check/concept_check.htm)
- OOPSLA 2003
  - R Garcia, J Jarvi, A Lumsdaine, JG Siek, J Willcock.  
A comparative study of language support for generic programming
- OOPSLA 2006
  - D Gregor, J Järvi, JG Siek, B Stroustrup, G Dos Reis, A Lumsdaine.  
Concepts: linguistic support for generic programming in C++  
ACM SIGPLAN Notices 41 (10), 291-310



# Concepts history

- 2009 No Concepts in C++0x
  - Stroustrup  
<https://www.accu.org/journals/overload/17/92/overload92.pdf#page=34>
- 2012 C++Now, best talk
  - Andrew Sutton: Concepts Lite: Constraining Templates with Predicates  
<https://www.youtube.com/watch?v=o1INd12uYjE>
- 2014 C++14 misses Concepts
- 2015 C++Now 2015 Keynote
  - Andrew Sutton: Generic Programming with Concepts  
[https://youtu.be/\\_rBhX-FJCdg](https://youtu.be/_rBhX-FJCdg)
- 2017 C++17 misses Concepts
- 2018 CppCon 2018
  - Andrew Sutton: Concepts in 60  
<https://www.youtube.com/watch?v=ZeU6OPaGxwM>
- 2020 Finally!

# Concepts basics

- Concept is a named predicate constraining template arguments
  - Syntax: operations, associated types
  - Semantics: how operations work
  - Complexity: operation performance
- Checking parameters at the point of they applies – instead of at the instantiation
- Therefore they give shorter and hopefully more direct diagnostics
- Possibility to specialize on concepts
- Example: ForwardIterator
  - `++i, i++, *i, i == j, i != j` // syntax
  - `i == j => ++i == ++j` `((void)[ ](auto x) { ++x; }(i), *i) == *i` // semantics:  
multipass
  - linear

# Concepts basics

- Concept is a named predicate constraining template arguments
  - Syntax: operations, associated types (checked by the compiler)
  - Semantics: how operations work (no check)
  - Complexity: operation performance (no check)
- Checking parameters at the point of they applies – instead of at the instantiation
- Therefore they give shorter and hopefully more direct diagnostics
- Possibility to specialize on concepts
- Example: ForwardIterator
  - `++i, i++, *i, i == j, i != j` // syntax
  - `i == j => ++i == ++j` `((void)[ ](auto x) { ++x; }(i), *i) == *i` // semantics:  
multipass
  - linear

# Concepts

- A type models a concepts
  - e.g. `char *` models `ForwardIterator`
  - `ForwardIterator<char *> && EqualityComparable<char>`
  - `ConvertibleTo<T, bool>`

// example from Sutton 2018

**template** <**typename** **Iter**>

```
Iter min_element(Iter first, Iter last)
{
    if ( first == last ) return first;
    Iter min = first;
    while ( ++first != last )
        if ( *first < *min ) min = first;
    return *min;
}
```

# Concepts

- A type models a concepts
  - e.g. `char *` models `ForwardIterator`
  - `ForwardIterator<char *> && EqualityComparable<char>`
  - `ConvertibleTo<T, bool>`
- **Iter**

```
// example from Sutton 2018
template <typename Iter>
```

```
Iter min_element(Iter first, Iter last)
{
    if ( first == last ) return first;
    Iter min = first;
    while ( ++first != last )
        if ( *first < *min ) min = first;
    return *min;
}
```

# Concepts

- A type models a concepts
  - e.g. `char *` models `ForwardIterator`
  - `ForwardIterator<char *> && EqualityComparable<char>`
  - `ConvertibleTo<T, bool>`
- **Iter**
  - Equality

```
// example from Sutton 2018
template <typename Iter>
```

```
Iter min_element(Iter first, Iter last)
{
    if ( first == last ) return first;
    Iter min = first;
    while ( ++first != last )
        if ( *first < *min ) min = first;
    return *min;
}
```

# Concepts

- A type models a concepts
  - e.g. `char *` models `ForwardIterator`
  - `ForwardIterator<char *> && EqualityComparable<char>`
  - `ConvertibleTo<T, bool>`
- **Iter**
  - Equality

```
// example from Sutton 2018
template <typename Iter>
```

```
Iter min_element(Iter first, Iter last)
{
    if ( first == last ) return first;
    Iter min = first;
    while ( ++first != last )
        if ( *first < *min ) min = first;
    return *min;
}
```

# Concepts

- A type models a concepts
  - e.g. `char *` models `ForwardIterator`
  - `ForwardIterator<char *> && EqualityComparable<char>`
  - `ConvertibleTo<T, bool>`

- **Iter**

- Equality
- Move constructible

```
// example from Sutton 2018
template <typename Iter>
```

```
Iter min_element(Iter first, Iter last)
{
    if ( first == last ) return first;
    Iter min = first;
    while ( ++first != last )
        if ( *first < *min ) min = first;
    return *min;
}
```



# Concepts

- A type models a concepts
  - e.g. `char *` models `ForwardIterator`
  - `ForwardIterator<char *> && EqualityComparable<char>`
  - `ConvertibleTo<T, bool>`

```
// example from Sutton 2018
template <typename Iter>
```

```
Iter min_element(Iter first, Iter last)
{
    if ( first == last ) return first;
    Iter min = first;
    while ( ++first != last )
        if ( *first < *min ) min = first;
    return *min;
}
```

- **Iter**
  - Equality
  - Move constructible
  - Copy constructible

# Concepts

- A type models a concepts
  - e.g. `char *` models `ForwardIterator`
  - `ForwardIterator<char *> && EqualityComparable<char>`
  - `ConvertibleTo<T, bool>`

// example from Sutton 2018

**template** <typename **Iter**>

```
Iter min_element(Iter first, Iter last)
{
    if ( first == last ) return first;
    Iter min = first;
    while ( ++first != last )
        if ( *first < *min ) min = first;
    return *min;
}
```

- **Iter**
  - Equality
  - Move constructible
  - Copy constructible
  - Incrementable

# Concepts

- A type models a concepts
  - e.g. `char *` models `ForwardIterator`
  - `ForwardIterator<char *> && EqualityComparable<char>`
  - `ConvertibleTo<T, bool>`

// example from Sutton 2018

**template** <typename **Iter**>

```
Iter min_element(Iter first, Iter last)
{
    if ( first == last ) return first;
    Iter min = first;
    while ( ++first != last )
        if ( *first < *min ) min = first;
    return *min;
}
```

- **Iter**
  - Equality
  - Move constructible
  - Copy constructible
  - Incrementable
  - Copy assignable

# Concepts

- A type models a concepts
  - e.g. `char *` models `ForwardIterator`
  - `ForwardIterator<char *> && EqualityComparable<char>`
  - `ConvertibleTo<T, bool>`

// example from Sutton 2018

**template** <typename **Iter**>

```
Iter min_element(Iter first, Iter last)
{
    if ( first == last ) return first;
    Iter min = first;
    while ( ++first != last )
        if ( *first < *min ) min = first;
    return *min;
}
```

- **Iter**
  - Equality
  - Move constructible
  - Copy constructible
  - Incrementable
  - Copy assignable
  - Dereferenceable

# Concepts

- A type models a concepts
  - e.g. `char *` models `ForwardIterator`
  - `ForwardIterator<char *> && EqualityComparable<char>`
  - `ConvertibleTo<T, bool>`

// example from Sutton 2018

**template** <typename **Iter**>

```
Iter min_element(Iter first, Iter last)
{
    if ( first == last ) return first;
    Iter min = first;
    while ( ++first != last )
        if ( *first < *min ) min = first;
    return *min;
}
```

- **Iter**
  - Equality
  - Move constructible
  - Copy constructible
  - Incrementable
  - Copy assignable
  - Dereferenceable
- **decltype(\*first)**
  - Ordered

# Concepts

- A type models a concepts
  - e.g. `char *` models `ForwardIterator`
  - `ForwardIterator<char *> && EqualityComparable<char>`
  - `ConvertibleTo<T, bool>`

