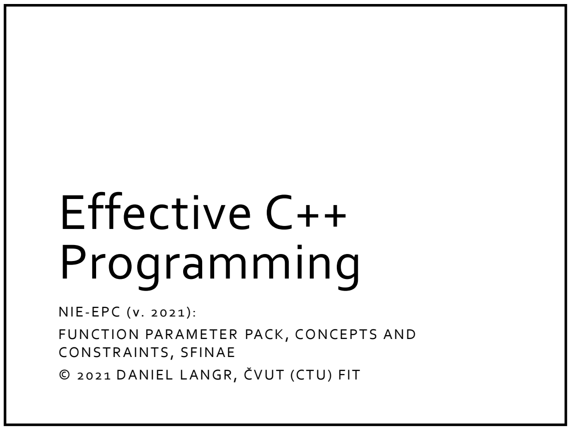
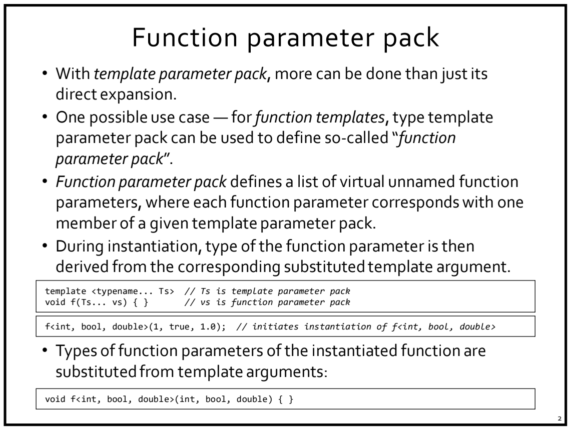


Effective C++ Programming

NIE-EPC (v. 2021):

FUNCTION PARAMETER PACK, CONCEPTS AND CONSTRAINTS, SFINAE

1



Function parameter pack

- With *template parameter pack*, more can be done than just its direct expansion.
- One possible use case — for *function templates*, type template parameter pack can be used to define so-called "*function parameter pack*".
- *Function parameter pack* defines a list of virtual unnamed function parameters, where each function parameter corresponds with one member of a given template parameter pack.
- During instantiation, type of the function parameter is then derived from the corresponding substituted template argument.

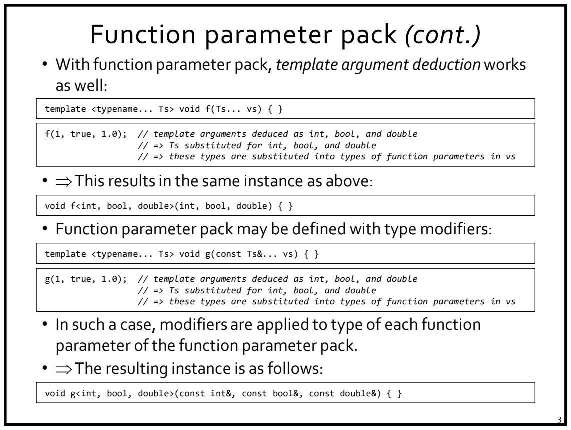
```
template <typename... Ts>  // Ts is template parameter pack
void f(Ts... vs) { }       // vs is function parameter pack
```

```
f<int, bool, double>(1, true, 1.0);  // initiates instantiation of f<int, bool, double>
```

- Types of function parameters of the instantiated function are substituted from template arguments:

```
void f<int, bool, double>(int, bool, double) { }
```

2



Function parameter pack *(cont.)*

- With function parameter pack, *template argument deduction* works as well:

```
template <typename... Ts> void f(Ts... vs) { }
```

```
f(1, true, 1.0);  // template arguments deduced as int, bool, and double
                  // => Ts substituted for int, bool, and double
                  // => these types are substituted into types of function parameters in vs
```

- ⇒ This results in the same instance as above:

```
void f<int, bool, double>(int, bool, double) { }
```

- Function parameter pack may be defined with type modifiers:

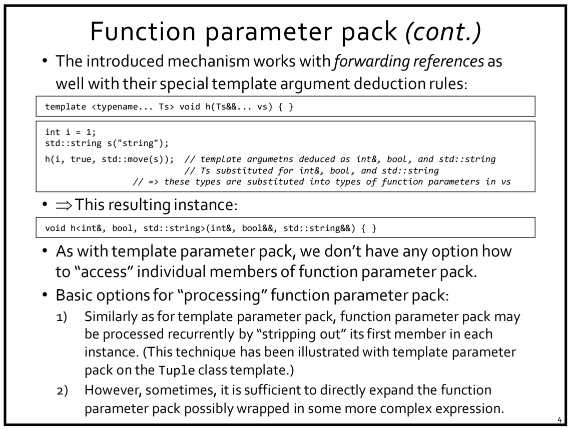
```
template <typename... Ts> void g(const Ts&... vs) { }
```

```
g(1, true, 1.0);  // template arguments deduced as int, bool, and double
                  // => Ts substituted for int, bool, and double
                  // => these types are substituted into types of function parameters in vs
```

- In such a case, modifiers are applied to type of each function parameter of the function parameter pack.
- ⇒ The resulting instance is as follows:

```
void g<int, bool, double>(const int&, const bool&, const double&) { }
```

3



Function parameter pack *(cont.)*

- The introduced mechanism works with *forwarding references* as well with their special template argument deduction rules:

```
template <typename... Ts> void h(Ts&&... vs) { }
```

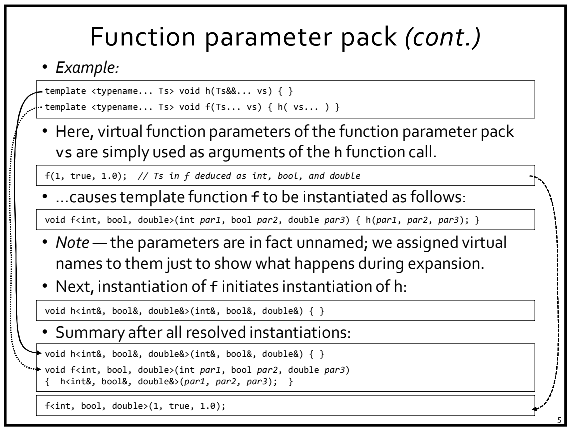
```
int i = 1;
std::string s("string");
h(i, true, std::move(s));  // template argumetns deduced as int&, bool, and std::string
                           // Ts substituted for int&, bool, and std::string
                // => these types are substituted into types of function parameters in vs
```

- ⇒ This resulting instance:

```
void h<int&, bool, std::string>(int&, bool&&, std::string&&) { }
```

- As with template parameter pack, we don't have any option how to "access" individual members of function parameter pack.
- Basic options for "processing" function parameter pack:
  1) Similarly as for template parameter pack, function parameter pack may be processed recurrently by "stripping out" its first member in each instance. (This technique has been illustrated with template parameter pack on the `Tuple` class template.)
  2) However, sometimes, it is sufficient to directly expand the function parameter pack possibly wrapped in some more complex expression.

4



Function parameter pack *(cont.)*

- *Example:*

```
template <typename... Ts> void h(Ts&&... vs) { }
template <typename... Ts> void f(Ts... vs) { h( vs... ) }
```

- Here, virtual function parameters of the function parameter pack `vs` are simply used as arguments of the `h` function call.

```
f(1, true, 1.0);  // Ts in f deduced as int, bool, and double
```

- …causes template function `f` to be instantiated as follows:

```
void f<int, bool, double>(int par1, bool par2, double par3) { h(par1, par2, par3); }
```

- *Note* — the parameters are in fact unnamed; we assigned virtual names to them just to show what happens during expansion.
- Next, instantiation of `f` initiates instantiation of `h`:

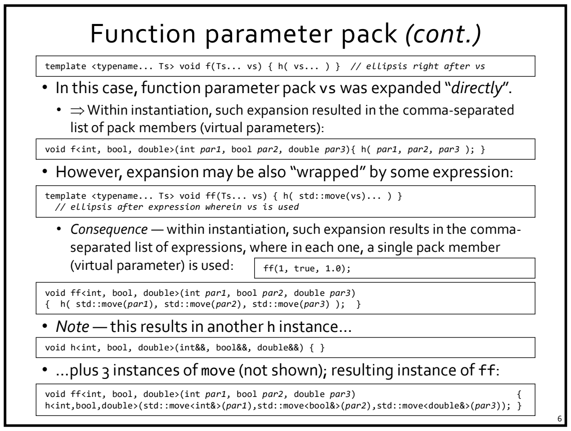
```
void h<int&, bool&, double&>(int&, bool&, double&) { }
```

