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 ?          YES: X is defined, foo defined
Can I call foo() ?                    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(), s.Filename.end(),
                                /* ... */
                                &rCppFiles);

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

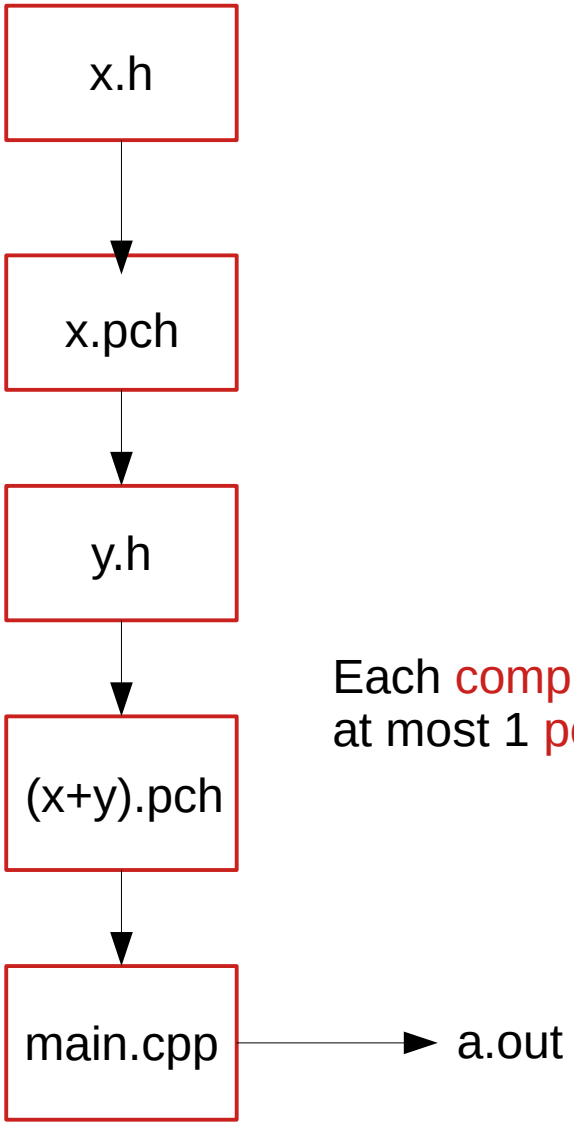
Build times

- 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

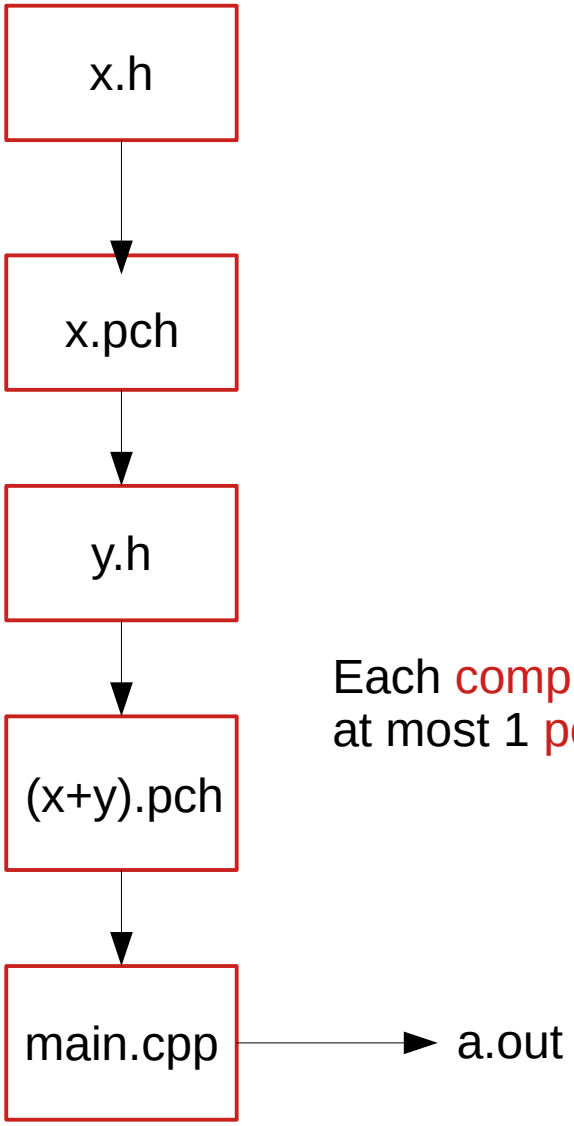
- Precompiled headers (PCH)
  - Safe and reuse AST
  - PCH require build system support
- HP aCC compiler
  - Automatic generation of PCH