// example from Sutton 2018

**template** <typename **Iter**>

```
Iter min_element(Iter first, Iter last)
{
    if ( first == last ) return first;
    Iter min = first;
    while ( ++first != last )
        if ( *first < *min ) min = first;
    return *min;
}
```

- **Iter**

- Equality
- Move constructible
- Copy constructible
- Copy assignable
- Incrementable
- Dereferenceable

- **decltype(\*first)**

- Ordered

# Concepts

- A type models a concepts
  - e.g. `char *` models `ForwardIterator`
  - `ForwardIterator<char *> && EqualityComparable<char>`
  - `ConvertibleTo<T, bool>`

```
// example from Sutton 2018
template <typename Iter>
```

```
Iter min_element(Iter first, Iter last)
{
    if ( first == last ) return first;
    Iter min = first;
    while ( ++first != last )
        if ( *first < *min ) min = first;
    return *min;
}
```

- **Iter**
  - Regular
  - Incrementable
  - Dereferenceable
- **decltype(\*first)**
  - Ordered

# Concepts

- A type models a concepts
  - e.g. `char *` models `ForwardIterator`
  - `ForwardIterator<char *> && EqualityComparable<char>`
  - `ConvertibleTo<T, bool>`

```
// example from Sutton 2018
template <typename Iter>
```

```
Iter min_element(Iter first, Iter last)
{
    if ( first == last ) return first;
    Iter min = first;
    while ( ++first != last )
        if ( *first < *min ) min = first;
    return *min;
}
```

- **Iter**
  - Regular

- Incrementable
- Dereferenceable

- **decltype(\*first)**
  - Ordered



# Concepts

- A type models a concepts
  - e.g. `char *` models `ForwardIterator`
  - `ForwardIterator<char *> && EqualityComparable<char>`
  - `ConvertibleTo<T, bool>`

- **Iter**

- Regular

// example from Sutton 2018

**template** <typename **Iter**>

```
Iter min_element(Iter first, Iter last)
{
    if ( first == last ) return first;
    Iter min = first;
    while ( ++first != last )
        if ( *first < *min ) min = first;
    return *min;
}
```

- `ForwardIterator`

- **decltype(\*first)**

- Ordered

# Concepts

- A type models a concepts
  - e.g. `char *` models `ForwardIterator`
  - `ForwardIterator<char *> && EqualityComparable<char>`
  - `ConvertibleTo<T, bool>`

- **Iter**

- Regular

// example from Sutton 2018

**template** <typename **Iter**>

```
Iter min_element(Iter first, Iter last)
{
    if ( first == last ) return first;
    Iter min = first;
    while ( ++first != last )
        if ( *first < *min ) min = first;
    return *min;
}
```

- `ForwardIterator`

- **decltype(\*first)**

- `Ordered`

# Concepts

- A type models a concepts
  - e.g. `char *` models `ForwardIterator`
  - `ForwardIterator<char *> && EqualityComparable<char>`
  - `ConvertibleTo<T, bool>`

- **Iter**
  - Regular

// example from Sutton 2018

**template** <**typename Iter**>

```
Iter min_element(Iter first, Iter last)
{
    if ( first == last ) return first;
    Iter min = first;
    while ( ++first != last )
        if ( *first < *min ) min = first;
    return *min;
}
```

- `ForwardIterator`
- **decltype(\*first)**
  - `TotallyOrdered`

# Concepts

- A type models a concepts
  - e.g. `char *` models `ForwardIterator`
  - `ForwardIterator<char *> && EqualityComparable<char>`
  - `ConvertibleTo<T, bool>`

- **Iter**

- `ForwardIterator` *//models Regular*

*// example from Sutton 2018*

**template** <typename **Iter**>

```
Iter min_element(Iter first, Iter last)
{
    if ( first == last ) return first;
    Iter min = first;
    while ( ++first != last )
        if ( *first < *min ) min = first;
    return *min;
}
```

- **decltype(\*first)**

- `TotallyOrdered`

# Concepts

- A type models a concepts
  - e.g. `char *` models `ForwardIterator`
  - `ForwardIterator<char *> && EqualityComparable<char>`
  - `ConvertibleTo<T, bool>`

```
// example from Sutton 2018
template <typename Iter>
    requires ForwardIterator<Iter>
           && TotallyOrdered<iter_value_t<Iter>>
Iter min_element(Iter first, Iter last)
{
    if ( first == last ) return first;
    Iter min = first;
    while ( ++first != last )
        if ( *first < *min ) min = first;
    return *min;
}
```

# Concepts

- A type models a concepts
  - e.g. `char *` models `ForwardIterator`
  - `ForwardIterator<char *> && EqualityComparable<char>`
  - `ConvertibleTo<T, bool>`

// example from Sutton 2018

```
template <ForwardIterator Iter>  
    requires TotallyOrdered<iter_value_t<Iter>>
```

```
Iter min_element(Iter first, Iter last)  
{  
    if ( first == last ) return first;  
    Iter min = first;  
    while ( ++first != last )  
        if ( *first < *min ) min = first;  
    return *min;  
}
```

# Concepts usage

- Semantic requirements
  - not necessary for ALL values
    - Floating points are TotallyOrdered but NaN does not work that way
    - Integers are Arithmetic, but overflow may exist
- Concepts may be overconstrained and underconstrained

// example from Sutton 2018

```
template <ObjectType T, AllocatorOf<T> Alloc = std::allocator<T>>  
class vector { /* ... */ };
```

```
vector<int>    v1;    // ok  
vector<int&>   v2;    // error  
vector<int&> *v3;    // error
```

```
template <FloatingPoint T> T pi = 3.14159265...;
```

```
namespace pmr {  
    template <ObjectType T>  
    using vector = std::vector<T, polymorphic_allocator<T>>;  
}
```

# Concepts usage

```
template <ObjectType T, ObjectType S>
struct pair
{
    template <ConvertibleTo<T> X, ConvertibleTo<S> Y>
    pair(const X& b, const Y& b) : first(a), second(b) { }

    pair() requires Defaultable<T> && Defaultable<S> : first(), second() { }

    pair(const pair& rhs) requires Copyable<T> && Copyable<S>
        : first(rhs.first), second(rhs.second) { }

    T first;
    S second;
};

pair<std::unique_ptr<int>, int> p1; // ok
pair<std::unique_ptr<int>, int> p2 = p1; // error
```



# Concepts definition

```
template <typename T, typename S>  
concept MyConcept =      std::same_as<T,S>  
                        && ( std::is_class_v<T> || std::is_enum_v<T> );
```

# Concepts definition

```
template <typename T, typename S>  
concept MyConcept =      std::same_as<T,S>  
                        && ( std::is_class_v<T> || std::is_enum_v<T> );
```

```
template <typename... Args>  
requires are_same_v<Args...>    // requires-clause  
auto add(Args&& ...args)  
{  
    return (... + args);  
}
```

# Concepts definition

```
template <typename T, typename S>
concept MyConcept =      std::same_as<T,S>
                        && ( std::is_class_v<T> || std::is_enum_v<T> );
```

```
template <typename... Args>
requires are_same_v<Args...>    // requires-clause
auto add(Args&& ...args)
{
    return (... + args);
}
```

```
template <typename... Args>
requires requires(Args... args) // requires-clause + requires-expression
{
    (... + args);           // simple requirement
    requires are_same_v<Args...>; // nested requirement
    requires sizeof...(Args) > 1; // nested requirement
    { (... + args) } noexcept -> same_as<first_arg_t<Args...>>;
}
auto add(Args&& ...args)
{
    return (... + args);
}
```

# Concepts definition

```
template <typename... Args>
concept Addable = requires(Args... args)
{
    (... + args);           // simple requirement
    requires are_same_v<Args...>; // nested requirement
    requires sizeof...(Args) > 1; // nested requirement
    { (... + args) } noexcept -> same_as<first_arg_t<Args...>>;
};
```

```
template <typename... Args>
requires Addable<Args...>
requires
{
    (... + args);           // simple requirement
    requires are_same_v<Args...>; // nested requirement
    requires sizeof...(Args) > 1; // nested requirement
    { (... + args) } noexcept -> same_as<first_arg_t<Args...>>;
}
auto add(Args&& ...args)
{
    return (... + args);
}
```