- Summary after all resolved instantiations:

```
void h<int&, bool&, double&>(int&, bool&, double&) { }
void f<int, bool, double>(int par1, bool par2, double par3)
{  h<int&, bool&, double&>(par1, par2, par3);  }
```

```
f<int, bool, double>(1, true, 1.0);
```

5



Function parameter pack *(cont.)*

```
template <typename... Ts> void f(Ts... vs) { h( vs... ) }  // ellipsis right after vs
```

- In this case, function parameter pack `vs` was expanded "*directly*".
  - ⇒ Within instantiation, such expansion resulted in the comma-separated list of pack members (virtual parameters):

```
void f<int, bool, double>(int par1, bool par2, double par3){ h( par1, par2, par3 ); }
```

- However, expansion may be also "wrapped" by some expression:

```
template <typename... Ts> void ff(Ts... vs) { h( std::move(vs)... ) }
  // ellipsis after expression wherein vs is used
```

  - *Consequence* — within instantiation, such expansion results in the comma-separated list of expressions, where in each one, a single pack member (virtual parameter) is used:

```
ff(1, true, 1.0);
```

```
void ff<int, bool, double>(int par1, bool par2, double par3)
{  h( std::move(par1), std::move(par2), std::move(par3) );  }
```

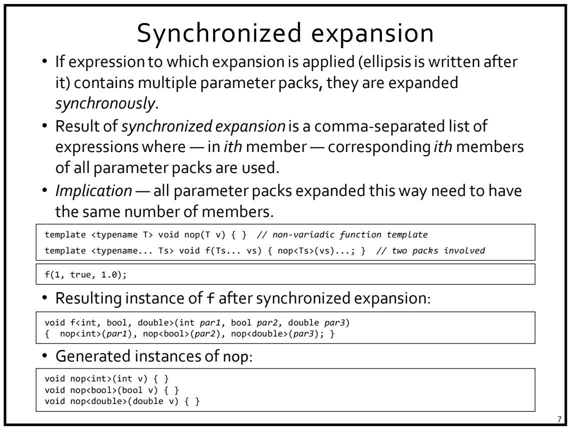
- *Note* — this results in another `h` instance…

```
void h<int, bool, double>(int&&, bool&&, double&&) { }
```

- …plus 3 instances of `move` (not shown); resulting instance of `ff`:

```
void ff<int, bool, double>(int par1, bool par2, double par3)                        {
h<int,bool,double>(std::move<int&>(par1),std::move<bool&>(par2),std::move<double&>(par3)); }
```

6



Synchronized expansion

- If expression to which expansion is applied (ellipsis is written after it) contains multiple parameter packs, they are expanded *synchronously*.
- Result of *synchronized expansion* is a comma-separated list of expressions where — in *ith* member — corresponding *ith* members of all parameter packs are used.
- *Implication* — all parameter packs expanded this way need to have the same number of members.

```
template <typename T> void nop(T v) { }  // non-variadic function template
template <typename... Ts> void f(Ts... vs) { nop<Ts>(vs)...; }  // two packs involved
```

```
f(1, true, 1.0);
```

- Resulting instance of `f` after synchronized expansion:

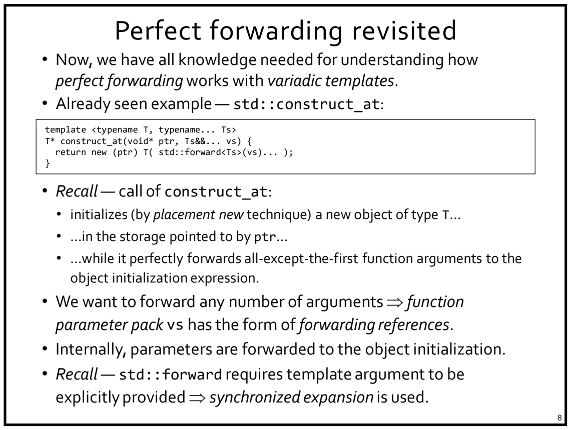
```
void f<int, bool, double>(int par1, bool par2, double par3)
{  nop<int>(par1), nop<bool>(par2), nop<double>(par3); }
```

- Generated instances of nop:

```
void nop<int>(int v) { }
void nop<bool>(bool v) { }
void nop<double>(double v) { }
```

7

7



Perfect forwarding revisited

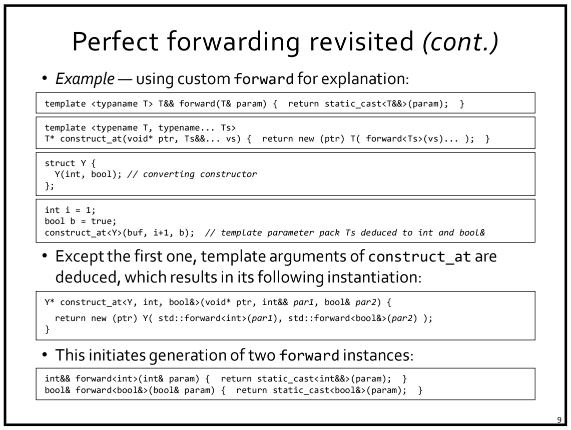
- Now, we have all knowledge needed for understanding how *perfect forwarding* works with *variadic templates*.
- Already seen example — `std::construct_at`:

```
template <typename T, typename... Ts>
T* construct_at(void* ptr, Ts&&... vs) {
  return new (ptr) T( std::forward<Ts>(vs)... );
}
```

- *Recall* — call of `construct_at`:
  - initializes (by *placement new* technique) a new object of type `T`…
  - …in the storage pointed to by `ptr`…
  - …while it perfectly forwards all-except-the-first function arguments to the object initialization expression.
- We want to forward any number of arguments ⇒ *function parameter pack* `vs` has the form of *forwarding references*.
- Internally, parameters are forwarded to the object initialization.
- *Recall* — `std::forward` requires template argument to be explicitly provided ⇒ *synchronized expansion* is used.

8

8



Perfect forwarding revisited *(cont.)*

- *Example* — using custom `forward` for explanation:

```
template <typaname T> T&& forward(T& param) {  return static_cast<T&&>(param);  }
```

```
template <typename T, typename... Ts>
T* construct_at(void* ptr, Ts&&... vs) {  return new (ptr) T( forward<Ts>(vs)... );  }
```

```
struct Y {
  Y(int, bool); // converting constructor
};
```

```
int i = 1;
bool b = true;
construct_at<Y>(buf, i+1, b);  // template parameter pack Ts deduced to int and bool&
```

- Except the first one, template arguments of `construct_at` are deduced, which results in its following instantiation:

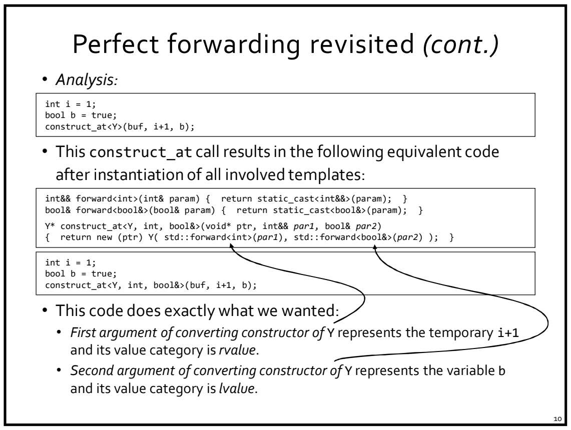
```
Y* construct_at<Y, int, bool&>(void* ptr, int&& par1, bool& par2) {
  return new (ptr) Y( std::forward<int>(par1), std::forward<bool&>(par2) );
}
```

- This initiates generation of two `forward` instances:

```
int&& forward<int>(int& param) {  return static_cast<int&&>(param);  }
bool& forward<bool&>(bool& param) {  return static_cast<bool&>(param);  }
```

9

9



Perfect forwarding revisited *(cont.)*

- *Analysis:*

```
int i = 1;
bool b = true;
construct_at<Y>(buf, i+1, b);
```

- This `construct_at` call results in the following equivalent code after instantiation of all involved templates:

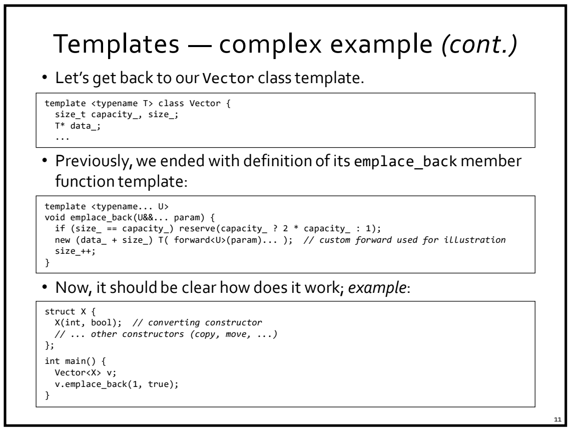
```
int&& forward<int>(int& param) {  return static_cast<int&&>(param);  }
bool& forward<bool&>(bool& param) {  return static_cast<bool&>(param);  }

Y* construct_at<Y, int, bool&>(void* ptr, int&& par1, bool& par2)
{  return new (ptr) Y( std::forward<int>(par1), std::forward<bool&>(par2) );  }
```

```
int i = 1;
bool b = true;
construct_at<Y, int, bool&>(buf, i+1, b);
```

- This code does exactly what we wanted:
  - *First argument of converting constructor of* `Y` represents the temporary `i+1` and its value category is *rvalue*.
  - *Second argument of converting constructor of* `Y` represents the variable `b` and its value category is *lvalue*.

10

10



Templates — complex example *(cont.)*

- Let's get back to our `Vector` class template.

```
template <typename T> class Vector {
  size_t capacity_, size_;
  T* data_;
  ...
```

- Previously, we ended with definition of its `emplace_back` member function template:

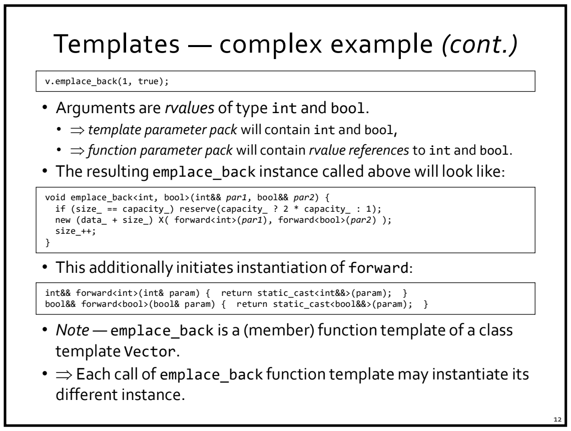
```
template <typename... U>
void emplace_back(U&&... param) {
  if (size_ == capacity_) reserve(capacity_ ? 2 * capacity_ : 1);
  new (data_ + size_) T( forward<U>(param)... );  // custom forward used for illustration
  size_++;
}
```

- Now, it should be clear how does it work; *example*:

```
struct X {
  X(int, bool);  // converting constructor
  // ... other constructors (copy, move, ...)
};

int main() {
  Vector<X> v;
  v.emplace_back(1, true);
}
```

11

11



Templates — complex example *(cont.)*

```
v.emplace_back(1, true);
```

- Arguments are *rvalues* of type `int` and `bool`.
  - ⇒ *template parameter pack* will contain `int` and `bool`,
  - ⇒ *function parameter pack* will contain *rvalue references* to `int` and `bool`.
- The resulting `emplace_back` instance called above will look like:

```
void emplace_back<int, bool>(int&& par1, bool&& par2) {
  if (size_ == capacity_) reserve(capacity_ ? 2 * capacity_ : 1);
  new (data_ + size_) X( forward<int>(par1), forward<bool>(par2) );
  size_++;
}
```

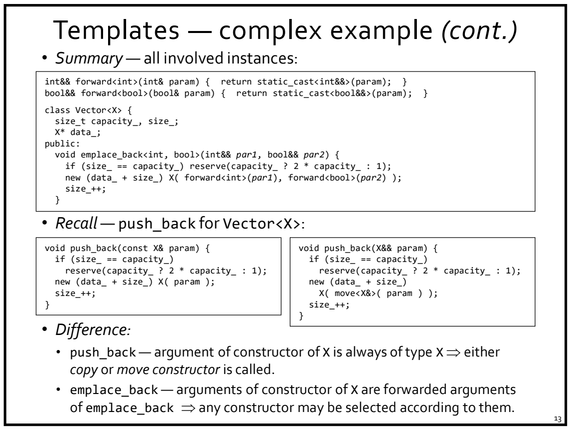
- This additionally initiates instantiation of `forward`:

```
int&& forward<int>(int& param) {  return static_cast<int&&>(param);  }
bool&& forward<bool>(bool& param) {  return static_cast<bool&&>(param);  }
```

- *Note* — `emplace_back` is a (member) function template of a class template `Vector`.
- ⇒ Each call of `emplace_back` function template may instantiate its different instance.

12

12



Templates — complex example *(cont.)*

- *Summary* — all involved instances:

```
int&& forward<int>(int& param) {  return static_cast<int&&>(param);  }
bool&& forward<bool>(bool& param) {  return static_cast<bool&&>(param);  }

class Vector<X> {
  size_t capacity_, size_;
  X* data_;
public:
  void emplace_back<int, bool>(int&& par1, bool&& par2) {
    if (size_ == capacity_) reserve(capacity_ ? 2 * capacity_ : 1);
    new (data_ + size_) X( forward<int>(par1), forward<bool>(par2) );
    size_++;
  }
```

- *Recall* — `push_back` for `Vector<X>`:

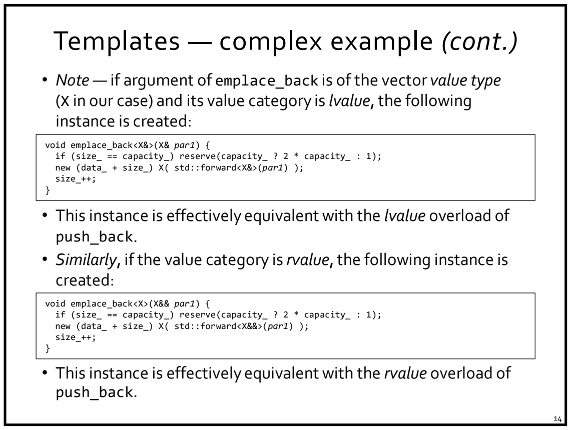
```
void push_back(const X& param) {
  if (size_ == capacity_)
    reserve(capacity_ ? 2 * capacity_ : 1);
  new (data_ + size_) X( param );
  size_++;
}
```

```
void push_back(X&& param) {
  if (size_ == capacity_)
    reserve(capacity_ ? 2 * capacity_ : 1);
  new (data_ + size_)
    X( move<X&>( param ) );
  size_++;
}
```

- *Difference:*
  - `push_back` — argument of constructor of `X` is always of type `X` ⇒ either *copy* or *move constructor* is called.
  - `emplace_back` — arguments of constructor of `X` are forwarded arguments of `emplace_back` ⇒ any constructor may be selected according to them.

13

13



Templates — complex example *(cont.)*

- *Note* — if argument of `emplace_back` is of the vector *value type* (`X` in our case) and its value category is *lvalue*, the following instance is created:

```
void emplace_back<X&>(X& par1) {
  if (size_ == capacity_) reserve(capacity_ ? 2 * capacity_ : 1);
  new (data_ + size_) X( std::forward<X&>(par1) );
  size_++;
}
```

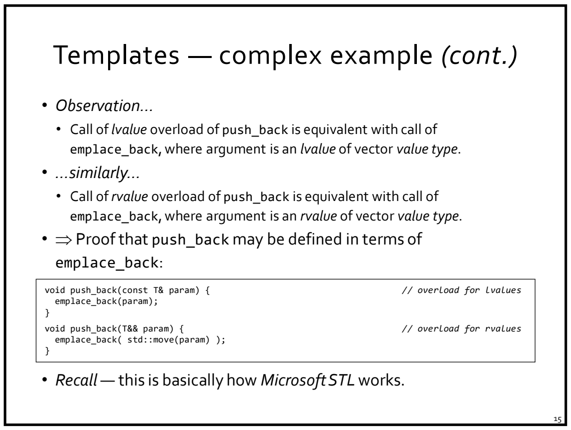
- This instance is effectively equivalent with the *lvalue* overload of `push_back`.
- *Similarly*, if the value category is *rvalue*, the following instance is created:

```
void emplace_back<X>(X&& par1) {
  if (size_ == capacity_) reserve(capacity_ ? 2 * capacity_ : 1);
  new (data_ + size_) X( std::forward<X&&>(par1) );
  size_++;
}
```

- This instance is effectively equivalent with the *rvalue* overload of `push_back`.