PCH vs Modules

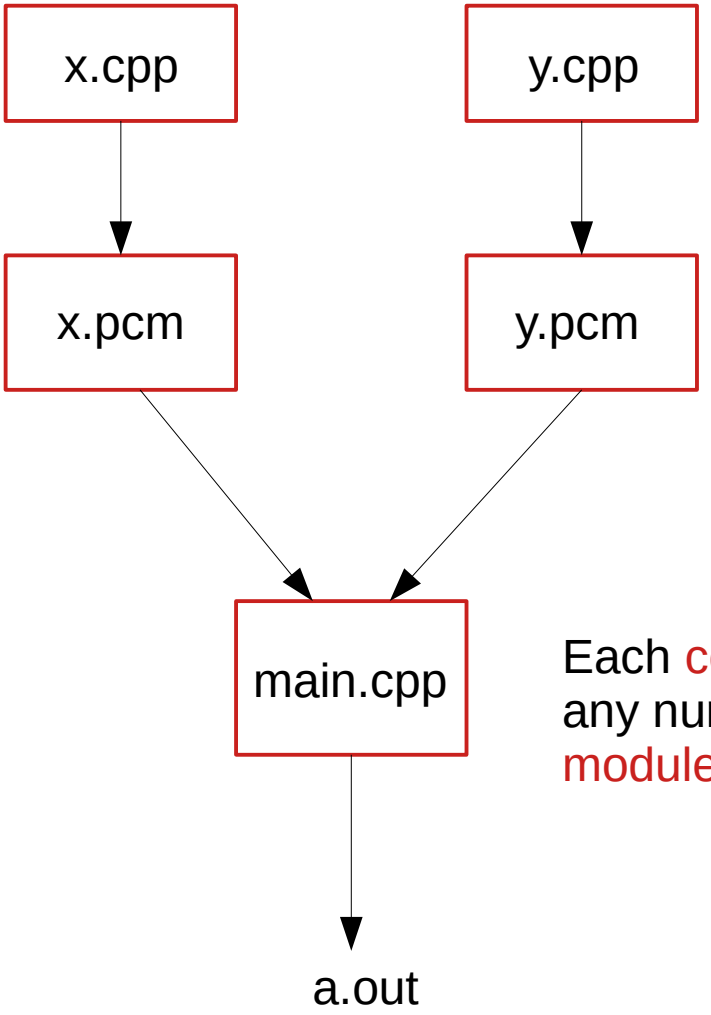


Each compilation
at most 1 pch input

PCH vs Modules



Each compilation
at most 1 pch input



Each compilation
any number of
module import

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
- 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=o1lNd12uYjE
- 2014 C++14 misses Concepts
- 2015 C++Now 2015 Keynote
  - Andrew Sutton: Generic Programming with Concepts
    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>

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

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

```
// 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
  - 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>

```
// 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
  - ForwardIterator
- **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
  - ForwardIterator
- **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
  - ForwardIterator
- **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>


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**
  - ForwardIterator `//models Regular`
- **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 overcontrained 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_conainer<A>::value);
static_assert( is_conainer<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
  t.begin();
  t.end();
  t.cbegin();
  t.cend();
};



struct A { };

static_assert(!conainer<A>);
static_assert( conainer<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 => Q
- The compiler selects the most constrained declaration if all the types are equivalent
- Problems
  - Easy to write incomparable constraints (!(P => Q) && !(Q => 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;
  memcopy(out, first, n);
  return out + n;
}

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