# Concepts definition

```
template <typename T, typename S>
concept MyConcept =      std::same_as<T,S>
                        && ( std::is_class_v<T> || std::is_enum_v<T> );

template <typename T>
concept Small =      sizeof(T) < 42;
```

# Concepts definition

```
template <typename T, typename S>
concept MyConcept =      std::same_as<T,S>
                        && ( std::is_class_v<T> || std::is_enum_v<T> );

template <typename T>
concept Small =      sizeof(T) < 42;

template <Small S>
int fun(S s) { return sizeof(s); }

struct big_t { char t[100]; };

int main()
{
    fun(1);
    fun( big_t{} );
    return 0;
}
```

# Concepts definition

```
template <typename T, typename S>
concept MyConcept =      std::same_as<T,S>
                        && ( std::is_class_v<T> || std::is_enum_v<T> );

template <typename T>
concept Small =      sizeof(T) < 42;
```

```
template <Small S>
int fun(S s) { return sizeof(s); }
```

```
struct big_t { char t[100]; };
```

```
int main()
{
    fun(1);
    fun( big_t{} );
    return 0;
}
```

\$ clang++ -std=c++20 small.cpp

small.cpp:12:3: error: no matching function for call to 'fun'

fun( big\_t{} );

^~~

small.cpp:5:5: note: candidate template ignored: constraints not satisfied [with S = big\_t]

int fun(S s) { return sizeof(s); }

^

small.cpp:4:11: note: because 'big\_t' does not satisfy 'Small'

template <Small S>

^

small.cpp:2:22: note: because 'sizeof(big\_t) < 42' (100 < 42) evaluated to false

concept Small = sizeof(T) < 42;

^

1 error generated.

# Concepts definition

```
template <typename T, typename S>
concept MyConcept =      std::same_as<T, S>
                        && ( std::is_class_v<T> || std::is_enum_v<T> );
```

```
template <typename T>
concept Small =          sizeof(T) < 42;
```

```
template <Small S>
int fun(S s) { return sizeof(s); }
```

```
struct big_t {
```

```
int main()
```

```
{
    fun(1);
    fun( big_t{} );
    return 0;
}
```

\$ g++ -std=c++20 small.cpp

small.cpp: In function 'int main()':

small.cpp:12:6: error: no matching function for call to 'fun(big\_t)'

12 | fun( big\_t{} );

| ~~~^~~~~~

small.cpp:5:5: note: candidate: 'template<class S> requires Small<S> int fun(S)'

5 | int fun(S s) { return sizeof(s); }

| ^~~

small.cpp:5:5: note: template argument deduction/substitution failed:

small.cpp:5:5: note: constraints not satisfied

small.cpp: In substitution of 'template<class S> requires Small<S> int fun(S) [with S = big\_t]':

small.cpp:12:6: required from here

small.cpp:2:9: required for the satisfaction of 'Small<S>' [with S = big\_t]

small.cpp:2:32: note: the expression 'sizeof (T) < 42 [with T = big\_t]' evaluated to 'false'

2 | concept Small = sizeof(T) < 42;

| ~~~~~~^~~~



# Concepts definition

```
template <typename T>
void DoLock(T&& f)
{
    std::lock_guard lock{std::mutex};
    f();
}
```

# Concepts definition

```
template <typename T>
void DoLock(T&& f)
{
    std::lock_guard lock{std::mutex};
    f();
}

void DoLock(std::invocable auto&& f)
{
    std::lock_guard lock{std::mutex};
    f();
}
```

# Concepts definition

```
template <typename T, typename U = void>
struct is_container : std::false_type { };
```

```
template <typename T>
struct is_container<
    T,
    std::void_t<typename T::value_type,
                typename T::size_type,
                typename T::allocator_type,
                typename T::iterator,
                typename T::const_iterator,
                decltype(std::declval<T>().size()),
                decltype(std::declval<T>().begin()),
                decltype(std::declval<T>().end()),
                decltype(std::declval<T>().cbegin()),
                decltype(std::declval<T>().cend()))
>> : std::true_type { };
```

```
struct A { };
```

```
static_assert(!is_container<A>::value);
static_assert( is_container<std::vector<A>>::value);
```

# Concepts definition

```
template <typename T>
concept container = requires(T t)
{
    typename T::value_type,
    typename T::size_type,
    typename T::allocator_type,
    typename T::iterator,
    typename T::const_iterator,
    t.size();,
    t.begin();
    t.end();
    t.cbegin();
    t.cend();
};
```

```
struct A { };
```

```
static_assert(!container<A>);
static_assert( container<std::vector<A>>);
```

# Concepts refinement

- Concept C **refines** concept D if when C is satisfied, D is also satisfied
- C **strictly refines** D if C refines D but D is not refines C
  - e.g. BidirectionalIterator refines ForwardIterator
- Refine is not inheritance
  - A BidirectionalIterator may not inherit from a ForwardIterator
- P subsumes Q if we can prove that  $P \Rightarrow Q$
- The compiler selects the most constrained declaration if all the types are equivalent
- Problems
  - Easy to write incomparable constraints (  $!(P \Rightarrow Q) \ \&\& \ !(Q \Rightarrow P)$  )
  - Easy to write ambiguous overloads

# Concepts specialization

```
template <InputIterator Iter>
int distance(Iter first, Iter last)
{
    int n = 0;
    while (first++ != last)
        ++n;
    return n;
}

template <RandomAccessIterator Iter>
int distance(Iter first, Iter last)
{
    return last - first;
}
```

# Concepts specialization

```
template <InputIterator In, OutputIterator<value_type_t<In>> Out>
Out copy(In first, In last, Out out)
{
    while (first != last)
        *out++ = *first++;
    return out;
}
```

```
template <TriviallyCopyable T>
T* copy(const T* first, const T* last, T* out)
{
    const int n = last - first;
    memcpy(out, first, n);
    return out + n;
}
```

```
const char *t1[10];
const char *t2[10];
copy( t1, t1+10, t2);  // ambiguous
```

# Selection statements with initializers

- ISO/IEC JTC1 SC22 WG21 P0305R1 (Thomas Köppe)



# Selection statements with initializers

```
/* C language, before C99 */
{
    int i;
    for ( i = 0; i < 10; ++i) {
        /* use i here */
    }
    /* i still visible here */
}
```

```
/* C++ language, C since C99 */
{
    for ( int i = 0; i < 10; ++i) {
        /* use i here */
    }
    /* i is not visible here */
}
```

# Selection statements with initializers

- ```
/* C++, since the beginning */
{
    if ( const char *path = std::getenv("PATH") ) {
        /* use path here */
    }
    else {
        /* path is also available here, nullptr */
    }
    /* path not available here */
}

{
    if ( auto sp = wp.lock() ) { /* shared_ptr from weak_ptr */
        /* use sp here */
    }
    /* sp is destructed here */
}
```

# Selection statements with initializers

- Not works well, when
  - it is not the declared variable we depend on
  - the success/fail is not usual int/bool/ptr != 0

```
std::set<int> s;
```

```
auto p = s.insert(42);  
if ( p.second ) {  
    std::cerr << "insert ok" << '\n';  
}  
else {  
    std::cerr << "insert failed" << '\n';  
}
```

```
std::mutex mut1, mut2, mut3;
```

```
int ret = std::try_lock( mut1, mut2, mut3 ); // many OS functions  
if ( -1 == ret ) {  
    std::cerr << "locks done" << '\n';  
}
```

# Selection statements with initializers

- Declaration is allowed in if and switch statements
  - The scope of declared variable is not “leaking” out
  - More flexibility for the condition

```
std::set<int> s;
```

```
// auto p = s.insert(42);  
if ( auto p = s.insert(42); p.second ) {  
    std::cerr << "insert ok" << '\n';  
}  
else {  
    std::cerr << "insert failed" << '\n';  
}
```

```
std::mutex mut1, mut2, mut3;
```

```
// int ret = std::try_lock( mu1t, mut2, mut3 );  
if ( int ret = std::try_lock( mu1t, mut2, mut3 ); -1 == ret ) {  
    std::cerr << "locks done" << '\n';  
}
```