14

14



Templates — complex example *(cont.)*

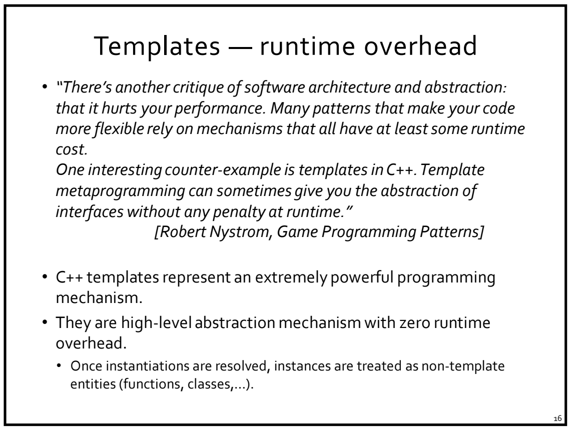
- *Observation…*
  - Call of *lvalue* overload of `push_back` is equivalent with call of `emplace_back`, where argument is an *lvalue* of vector *value type*.
- *…similarly…*
  - Call of *rvalue* overload of `push_back` is equivalent with call of `emplace_back`, where argument is an *rvalue* of vector *value type*.
- ⇒ Proof that `push_back` may be defined in terms of `emplace_back`:

```
void push_back(const T& param) {                                    // overload for lvalues
  emplace_back(param);
}
void push_back(T&& param) {                                         // overload for rvalues
  emplace_back( std::move(param) );
}
```

- *Recall* — this is basically how *Microsoft STL* works.

15

15

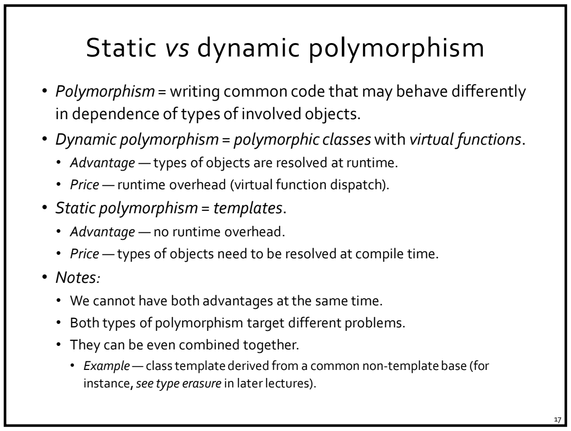


Templates — runtime overhead

- *"There's another critique of software architecture and abstraction: that it hurts your performance. Many patterns that make your code more flexible rely on mechanisms that all have at least some runtime cost.*
  One interesting counter-example is templates in C++. Template metaprogramming can sometimes give you the abstraction of interfaces without any penalty at runtime."
  [Robert Nystrom, Game Programming Patterns]
- C++ templates represent an extremely powerful programming mechanism.
- They are high-level abstraction mechanism with zero runtime overhead.
  - Once instantiations are resolved, instances are treated as non-template entities (functions, classes,…).

16

16

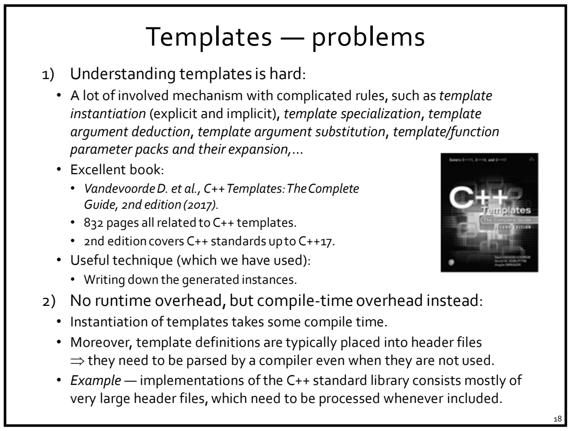


Static *vs* dynamic polymorphism

- *Polymorphism* = writing common code that may behave differently in dependence of types of involved objects.
- *Dynamic polymorphism* = *polymorphic classes* with *virtual functions*.
  - *Advantage* — types of objects are resolved at runtime.
  - *Price* — runtime overhead (virtual function dispatch).
- *Static polymorphism* = *templates*.
  - *Advantage* — no runtime overhead.
  - *Price* — types of objects need to be resolved at compile time.
- *Notes:*
  - We cannot have both advantages at the same time.
  - Both types of polymorphism target different problems.
  - They can be even combined together.
    - *Example* — class template derived from a common non-template base (for instance, *see type erasure* in later lectures).

17

17

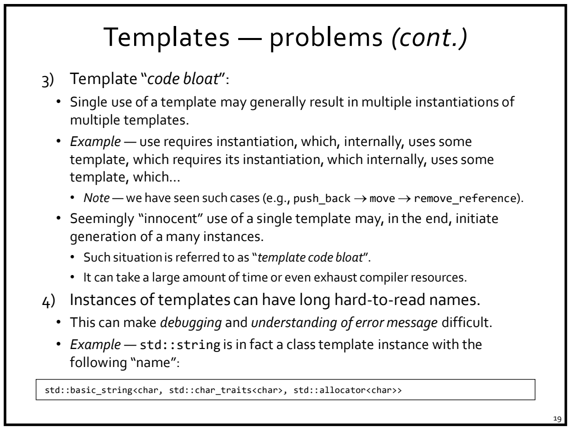


Templates — problems

1) Understanding templates is hard:
   - A lot of involved mechanism with complicated rules, such as *template instantiation* (explicit and implicit), *template specialization*, *template argument deduction*, *template argument substitution*, *template/function parameter packs and their expansion*,…
   - Excellent book:
     - *Vandevoorde D. et al., C++ Templates: The Complete Guide, 2nd edition (2017).*
     - 832 pages all related to C++ templates.
     - 2nd edition covers C++ standards up to C++17.
   - Useful technique (which we have used):
     - Writing down the generated instances.
2) No runtime overhead, but compile-time overhead instead:
   - Instantiation of templates takes some compile time.
   - Moreover, template definitions are typically placed into header files ⇒ they need to be parsed by a compiler even when they are not used.
   - *Example* — implementations of the C++ standard library consists mostly of very large header files, which need to be processed whenever included.

18

18

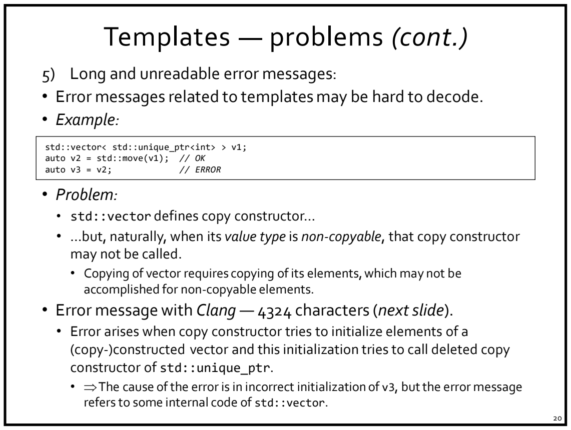


Templates — problems *(cont.)*

3) Template "*code bloat*":
   - Single use of a template may generally result in multiple instantiations of multiple templates.
   - *Example* — use requires instantiation, which, internally, uses some template, which requires its instantiation, which internally, uses some template, which…
     - *Note* — we have seen such cases (e.g., push_back → move → `remove_reference`).
   - Seemingly "innocent" use of a single template may, in the end, initiate generation of a many instances.
     - Such situation is referred to as "*template code bloat*".
     - It can take a large amount of time or even exhaust compiler resources.
4) Instances of templates can have long hard-to-read names.
   - This can make *debugging* and *understanding of error message* difficult.
   - *Example* — `std::string` is in fact a class template instance with the following "name":

```
std::basic_string<char, std::char_traits<char>, std::allocator<char>>
```