# Selection statements with initializers

- Declaration is allowed in if and switch statements
  - The scope of declared variable is not “leaking” out
  - More flexibility for the condition

```
std::set<int> s;
```

```
// auto p = s.insert(42);  
if ( auto p = s.insert(42); p.second ) {  
    std::cerr << "insert ok" << '\n';  
}  
else {  
    std::cerr << "insert failed" << '\n';  
}
```

```
std::mutex mut1, mut2, mut3;
```

```
// int ret = std::try_lock( mu1t, mut2, mut3 );  
if ( int ret = std::try_lock( mu1t, mut2, mut3 ); -1 == ret ) {  
    std::cerr << "locks done" << '\n';  
} // unlock???
```

# Selection statements with initializers

- Use `lock_guard`, `unique_lock`, `scoped_lock`, ...

```
std::mutex      mut;  
std::deque<int> data;
```

```
// producer  
{  
    std::lock_guard sl(mut);  
    data.push_back(i);  
}
```

```
// consumer  
if ( std::lock_guard sl(mut); !data.empty() ) {  
    int i = data.front();  
    data.pop_front();  
}
```

# Selection statements with initializers

- Don't trick yourself!!!

```
std::mutex      mut;  
std::deque<int> data;
```

```
// producer  
{  
    std::lock_guard sl(mut);  
    data.push_back(i);  
}
```

```
// consumer  
if ( std::lock_guard(mut); !data.empty() ) { // bad!!!  
    int i = data.front();  
    data.pop_front();  
}
```

# Selection statements with initializers

- Switch is more interesting than you think.

```
#include <iostream>

int main(int argc, char *argv[])
{
    switch ( argc )
    {

        case 1: std::cout << "1"          << '\n'; break;
        case 2: std::cout << "2"          << '\n'; break;
        default: std::cout << "d"         << '\n'; break;
    }
    return 0;
}
```



# Selection statements with initializers

- Switch is more interesting than you think.

```
#include <iostream>

int main(int argc, char *argv[])
{
    switch ( argc )
    {
        int x;

        case 1: std::cout << "1" << x << '\n'; break;
        case 2: std::cout << "2" << x << '\n'; break;
        default: std::cout << "d" << x << '\n'; break;
    }
    return 0;
}
```

# Selection statements with initializers

- Switch is more interesting than you think.

```
#include <iostream>

int main(int argc, char *argv[])
{
    switch ( argc )
    {
        int x;

        case 1: std::cout << "1" << x << '\n'; break; // undefined beh.
        case 2: std::cout << "2" << x << '\n'; break;
        default: std::cout << "d" << x << '\n'; break;
    }
    return 0;
}
```

# Selection statements with initializers

- Switch is more interesting than you think.

```
#include <iostream>

int main(int argc, char *argv[])
{
    switch ( argc )
    {
        int x = argc;

        case 1: std::cout << "1" << x << '\n'; break;
        case 2: std::cout << "2" << x << '\n'; break;
        default: std::cout << "d" << x << '\n'; break;
    }
    return 0;
}
```

# Selection statements with initializers

- Switch is more interesting than you think.

```
#include <iostream>

int main(int argc, char *argv[])
{
    switch ( argc )
    {
        int x = argc;

        case 1: std::cout << "1" << x << '\n'; break;
        case 2: std::cout << "2" << x << '\n'; break;
        default: std::cout << "d" << x << '\n'; break;
    }
    return 0;
}
```

error: jump to **case** label XXX crosses initialization of **int** x

# Selection statements with initializers

- Switch is more interesting than you think.

```
#include <iostream>

int main(int argc, char *argv[])
{
    switch ( int x = argc )
    {
        // works even in "old" C++

        case 1: std::cout << "1" << x << '\n'; break;
        case 2: std::cout << "2" << x << '\n'; break;
        default: std::cout << "d" << x << '\n'; break;
    }
    return 0;
}
```

# Selection statements with initializers

- Switch is more interesting than you think.

```
#include <iostream>

int main(int argc, char *argv[])
{
    switch ( int x = argc; ++x )
    {
        // works since C++17

        case 1: std::cout << "1" << x << '\n'; break;
        case 2: std::cout << "2" << x << '\n'; break;
        default: std::cout << "d" << x << '\n'; break;
    }
    return 0;
}
```

# Selection statements with initializers

- Declaration list is allowed

```
#include <iostream>
#include <vector>

int main()
{
    std::vector v = { 1, 2, 3 }; // CTAD, C++17

    if (int s = v.size(), c = v.capacity(); s < c ) {
        std::cerr << "s < c" << '\n';
    }
    else {
        std::cerr << "s == c" << '\n';
    }
    return 0;
}
```

# Selection statements with initializers

- A bit more interesting case

```
#include <iostream>
#include <vector>

int main()
{
    std::vector v = { 1, 2, 3 }; // CTAD, C++17

    if (int s = v.size(), it = v.begin(); s > 0 && s < *it ) {
        std::cerr << "s < c" << '\n';
    }
    else {
        std::cerr << "s == c" << '\n';
    }
    return 0;
}
```

error: v.begin() is not convertible to int



# Selection statements with initializers

- Auto deduction must be consistent

```
#include <iostream>
#include <vector>

int main()
{
    std::vector v = { 1, 2, 3 }; // CTAD, C++17

    if (auto s = v.size(), it = v.begin(); s > 0 && s < *it ) {
        std::cerr << "s < c" << '\n';
    }
    else {
        std::cerr << "s == c" << '\n';
    }
    return 0;
}
```

error: inconsistent deduction for 'auto'

# Selection statements with initializers

- Structured binding helps

```
#include <iostream>
#include <vector>

int main()
{
    std::vector v = { 1, 2, 3 }; // CTAD, C++17

    if (auto [s,it] = std::pair{ v.size(),v.begin()}; s > 0 && s < *it){
        std::cerr << "s < c" << '\n';
    }
    else {
        std::cerr << "s == c" << '\n';
    }
    return 0;
}
```

works fine

# Selection statements with initializers

- Ideally, we should allow multiple statements

```
#include <iostream>
#include <vector>

int main()
{
    std::vector v = { 1, 2, 3 }; // CTAD, C++17

    if (auto s = v.size(); auto it = v.begin(); s > 0 && s < *it){
        std::cerr << "s < c" << '\n';
    }
    else {
        std::cerr << "s == c" << '\n';
    }
    return 0;
}
```

error: parse error

# Range issues

Nikolai Josuttis: C++ Standard Views @ Accu 2023

<https://www.youtube.com/watch?v=qv29fo9sUjY>

# Range issues

```
#include <vector>
#include <list>
#include <iostream>

template <typename Cont>
void print(const Cont& c)           // generic function to print a range
{
    for ( const auto& e : c)
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::vector v = { 1, 2, 3, 4, 5, 6, 7 };
    std::list l = { 1, 2, 3, 4, 5, 6, 7 };

    print(v);
    print(l);

    return 0;
}

$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
1 2 3 4 5 6 7
1 2 3 4 5 6 7
```

# Range issues

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

```
void print(const std::ranges::input_range auto& c) // C++20 format with concept
{
    for ( const auto& e : c)
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';
}
```

```
int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    std::list l = { 1,2,3,4,5,6,7 };

    print(v);
    print(l);
```

```
    return 0;
```

```
}
$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
1 2 3 4 5 6 7
1 2 3 4 5 6 7
```

# Range issues

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

void print(const std::ranges::input_range auto& c) // C++20 format with concept
{
    for ( const auto& e : c)
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    std::list l = { 1,2,3,4,5,6,7 };

    print(v);
    print(l);
    print(std::views::take(v,5)); // range adaptor representing a view of 1..5 subrange

    return 0;
}

$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
1 2 3 4 5 6 7
1 2 3 4 5 6 7
1 2 3 4 5
```

# Range issues

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

void print(const std::ranges::input_range auto& c) // C++20 format with concept
{
    for ( const auto& e : c)
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    std::list l = { 1,2,3,4,5,6,7 };

    print(v);
    print(l);
    print(std::views::take(v,5)); // range adaptor representing a view of 1..5 subrange
    print(std::views::take(l,5)); // range adaptor representing a view of 1..5 subrange
    return 0;
}

$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
1 2 3 4 5 6 7
1 2 3 4 5 6 7
1 2 3 4 5
1 2 3 4 5
```



# Range issues

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

void print(const std::ranges::input_range auto& c) // C++20 format with concept
{
    for ( const auto& e : c)
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    std::list l = { 1,2,3,4,5,6,7 };

    print(v);
    print(l);
    print(v | std::views::take(5)); // range adaptor representing a view of 1..5 subrange
    print(l | std::views::take(5)); // range adaptor representing a view of 1..5 subrange
    return 0;
}

$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
1 2 3 4 5 6 7
1 2 3 4 5 6 7
1 2 3 4 5
1 2 3 4 5
```

# Range issues

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

void print(const std::ranges::input_range auto& c) // C++20 format with concept
{
    //
}

int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    std::list l = { 1,2,3,4,5,6,7 };

    print(v);
    print(l);
    print( v | std::views::take(5)
          | std::views::transform([](auto e){return std::to_string(e)+'s';}));

    return 0;
}

$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
1 2 3 4 5 6 7
1 2 3 4 5 6 7
1s 2s 3s 4s 5s
```

# Range issues

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

void print(const std::ranges::input_range auto& c) // C++20 format with concept
{
    //
}

int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    std::list l = { 1,2,3,4,5,6,7 };

    print(v);
    print(l);
    print( v | std::views::take(5)
          | std::views::transform([](auto e){return std::to_string(e)+'s';}));
    print( l | std::views::take(5)
          | std::views::transform([](auto e){return std::to_string(e)+'s';}));
    return 0;
}

$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
1 2 3 4 5 6 7
1 2 3 4 5 6 7
1s 2s 3s 4s 5s
1s 2s 3s 4s 5s
```

# Range issues

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

void print(const std::ranges::input_range auto& c) // C++20 format with concept
{
    //
}

int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    std::list l = { 1,2,3,4,5,6,7 };

    print(v);
    print(l);
    print( v | std::views::take(5)
          | std::views::transform([](auto e){return std::to_string(e)+'s';}));
    print( l | std::views::take(5)
          | std::views::transform([](auto e){return std::to_string(e)+'s';}));
    return 0;
}

$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
1 2 3 4 5 6 7
1 2 3 4 5 6 7
1s 2s 3s 4s 5s
1s 2s 3s 4s 5s
```

# Range issues

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

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

```
int main()
{
    auto v = std::views::iota(1)
        | std::views::filter([](auto par){return 0 == par%3;})
        | std::views::drop(2)
        | std::views::transform([](auto par){return par*2;})
        | std::views::take(10);

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

    return 0;
}
$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
```

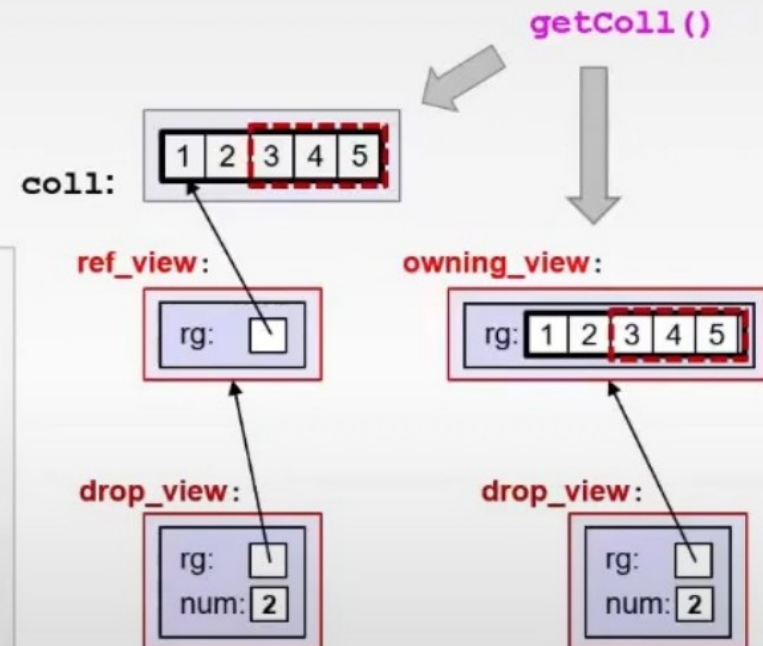
18  
24  
30  
36  
42  
48  
54  
60  
66  
72

# Range issues

- **ref\_view** when referring to named objects (lvalues)
- **owning\_view** when referring to temporaries (rvalues)

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

```
auto coll = getColl();  
auto v1 = coll | std::views::drop(2);  
           drop_view<ref_view<vector<int>>>  
  
auto v2 = getColl() | std::views::drop(2);  
           drop_view<owning_view<vector<int>>>
```



# Range issues

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

void print(const std::ranges::input_range auto& c)
{
    std::cout << "size == " << c.size() << '\n';
}

int main()
{
    std::vector v = { 1, 2, 3, 4, 5, 6, 7 };
    std::list l = { 1, 2, 3, 4, 5, 6, 7 };

    print(v);
    print(l);
    print(v | std::views::take(4));
    print(l | std::views::take(4));

    return 0;
}

$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
size == 7
size == 7
size == 4
size == 4
```

# Range issues

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

void print(const std::ranges::input_range auto& c)
{
    std::cout << "size == " << c.size() << '\n';
}

int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    std::list l = { 1,2,3,4,5,6,7 };

    print(v);
    print(l);
    print(v | std::views::filter([](auto e){return e%3==0;}));

    return 0;
}

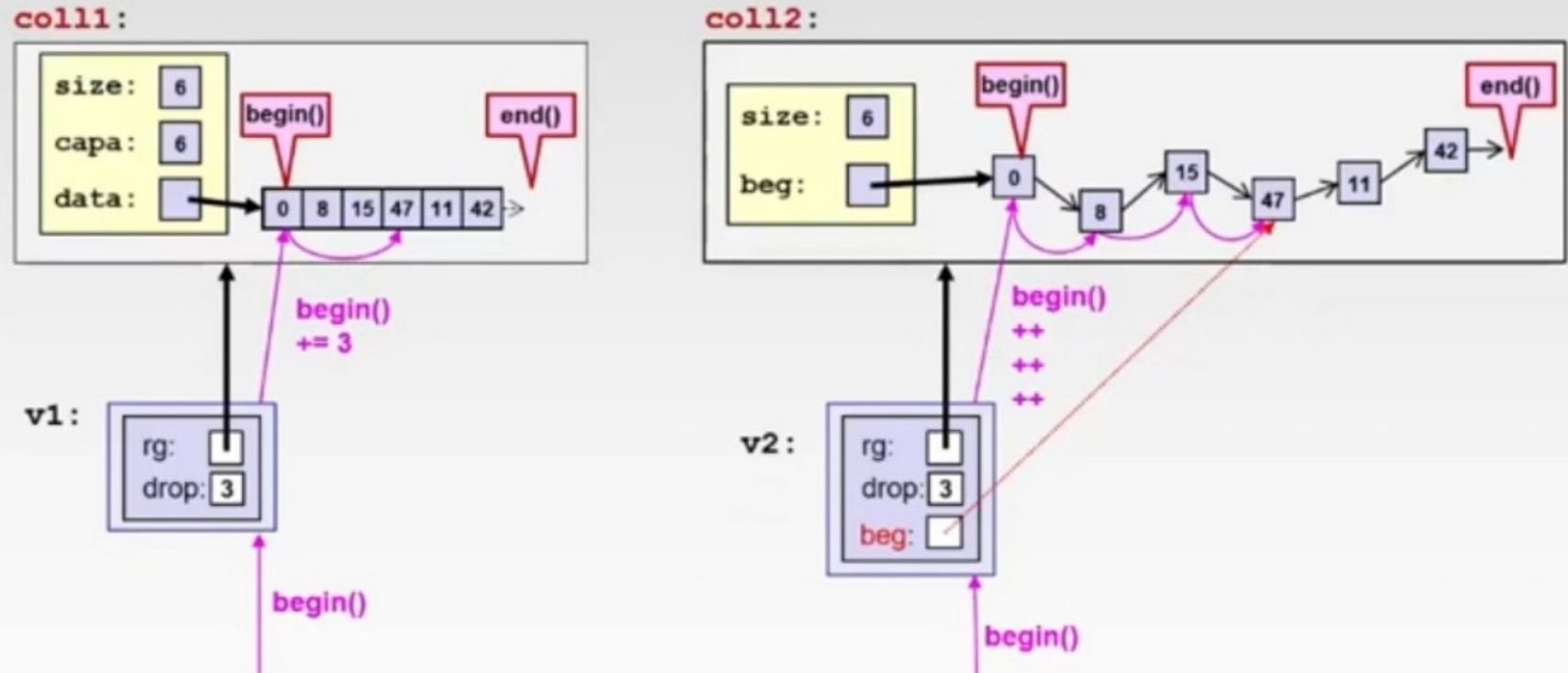
$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
...
error: no matching function for call to
'std::ranges::filter_view<std::ranges::ref_view<std::vector<int, std::allocator<int> > >, main()::<lambda(auto:17)>
>::size() const'
...
```