19



Templates — problems *(cont.)*

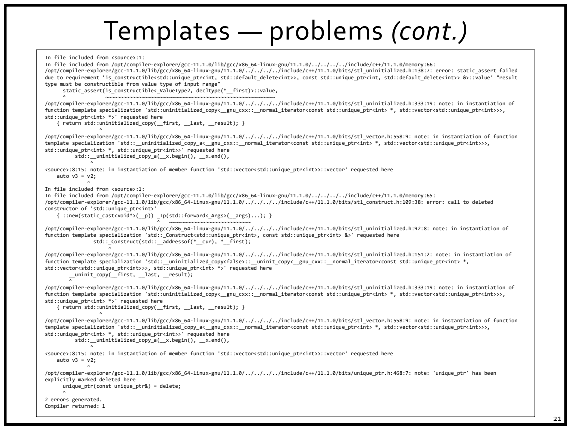
5) Long and unreadable error messages:

- Error messages related to templates may be hard to decode.
- *Example:*

```
std::vector< std::unique_ptr<int> > v1;
auto v2 = std::move(v1);  // OK
auto v3 = v2;             // ERROR
```

- *Problem:*
  - `std::vector` defines copy constructor…
  - …but, naturally, when its *value type* is *non-copyable*, that copy constructor may not be called.
    - Copying of vector requires copying of its elements, which may not be accomplished for non-copyable elements.
- Error message with *Clang* — 4324 characters (*next slide*).
  - Error arises when copy constructor tries to initialize elements of a (copy-)constructed vector and this initialization tries to call deleted copy constructor of `std::unique_ptr`.
    - ⇒ The cause of the error is in incorrect initialization of v3, but the error message refers to some internal code of `std::vector`.

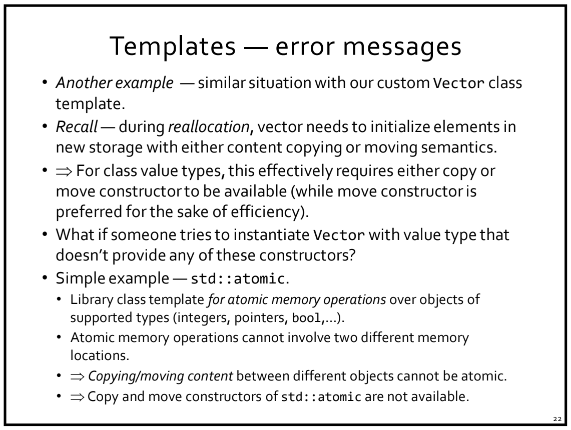
20



Templates — problems *(cont.)*

```
In file included from <source>:1:
In file included from /opt/compiler-explorer/gcc-11.1.0/lib/gcc/x86_64-linux-gnu/11.1.0/../../../../include/c++/11.1.0/memory:66:
/opt/compiler-explorer/gcc-11.1.0/lib/gcc/x86_64-linux-gnu/11.1.0/../../../../include/c++/11.1.0/bits/stl_uninitialized.h:138:7: error: static_assert failed
due to requirement 'is_constructible<std::unique_ptr<int, std::default_delete<int>>, const std::unique_ptr<int, std::default_delete<int>> &>::value' "result
type must be constructible from value type of input range"
      static_assert(is_constructible<_ValueType2, decltype(*__first)>::value,
      ^             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/opt/compiler-explorer/gcc-11.1.0/lib/gcc/x86_64-linux-gnu/11.1.0/../../../../include/c++/11.1.0/bits/stl_uninitialized.h:333:19: note: in instantiation of
function template specialization 'std::uninitialized_copy<__gnu_cxx::__normal_iterator<const std::unique_ptr<int> *, std::vector<std::unique_ptr<int>>>,
std::unique_ptr<int> *>' requested here
    { return std::uninitialized_copy(__first, __last, __result); }
                  ^
/opt/compiler-explorer/gcc-11.1.0/lib/gcc/x86_64-linux-gnu/11.1.0/../../../../include/c++/11.1.0/bits/stl_vector.h:558:9: note: in instantiation of function
template specialization 'std::__uninitialized_copy_a<__gnu_cxx::__normal_iterator<const std::unique_ptr<int> *, std::vector<std::unique_ptr<int>>>,
std::unique_ptr<int> *, std::unique_ptr<int>>' requested here
          std::__uninitialized_copy_a(__x.begin(), __x.end(),
               ^
<source>:8:15: note: in instantiation of member function 'std::vector<std::unique_ptr<int>>::vector' requested here
    auto v3 = v2;
              ^
In file included from <source>:1:
In file included from /opt/compiler-explorer/gcc-11.1.0/lib/gcc/x86_64-linux-gnu/11.1.0/../../../../include/c++/11.1.0/memory:65:
/opt/compiler-explorer/gcc-11.1.0/lib/gcc/x86_64-linux-gnu/11.1.0/../../../../include/c++/11.1.0/bits/stl_construct.h:109:38: error: call to deleted
constructor of 'std::unique_ptr<int>'
    { ::new(static_cast<void*>(__p)) _Tp(std::forward<_Args>(__args)...); }
                                     ^   ~~~~~~~~~~~~~~~~~~~~~~~~~~~
/opt/compiler-explorer/gcc-11.1.0/lib/gcc/x86_64-linux-gnu/11.1.0/../../../../include/c++/11.1.0/bits/stl_uninitialized.h:92:8: note: in instantiation of
function template specialization 'std::_Construct<std::unique_ptr<int>, const std::unique_ptr<int> &>' requested here
                std::_Construct(std::__addressof(*__cur), *__first);
                     ^
/opt/compiler-explorer/gcc-11.1.0/lib/gcc/x86_64-linux-gnu/11.1.0/../../../../include/c++/11.1.0/bits/stl_uninitialized.h:151:2: note: in instantiation of
function template specialization 'std::__uninitialized_copy<false>::__uninit_copy<__gnu_cxx::__normal_iterator<const std::unique_ptr<int> *,
std::vector<std::unique_ptr<int>>>, std::unique_ptr<int> *>' requested here
        __uninit_copy(__first, __last, __result);
        ^
/opt/compiler-explorer/gcc-11.1.0/lib/gcc/x86_64-linux-gnu/11.1.0/../../../../include/c++/11.1.0/bits/stl_uninitialized.h:333:19: note: in instantiation of
function template specialization 'std::uninitialized_copy<__gnu_cxx::__normal_iterator<const std::unique_ptr<int> *, std::vector<std::unique_ptr<int>>>,
std::unique_ptr<int> *>' requested here
    { return std::uninitialized_copy(__first, __last, __result); }
                  ^
/opt/compiler-explorer/gcc-11.1.0/lib/gcc/x86_64-linux-gnu/11.1.0/../../../../include/c++/11.1.0/bits/stl_vector.h:558:9: note: in instantiation of function
template specialization 'std::__uninitialized_copy_a<__gnu_cxx::__normal_iterator<const std::unique_ptr<int> *, std::vector<std::unique_ptr<int>>>,
std::unique_ptr<int> *, std::unique_ptr<int>>' requested here
          std::__uninitialized_copy_a(__x.begin(), __x.end(),
               ^
<source>:8:15: note: in instantiation of member function 'std::vector<std::unique_ptr<int>>::vector' requested here
    auto v3 = v2;
              ^
/opt/compiler-explorer/gcc-11.1.0/lib/gcc/x86_64-linux-gnu/11.1.0/../../../../include/c++/11.1.0/bits/unique_ptr.h:468:7: note: 'unique_ptr' has been
explicitly marked deleted here
      unique_ptr(const unique_ptr&) = delete;
      ^
2 errors generated.
Compiler returned: 1
```

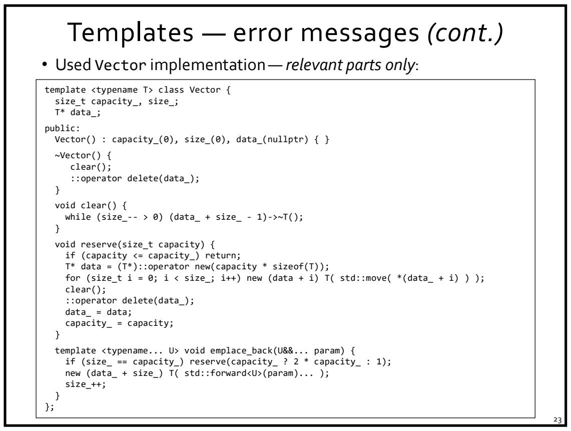
21



Templates — error messages

- *Another example* — similar situation with our custom `Vector` class template.
- *Recall* — during *reallocation*, vector needs to initialize elements in new storage with either content copying or moving semantics.
- ⇒ For class value types, this effectively requires either copy or move constructor to be available (while move constructor is preferred for the sake of efficiency).
- What if someone tries to instantiate `Vector` with value type that doesn't provide any of these constructors?
- Simple example — `std::atomic`.
  - Library class template *for atomic memory operations* over objects of supported types (integers, pointers, bool,…).
  - Atomic memory operations cannot involve two different memory locations.
  - ⇒ *Copying/moving content* between different objects cannot be atomic.
  - ⇒ Copy and move constructors of `std::atomic` are not available.