# Range issues

| member                           | Description            | Constraints                                |
|----------------------------------|------------------------|--------------------------------------------|
| <code>begin()</code>             | First iterator         |                                            |
| <code>end()</code>               | Last/sentinel          |                                            |
| <code>empty()</code>             | Is empty?              | Forward range                              |
| <code>operator bool()</code>     | Has elements?          | <code>std::ranges::empty()</code> is valid |
| <code>size()</code>              | <code>#elements</code> | Forward range and cheap                    |
| <code>front()</code>             | First element          | Forward range                              |
| <code>back()</code>              | Last element           | Bidirectional and common range             |
| <code>operator[n](size_t)</code> | Nth element            | Random-access range                        |
| <code>data()</code>              | Ptr to elements        | Contiguous range                           |

# Range issues



```
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();
```

# Range issues

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

```
template <typename Cont>
void print(const Cont& c)
{
    for ( const auto& e : c)
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';
}
```

```
int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    std::list l = { 1,2,3,4,5,6,7 };

    print(v); // 1 2 3 4 5 6 7
    print(l); // 1 2 3 4 5 6 7
    print(v | std::views::take(4)); // 1 2 3 4
    print(l | std::views::take(4)); // 1 2 3 4
    print(v | std::views::drop(2)); // 3 4 5 6 7

    return 0;
}
$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
```

```
jos11.cpp: In instantiation of 'void print(const Cont&) [with Cont =
std::ranges::drop_view<std::ranges::ref_view<std::__
cxx11::list<int, std::allocator<int> > > >]':
jos11.cpp:26:8:   required from here
jos11.cpp:9:3: error: passing 'const
std::ranges::drop_view<std::ranges::ref_view<std::__
cxx11::list<int, std::allocator<int> > > >' as
'this' argument discards qualifiers [-fpermissive]
    9 |     for ( const auto& e : c)
      |     ^~~
```

# Range issues

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

```
template <typename Cont>
void print(const Cont& c)
{
    for ( const auto& e : c)
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';
}
```

```
int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    std::list l = { 1,2,3,4,5,6,7 };

    print(v); // 1 2 3 4 5 6 7
    print(l); // 1 2 3 4 5 6 7
    print(v | std::views::take(4)); // 1 2 3 4
    print(l | std::views::take(4)); // 1 2 3 4
    print(v | std::views::drop(2)); // 3 4 5 6 7
    print(l | std::views::drop(2)); // compile error

    return 0;
}
$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
```

```
jos11.cpp: In instantiation of 'void print(const Cont&) [with Cont =
std::ranges::drop_view<std::ranges::ref_view<std::__
cxx11::list<int, std::allocator<int> > > >]':
jos11.cpp:26:8:   required from here
jos11.cpp:9:3: error: passing 'const
std::ranges::drop_view<std::ranges::ref_view<std::__
cxx11::list<int, std::allocator<int> > > >' as
'this' argument discards qualifiers [-fpermissive]
    9 |     for ( const auto& e : c)
      |     ^~~
```

# Range issues

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

```
template <typename Cont>
void print(const Cont& c)
{
    for ( const auto& e : c)
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';
}
```

```
int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    std::list l = { 1,2,3,4,5,6,7 };

    print(v); // 1 2 3 4 5 6 7
    print(l); // 1 2 3 4 5 6 7
    print(v | std::views::take(4)); // 1 2 3 4
    print(l | std::views::take(4)); // 1 2 3 4
    print(v | std::views::drop(2)); // 3 4 5 6 7
    print(l | std::views::drop(2)); // compile error
    for(int e : l | std::views::drop(2)) {std::cout << e << ' ';}; // 3 4 5 6 7
    return 0;
}
$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
```

```
jos11.cpp: In instantiation of 'void print(const Cont&) [with Cont =
std::ranges::drop_view<std::ranges::ref_view<std::__
cxx11::list<int, std::allocator<int> > > >]':
jos11.cpp:26:8:   required from here
jos11.cpp:9:3: error: passing 'const
std::ranges::drop_view<std::ranges::ref_view<std::__
cxx11::list<int, std::allocator<int> > > >' as
'this' argument discards qualifiers [-fpermissive]
    9 |     for ( const auto& e : c)
      |     ^~~
```

# Range issues

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

```
template <typename Cont>
void print(const Cont& c)
{
    for ( const auto& e : c)
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';
}
```

```
int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    std::list l = { 1,2,3,4,5,6,7 };
```

```
    print(v); // 1 2 3 4 5 6 7
    print(l); // 1 2 3 4 5 6 7
    print(v | std::views::take(4)); // 1 2 3 4
    print(l | std::views::take(4)); // 1 2 3 4
    print(v | std::views::drop(2)); // 3 4 5 6 7
    print(l | std::views::drop(2)); // compile error: caching the begin iterator
    for(int e : l | std::views::drop(2)) {std::cout << e << ' ';;} // 3 4 5 6 7
    return 0;
```

```
}
$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
```

```
jos11.cpp: In instantiation of 'void print(const Cont&) [with Cont =
std::ranges::drop_view<std::ranges::ref_view<std::__
cxx11::list<int, std::allocator<int> > > >]':
jos11.cpp:26:8:   required from here
jos11.cpp:9:3: error: passing 'const
std::ranges::drop_view<std::ranges::ref_view<std::__
cxx11::list<int, std::allocator<int> > > >' as
'this' argument discards qualifiers [-fpermissive]
    9 |     for ( const auto& e : c)
      |     ^~~
```

# Range issues

Amortized constant cost on the second call of `begin()` and `empty()`

|                                        | <code>begin()</code> 1st | <code>begin()</code> 2nd | <code>size()</code> | <code>empty()</code> 1st | <code>empty()</code> 2nd |
|----------------------------------------|--------------------------|--------------------------|---------------------|--------------------------|--------------------------|
| <code>std::vector vec</code>           | constant                 | constant                 | constant            | constant                 | constant                 |
| <code>std::list lst</code>             | constant                 | constant                 | constant            | constant                 | constant                 |
| <code>vec   drop(n)</code>             | constant                 | constant                 | constant            | constant                 | constant                 |
| <code>lst   drop(n)</code>             | linear                   | constant                 | constant            | constant                 | constant                 |
| <code>vec   filter(f)</code>           | linear                   | constant                 | ---                 | linear                   | constant                 |
| <code>lst   filter(f)</code>           | linear                   | constant                 | ---                 | linear                   | constant                 |
| <code>vec   filter(f)   drop(n)</code> | linear                   | constant                 | ---                 | linear                   | constant                 |
| <code>lst   filter(f)   drop(n)</code> | linear                   | constant                 | ---                 | linear                   | constant                 |

# Range issues

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

```
template <typename Cont>
void print(const Cont& c)
{
    for ( const auto& e : c)
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';
}
```

```
int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    std::list l = { 1,2,3,4,5,6,7 };

    print(v); // 1 2 3 4 5 6 7
    print(l); // 1 2 3 4 5 6 7
    print(v | std::views::take(4)); // 1 2 3 4
    print(l | std::views::take(4)); // 1 2 3 4
    print(v | std::views::drop(2)); // 3 4 5 6 7
    auto vw = l | std::views::drop(2);
    print(std::ranges::subrange{vw.begin(),vw.end()}); // 3 4 5 6 7

    return 0;
}
$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
```

```
1 2 3 4 5 6 7
1 2 3 4 5 6 7
1 2 3 4
1 2 3 4
3 4 5 6 7
3 4 5 6 7
3 4 5 6 7
```



# Range issues

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

```
template <typename Cont>
void print(Cont&& c)      // forwarding reference
{
    for ( const auto& e : c)
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';
}
```

```
int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    std::list l = { 1,2,3,4,5,6,7 };

    print(v);           // 1 2 3 4 5 6 7
    print(l);           // 1 2 3 4 5 6 7
    print(v | std::views::take(4)); // 1 2 3 4
    print(l | std::views::take(4)); // 1 2 3 4
    print(v | std::views::drop(2)); // 3 4 5 6 7
    print(l | std::views::drop(2)); // 3 4 5 6 7

    return 0;
}

$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
```

```
1 2 3 4 5 6 7
1 2 3 4 5 6 7
1 2 3 4
1 2 3 4
3 4 5 6 7
3 4 5 6 7
3 4 5 6 7
```

# Range issues

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

template <typename Cont>
void print(Cont&& c)          // forwarding reference
{
    std::jthread t1{[&] { for ( const auto& e : c)
        { std::cout << e << ' '; } std::cout << '\n'; }};
    std::jthread t2{[&] { for ( const auto& e : c)
        { std::cout << e << ' '; } std::cout << '\n'; }};
}

int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    std::list l = { 1,2,3,4,5,6,7 };

    print(v);                // 1 2 3 4 5 6 7
    print(l);                // 1 2 3 4 5 6 7
    print(v | std::views::take(4)); // 1 2 3 4
    print(l | std::views::take(4)); // 1 2 3 4
    print(v | std::views::drop(2)); // 3 4 5 6 7
    print(l | std::views::drop(2)); // 3 4 5 6 7

    return 0;
}

$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
```

# Range issues

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

```
template <typename Cont>
void print(Cont&& c)          // forwarding reference
{
    std::jthread t1{[&] { for ( const auto& e : c)
        { std::cout << e << ' '; } std::cout << '\n'; } } };
    std::jthread t2{[&] { for ( const auto& e : c)
        { std::cout << e << ' '; } std::cout << '\n'; } } };
}
```

Data RACE on begin()

```
int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    std::list l = { 1,2,3,4,5,6,7 };

    print(v);           // 1 2 3 4 5 6 7
    print(l);           // 1 2 3 4 5 6 7
    print(v | std::views::take(4)); // 1 2 3 4
    print(l | std::views::take(4)); // 1 2 3 4
    print(v | std::views::drop(2)); // 3 4 5 6 7
    print(l | std::views::drop(2)); // 3 4 5 6 7 Runtime error

    return 0;
}
$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
```

# Modifications

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

template <typename Cont>
void print(Cont&& c)
{
    for ( const auto& e : c)
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    print(v);

    auto evens = v | std::views::filter([](auto&& i){return i%2 == 0;});
    for (int& e : evens) { e+=2; } // modification via views is allowed
    print(v);

    return 0;
}

$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
1 2 3 4 5 6 7
1 4 3 6 5 8 7
```

# Modifications

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

template <typename Cont>
void print(Cont&& c)
{
    for ( const auto& e : c)
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    print(v);

    auto evens = v | std::views::filter([](auto&& i){return i%2 == 0;});
    for (int& e : evens) { e+=2; } // modification via views is allowed
    print(v);
    for (int& e : evens) { e+=2; } // modification via views is allowed
    print(v);

    return 0;
}

$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
1 2 3 4 5 6 7
1 4 3 6 5 8 7
1 6 3 8 5 10 7
```

# Modifications

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

template <typename Cont>
void print(Cont&& c)
{
    for ( const auto& e : c)
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    print(v);

    auto evens = v | std::views::filter([](auto&& i){return i%2 == 0;});
    for (int& e : evens) { e+=1; } // modification via views is allowed
    print(v);

    return 0;
}

$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
1 2 3 4 5 6 7
1 3 3 5 5 7 7
```

# Modifications

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

```
template <typename Cont>
void print(Cont&& c)
```

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

```
int main()
```

```
{
    std::vector v = { 1,2,3,4,5,6,7 };
    print(v);
```

```
    auto evens = v | std::views::filter([](auto&& i){return i%2 == 0;});
```

```
    for (int& e : evens) { e+=1; } // modification via views is allowed
```

```
    print(v);
```

```
    for (int& e : evens) { e+=1; } // modification via views is allowed
```

```
    print(v);
```

```
    return 0;
```

```
}
```

```
$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
```

```
1 2 3 4 5 6 7
```

```
1 3 3 5 5 7 7
```

```
1 4 3 5 5 7 7
```

Begin was cached

# Modifications

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

```
template <typename Cont>
void print(Cont&& c)
{
    for ( const auto& e : c)
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';
}
```

```
int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    print(v);

    auto evens = v | std::views::filter([](auto&& i){return i%2 == 0;});
    for (int& e : evens) { e+=1; } // modification via views is allowed
    print(v);
    for (int& e : evens) { e+=1; } // modification via views is allowed
    print(v);

    return 0;
}
```

```
$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
1 2 3 4 5 6 7
1 3 3 5 5 7 7
1 4 3 5 5 7 7
```

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.

Filter is not satisfied  
Undefined behavior



# Modifications

```
#include <array>
#include <cctype>
#include <iostream>
#include <ranges>
```

```
template <typename Cont>
void print(Cont&& c)
```

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

```
int main()
```

```
{
    std::array arr = std::to_array("Hello World\n"); // CTAD
    print(arr);
```

```
    auto uppers = arr | std::views::filter([](auto&& ch){return std::isupper(ch);});
    for (auto& ch : uppers) { ch = std::tolower(ch); }
    print(arr);
```

```
    return 0;
```

```
}
$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
H e l l o   W o r l d
h e l l o   w o r l d
```

# Modifications

```
#include <array>
#include <cctype>
#include <iostream>
#include <ranges>
```

```
template <typename Cont>
```

```
void print(Cont&& c)
```

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

```
int main()
```

```
{
    std::array arr = std::to_array("Hello World\n"); // CTAD
    print(arr);
```

```
    auto uppers = arr | std::views::filter([](auto&& ch){return std::isupper(ch);});
    for (auto& ch : uppers) { ch = std::tolower(ch); }
    print(arr);
```

```
    return 0;
```

```
}
$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
H e l l o   W o r l d
h e l l o   w o r l d
```

Filter is not satisfied  
Undefined behavior

# Modifications

```
#include <array>
#include <cctype>
#include <iostream>
#include <ranges>

template <typename Cont>
void print(Cont&& c)
{
    for ( const auto& e : c)
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::array arr = std::to_array("Hello World\n"); // CTAD
    print(arr);

    auto is_upper = [](auto&& ch){return std::isupper(ch);};
    for (auto&& ch : arr | std::views::filter(is_upper)) { ch = std::tolower(ch); } // ub
    for (auto&& ch : arr | std::views::filter(is_upper)) { ch = std::tolower(ch); } // ub
    print(arr);

    return 0;
}

$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
H e l l o   W o r l d
h e l l o   w o r l d
```

# Modifications

```
#include <array>
#include <cctype>
#include <iostream>
#include <ranges>

template <typename Cont>
void print(Cont&& c)
{
    for ( const auto& e : c)
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::array arr = std::to_array("Hello World\n"); // CTAD
    print(arr);

    auto is_upper = [](auto&& ch){return std::isupper(ch);};
    for (auto&& ch : arr | std::views::filter(is_upper)) { ch = std::tolower(ch); } // ub
    for (auto&& ch : arr | std::views::filter(is_upper)) { ch = std::tolower(ch); } // ub
    print(arr);

    return 0;
}

$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
H e l l o   W o r l d
h e l l o   w o r l d
```