22

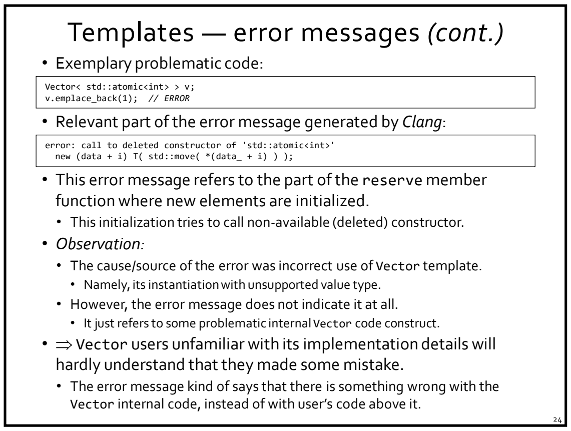


Templates — error messages *(cont.)*

- Used `Vector` implementation — *relevant parts only*:

```
template <typename T> class Vector {
  size_t capacity_, size_;
  T* data_;
public:
  Vector() : capacity_(0), size_(0), data_(nullptr) { }
  ~Vector() {
     clear();
     ::operator delete(data_);
  }
  void clear() {
    while (size_-- > 0) (data_ + size_ - 1)->~T();
  }
  void reserve(size_t capacity) {
    if (capacity <= capacity_) return;
    T* data = (T*)::operator new(capacity * sizeof(T));
    for (size_t i = 0; i < size_; i++) new (data + i) T( std::move( *(data_ + i) ) );
    clear();
    ::operator delete(data_);
    data_ = data;
    capacity_ = capacity;
  }
  template <typename... U> void emplace_back(U&&... param) {
    if (size_ == capacity_) reserve(capacity_ ? 2 * capacity_ : 1);
    new (data_ + size_) T( std::forward<U>(param)... );
    size_++;
  }
};
```

23



Templates — error messages *(cont.)*

- Exemplary problematic code:

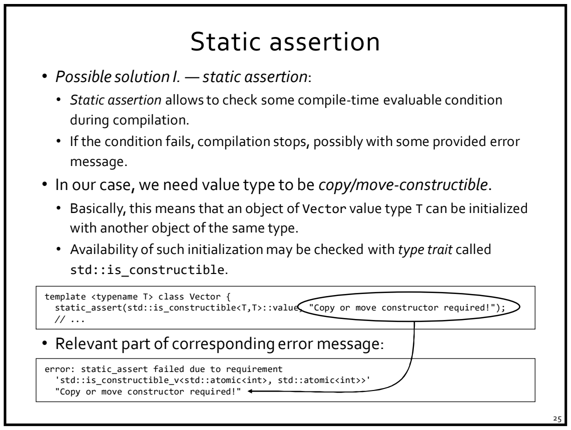
```
Vector< std::atomic<int> > v;
v.emplace_back(1);  // ERROR
```

- Relevant part of the error message generated by *Clang*:

```
error: call to deleted constructor of 'std::atomic<int>'
  new (data + i) T( std::move( *(data_ + i) ) );
```

- This error message refers to the part of the `reserve` member function where new elements are initialized.
  - This initialization tries to call non-available (deleted) constructor.
- *Observation:*
  - The cause/source of the error was incorrect use of `Vector` template.
    - Namely, its instantiation with unsupported value type.
  - However, the error message does not indicate it at all.
    - It just refers to some problematic internal `Vector` code construct.
- ⇒ `Vector` users unfamiliar with its implementation details will hardly understand that they made some mistake.
  - The error message kind of says that there is something wrong with the `Vector` internal code, instead of with user's code above it.

24



Static assertion

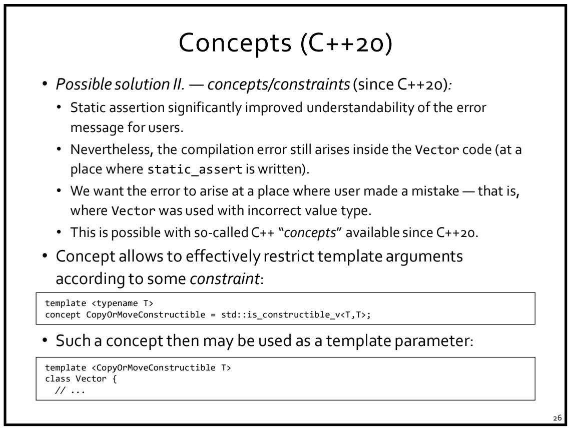
- *Possible solution I. — static assertion*:
  - *Static assertion* allows to check some compile-time evaluable condition during compilation.
  - If the condition fails, compilation stops, possibly with some provided error message.
- In our case, we need value type to be *copy/move-constructible*.
  - Basically, this means that an object of `Vector` value type `T` can be initialized with another object of the same type.
  - Availability of such initialization may be checked with *type trait* called `std::is_constructible`.

```
template <typename T> class Vector {
  static_assert(std::is_constructible<T,T>::value, "Copy or move constructor required!");
  // ...
```

- Relevant part of corresponding error message:

```
error: static_assert failed due to requirement
  'std::is_constructible_v<std::atomic<int>, std::atomic<int>>'
  "Copy or move constructor required!"
```

25



Concepts (C++20)

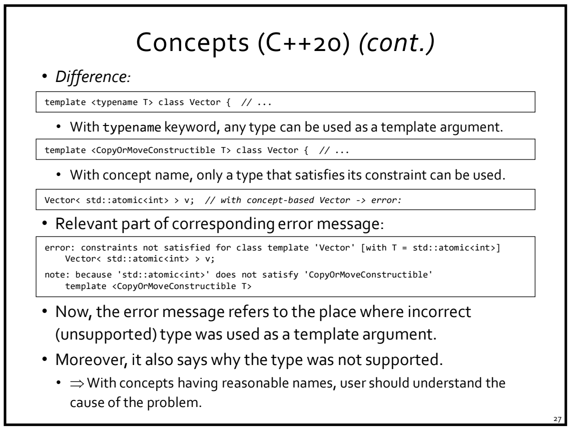
- *Possible solution II. — concepts/constraints* (since C++20)*:*
  - Static assertion significantly improved understandability of the error message for users.
  - Nevertheless, the compilation error still arises inside the `Vector` code (at a place where `static_assert` is written).
  - We want the error to arise at a place where user made a mistake — that is, where `Vector` was used with incorrect value type.
  - This is possible with so-called C++ "*concepts*" available since C++20.
- Concept allows to effectively restrict template arguments according to some *constraint*:

```
template <typename T>
concept CopyOrMoveConstructible = std::is_constructible_v<T,T>;
```

- Such a concept then may be used as a template parameter:

```
template <CopyOrMoveConstructible T>
class Vector {
  // ...
```

26



Concepts (C++20) *(cont.)*

- *Difference:*

```
template <typename T> class Vector {  // ...
```

  - With `typename` keyword, any type can be used as a template argument.

```
template <CopyOrMoveConstructible T> class Vector {  // ...
```

  - With concept name, only a type that satisfies its constraint can be used.

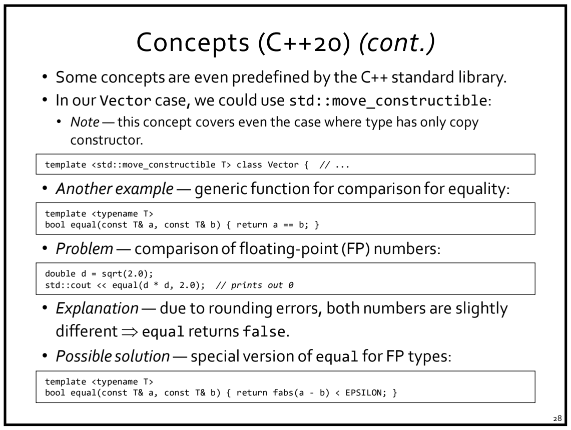
```
Vector< std::atomic<int> > v;  // with concept-based Vector -> error:
```

- Relevant part of corresponding error message:

```
error: constraints not satisfied for class template 'Vector' [with T = std::atomic<int>]
    Vector< std::atomic<int> > v;

note: because 'std::atomic<int>' does not satisfy 'CopyOrMoveConstructible'
    template <CopyOrMoveConstructible T>
```

- Now, the error message refers to the place where incorrect (unsupported) type was used as a template argument.
- Moreover, it also says why the type was not supported.
  - ⇒ With concepts having reasonable names, user should understand the cause of the problem.