Undefined behavior  
But works

# Modifications

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

```
template <typename Cont>
void print(Cont&& c)
```

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

```
int main()
```

```
{
    std::vector v = { 1,2,3,4,5,6,7 };
    print(v);
```

```
    auto is_even = [](auto&& i){ return i%2 == 0; };
    for (int& e : v | std::views::filter(is_even)) { e+=1; }
```

```
    print(v);
    for (int& e : v | std::views::filter(is_even)) { e+=1; }
```

```
    print(v);
```

```
    return 0;
}
```

```
$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
```

```
1 2 3 4 5 6 7
```

```
1 3 3 5 5 7 7
```

```
1 3 3 5 5 7 7
```

Undefined behavior

But works

# Modifications

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

```
template <typename Cont>
void print(Cont&& c)
{
    for ( const auto& e : c)
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';
}
```

```
int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    std::list l = { 1,2,3,4,5,6,7 };

    v.reserve(16);
    auto v2 = v | std::views::drop(2);
    auto l2 = l | std::views::drop(2);
    print(v2);
    print(l2);

    v.insert( v.begin(), { -3,-2, -1, 0 } );
    l.insert( l.begin(), { -3,-2, -1, 0 } );
    print(v3);
    print(l3);

    return 0;
}

$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
```

```
3 4 5 6 7
3 4 5 6 7
-1 0 1 2 3 4 5 6 7
3 4 5 6 7
```

# Modifications

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

```
template <typename Cont>
void print(Cont&& c)
{
    for ( const auto& e : c)
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';
}
```

```
int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    std::list l = { 1,2,3,4,5,6,7 };

    v.reserve(16);
    auto v2 = v | std::views::drop(2);
    auto l2 = l | std::views::drop(2);
    print(v2);
    print(l2);

    v.insert( v.begin(), { -3,-2, -1, 0 } );
    l.insert( l.begin(), { -3,-2, -1, 0 } );
    auto v3 = v2; print(v3);      // copy erases the cache
    auto l3 = l2; print(l3);     // copy erases the cache

    return 0;
}

$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
```

```
3 4 5 6 7
3 4 5 6 7
-1 0 1 2 3 4 5 6 7
-1 0 1 2 3 4 5 6 7
```

# Constness

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

template <typename Cont>
void print(Cont&& c)
{
    auto pos = c.cbegin();
    std::cout << *pos << '\n';
}

int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    print(v);
    print(v | std::views::drop(2)); // compile error, no cbegin() on views.

    return 0;
}
$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
```



# Constness

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

template <typename Cont>
void print(Cont&& c)
{
    *std::cbegin(c) = 42;
    *std::ranges::cbegin(c) = 42;
}

int main()
{
    std::vector v = { 1,2,3,4,5,6,7 };
    print(v); // compile error
    print(v | std::views::drop(2)); // compiles and modifies the underlying data

    for ( auto e : v ) { std::cout << e << ' '; } std::cout << '\n';

    return 0;
}
$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
```

# Constness

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

```
template <typename Cont>
```

```
void print(Cont&& c)
```

```
{
    *std::cbegin(c) = 42;
    *std::ranges::cbegin(c) = 42;
}
```

```
int main()
```

```
{
    std::vector v = { 1,2,3,4,5,6,7 };
    print(v); // compile error
    print(v | std::views::drop(2)); // compiles and modifies the underlying data
```

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

```
    return 0;
```

```
}
$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
```

```
1 2 42 4 5 6 7
```

A view == pointer

The pointer is const

Not the pointed element

# Constness

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

```
template <typename Cont>
```

```
void print(Cont&& c)
```

```
{
    *std::cbegin(c) = 42;
    *std::ranges::cbegin(c) = 42;
}
```

```
int main()
```

```
{
    std::vector v = { 1,2,3,4,5,6,7 };
    print(v); // compile error
    print(v | std::views::drop(2)); // compiles and modifies the underlying data
```

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

```
    return 0;
```

```
}
$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
```

```
1 2 42 4 5 6 7
```

An lvalue does not propagate const  
An rvalue does propagate const

# Constness

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

```
template <typename Cont>
```

```
void print(Cont&& c)
```

```
{
    *std::cbegin(c) = 42;
    *std::ranges::cbegin(c) = 42;
}
```

```
int main()
```

```
{
    std::vector v = { 1,2,3,4,5,6,7 };
    print(v); // compile error
    print(v | std::views::drop(2)); // compiles and modifies the underlying data
```

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

```
    return 0;
```

```
}
$ g++ -Wextra -std=c++20 jos1.cpp && ./a.out
```

```
1 2 42 4 5 6 7
```

An lvalue **does not** propagate const

An rvalue **does** propagate const

**NEVER NAME A VIEW!**

# Constness

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

template <typename Cont>
void printPairs(const auto& c)
{
    for (const auto& e : c)
    {
        e.first = 42;
        std::cout << e.first << ' ' << e.second << '\n';
    }
}

int main()
{
    std::vector v1 = { 1, 2, 3, 4, 5, 6, 7 };
    std::vector v2 = { -1, -2, -3, -4, -5, -6, -7 };

    printPairs(std::views::zip(v1, v2));

    return 0;
}

$ g++ -Wextra -std=c++23 jos1.cpp && ./a.out
42 -1
42 -2
42 -3
42 -4
42 -5
42 -6
42 -7
```

# Some fixes in C++23

- New type traits `std::const_iterator<>`, `std::const_sentinel<>`, ...
- Fixing `std::ranges::cbegin()` and `std::ranges::cend()`
- All views have `cbegin()` and `cend()`
- New concept `std::ranges::constant_range<>`
- New helper function `std::ranges::as_const()`
- But `std::as_const()` does not makes the elements const
- Josuttis: Belleviews

# Comparison

# Comparison

- Until C++20
  - Primary: `==` `!=`
  - Secondary: `<` `<=` `>` `>=`
- All the operators treated equally
- We rely on basic idioms, e.g, `a <= b` is the same as `!(b < a)`
  - But `1.f < NaN` `1.f == NaN` `1.f > NaN` are all false, and `!(NaN < 1.f)` is true
- Operators are recommended to write as non-member (conversions)



# Comparison

- Until C++20
  - Primary:     ==   !=
  - Secondary: <   <=   >   >=
- Since C++20
  - `operator!=` removed and synthesized from `operator==`  
`operator! (a, b) { return ! operator==(a, b); }`
- Defaulted comparison operator
  - Non-templated
  - No-static member or friend of class X
  - Defaulted
  - Two parameters are X or const X& (and “this” is the first param)

# Comparison

```
struct X
{
    bool operator==(const X&) const = default;    // OK
    bool operator==(this X, X)      = default;    // OK

    bool operator==(const X&)        = default;    // error, left parameter is not const
};

struct Y
{
    friend bool operator==(Y, Y)      = default;    // OK

    friend bool operator==(Y, const Y&) = default;    // error, different parameter types
};

int f(X a, X b)
{
    a != b;    // ! a.operator==(b)
    a < b;     // a.operator<=>(b) < 0
}
```

# Comparison

- Since C++20: “starship” operator
  - 3 results: `<0` `==0` `>0`
- Return type: comparison category
  - `strong_ordering` `42 == 40+2` `strong_ordering::equal`
  - `weak_ordering` `“Hello” == “HELLO”` `weak_ordering::equivalent`
  - `partial_ordering` `1.f == NaN` `partial_ordering::unordered`
  -
- `a @ b` is the same as `( a <=> b ) @ 0` // where @ is `<` or `==` or `>`

# Comparison

```
struct X
{
    bool operator==(const X&) const = default;    // OK
    bool operator==(this X, X)      = default;    // OK

    bool operator==(const X&)        = default;    // error, left parameter is not const
};

struct Y
{
    friend bool operator==(Y, Y)      = default;    // OK

    friend bool operator==(Y, const Y&) = default;    // error, different parameter types
};

int f(X a, X b)
{
    a != b;    // ! a.operator==(b)
    a < b;     // a.operator<=>(b) < 0
}
```