27



Concepts (C++20) *(cont.)*

- Some concepts are even predefined by the C++ standard library.
- In our `Vector` case, we could use `std::move_constructible`:
  - *Note* — this concept covers even the case where type has only copy constructor.

```
template <std::move_constructible T> class Vector {  // ...
```

- *Another example* — generic function for comparison for equality:

```
template <typename T>
bool equal(const T& a, const T& b) { return a == b; }
```

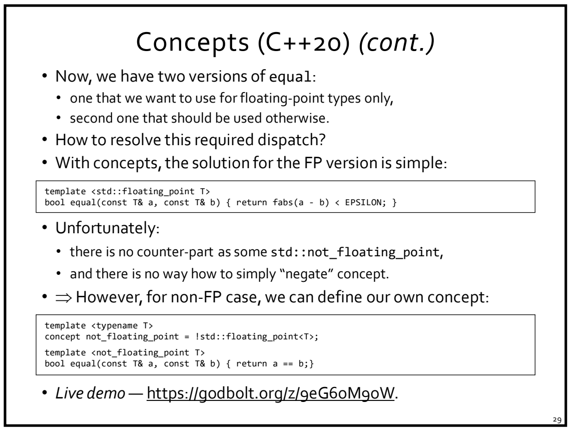
- *Problem* — comparison of floating-point (FP) numbers:

```
double d = sqrt(2.0);
std::cout << equal(d * d, 2.0);  // prints out 0
```

- *Explanation* — due to rounding errors, both numbers are slightly different ⇒ `equal` returns `false`.
- *Possible solution* — special version of `equal` for FP types:

```
template <typename T>
bool equal(const T& a, const T& b) { return fabs(a - b) < EPSILON; }
```

28



Concepts (C++20) *(cont.)*

- Now, we have two versions of `equal`:
  - one that we want to use for floating-point types only,
  - second one that should be used otherwise.
- How to resolve this required dispatch?
- With concepts, the solution for the FP version is simple:

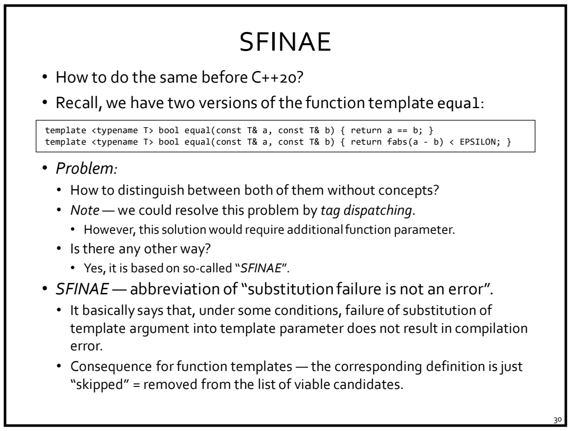
```
template <std::floating_point T>
bool equal(const T& a, const T& b) { return fabs(a - b) < EPSILON; }
```

- Unfortunately:
  - there is no counter-part as some `std::not_floating_point`,
  - and there is no way how to simply "negate" concept.
- ⇒ However, for non-FP case, we can define our own concept:

```
template <typename T>
concept not_floating_point = !std::floating_point<T>;

template <not_floating_point T>
bool equal(const T& a, const T& b) { return a == b;}
```

- *Live demo* — https://godbolt.org/z/9eG6oM9oW.

29



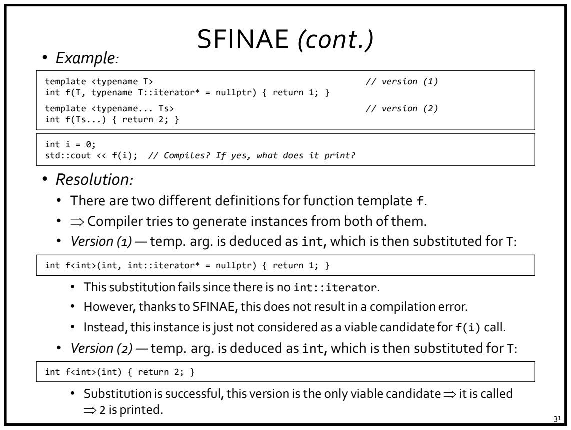
SFINAE

- How to do the same before C++20?
- Recall, we have two versions of the function template `equal`:

```
template <typename T> bool equal(const T& a, const T& b) { return a == b; }
template <typename T> bool equal(const T& a, const T& b) { return fabs(a - b) < EPSILON; }
```

- *Problem:*
  - How to distinguish between both of them without concepts?
  - *Note* — we could resolve this problem by *tag dispatching*.
    - However, this solution would require additional function parameter.
  - Is there any other way?
    - Yes, it is based on so-called "*SFINAE*".
- *SFINAE* — abbreviation of "substitution failure is not an error".
  - It basically says that, under some conditions, failure of substitution of template argument into template parameter does not result in compilation error.
  - Consequence for function templates — the corresponding definition is just "skipped" = removed from the list of viable candidates.

30



SFINAE *(cont.)*

- *Example:*

```
template <typename T>                                         // version (1)
int f(T, typename T::iterator* = nullptr) { return 1; }

template <typename... Ts>                                     // version (2)
int f(Ts...) { return 2; }
```

```
int i = 0;
std::cout << f(i);  // Compiles? If yes, what does it print?
```

- *Resolution:*
  - There are two different definitions for function template `f`.
  - ⇒ Compiler tries to generate instances from both of them.
  - *Version (1)* — temp. arg. is deduced as `int`, which is then substituted for `T`:

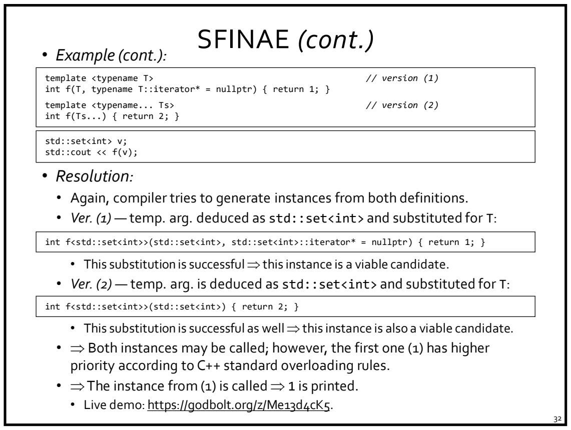
```
int f<int>(int, int::iterator* = nullptr) { return 1; }
```

    - This substitution fails since there is no `int::iterator`.
    - However, thanks to SFINAE, this does not result in a compilation error.
    - Instead, this instance is just not considered as a viable candidate for `f(i)` call.
  - *Version (2)* — temp. arg. is deduced as `int`, which is then substituted for `T`:

```
int f<int>(int) { return 2; }
```

    - Substitution is successful, this version is the only viable candidate ⇒ it is called ⇒ 2 is printed.

31



SFINAE *(cont.)*

- *Example (cont.):*

```
template <typename T>                                         // version (1)
int f(T, typename T::iterator* = nullptr) { return 1; }

template <typename... Ts>                                     // version (2)
int f(Ts...) { return 2; }
```

```
std::set<int> v;
std::cout << f(v);
```

- *Resolution:*
  - Again, compiler tries to generate instances from both definitions.
  - *Ver. (1)* — temp. arg. deduced as `std::set<int>` and substituted for `T`:

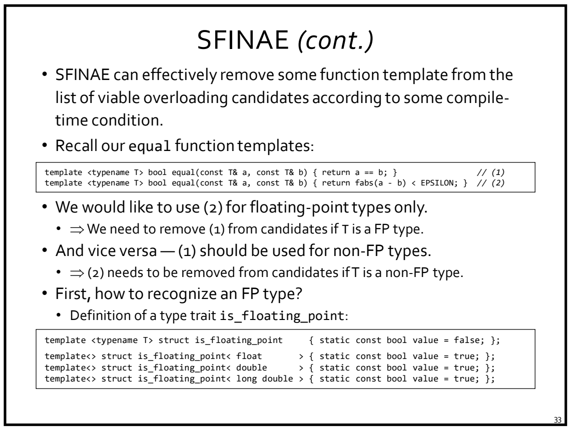
```
int f<std::set<int>>(std::set<int>, std::set<int>::iterator* = nullptr) { return 1; }
```

    - This substitution is successful ⇒ this instance is a viable candidate.
  - *Ver. (2)* — temp. arg. is deduced as `std::set<int>` and substituted for `T`:

```
int f<std::set<int>>(std::set<int>) { return 2; }
```

    - This substitution is successful as well ⇒ this instance is also a viable candidate.
  - ⇒ Both instances may be called; however, the first one (1) has higher priority according to C++ standard overloading rules.
  - ⇒ The instance from (1) is called ⇒ `1` is printed.
    - Live demo: https://godbolt.org/z/Me13d4cK5.

32



SFINAE *(cont.)*

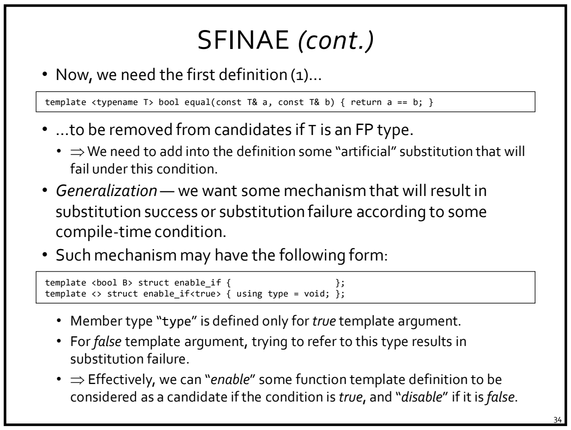
- SFINAE can effectively remove some function template from the list of viable overloading candidates according to some compile-time condition.
- Recall our `equal` function templates:

```
template <typename T> bool equal(const T& a, const T& b) { return a == b; }               // (1)
template <typename T> bool equal(const T& a, const T& b) { return fabs(a - b) < EPSILON; }  // (2)
```

- We would like to use (2) for floating-point types only.
  - ⇒ We need to remove (1) from candidates if `T` is a FP type.
- And vice versa — (1) should be used for non-FP types.
  - ⇒ (2) needs to be removed from candidates if T is a non-FP type.
- First, how to recognize an FP type?
  - Definition of a type trait `is_floating_point`:

```
template <typename T> struct is_floating_point     { static const bool value = false; };

template<> struct is_floating_point< float       > { static const bool value = true; };
template<> struct is_floating_point< double      > { static const bool value = true; };
template<> struct is_floating_point< long double > { static const bool value = true; };
```

33



SFINAE *(cont.)*

- Now, we need the first definition (1)…

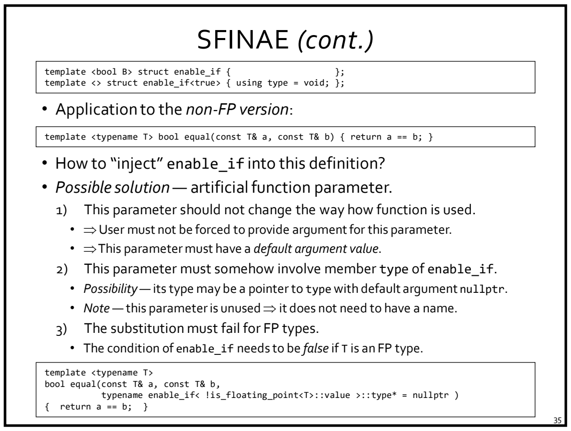
```
template <typename T> bool equal(const T& a, const T& b) { return a == b; }
```

- …to be removed from candidates if `T` is an FP type.
  - ⇒ We need to add into the definition some "artificial" substitution that will fail under this condition.
- *Generalization* — we want some mechanism that will result in substitution success or substitution failure according to some compile-time condition.
- Such mechanism may have the following form:

```
template <bool B> struct enable_if {                    };
template <> struct enable_if<true> { using type = void; };
```

  - Member type "`type`" is defined only for *true* template argument.
  - For *false* template argument, trying to refer to this type results in substitution failure.
  - ⇒ Effectively, we can "*enable*" some function template definition to be considered as a candidate if the condition is *true*, and "*disable*" if it is *false*.

34



SFINAE *(cont.)*

```
template <bool B> struct enable_if {                    };
template <> struct enable_if<true> { using type = void; };
```

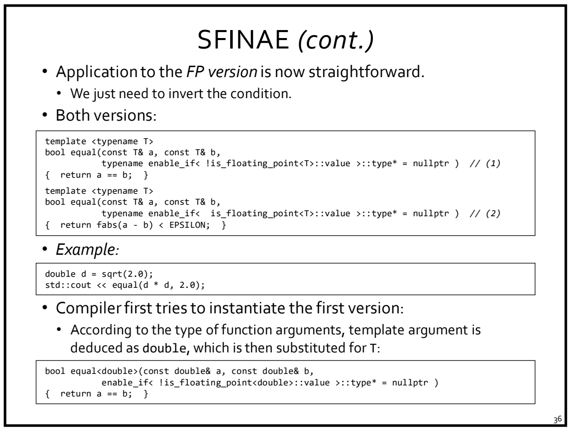
- Application to the *non-FP version*:

```
template <typename T> bool equal(const T& a, const T& b) { return a == b; }
```

- How to "inject" `enable_if` into this definition?
- *Possible solution* — artificial function parameter.
  1) This parameter should not change the way how function is used.
     - ⇒ User must not be forced to provide argument for this parameter.
     - ⇒ This parameter must have a *default argument value*.
  2) This parameter must somehow involve member `type` of `enable_if`.
     - *Possibility* — its type may be a pointer to `type` with default argument `nullptr`.
     - *Note* — this parameter is unused ⇒ it does not need to have a name.
  3) The substitution must fail for FP types.
     - The condition of `enable_if` needs to be *false* if `T` is an FP type.

```
template <typename T>
bool equal(const T& a, const T& b,
           typename enable_if< !is_floating_point<T>::value >::type* = nullptr )
{  return a == b;  }
```

35



SFINAE *(cont.)*

- Application to the *FP version* is now straightforward.
  - We just need to invert the condition.
- Both versions:

```
template <typename T>
bool equal(const T& a, const T& b,
           typename enable_if< !is_floating_point<T>::value >::type* = nullptr )  // (1)
{  return a == b;  }

template <typename T>
bool equal(const T& a, const T& b,
           typename enable_if<  is_floating_point<T>::value >::type* = nullptr )  // (2)
{  return fabs(a - b) < EPSILON;  }
```

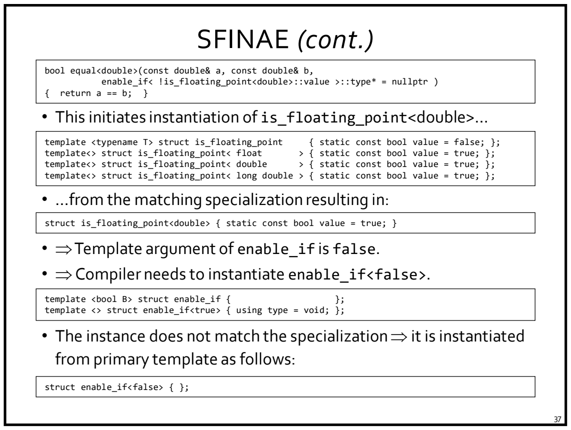
- *Example:*

```
double d = sqrt(2.0);
std::cout << equal(d * d, 2.0);
```

- Compiler first tries to instantiate the first version:
  - According to the type of function arguments, template argument is deduced as `double`, which is then substituted for `T`:

```
bool equal<double>(const double& a, const double& b,
           enable_if< !is_floating_point<double>::value >::type* = nullptr )
{  return a == b;  }
```

36



SFINAE *(cont.)*

```
bool equal<double>(const double& a, const double& b,
          enable_if< !is_floating_point<double>::value >::type* = nullptr )
{  return a == b;  }
```

- This initiates instantiation of `is_floating_point`<double>…

```
template <typename T> struct is_floating_point     { static const bool value = false; };
template<> struct is_floating_point< float       > { static const bool value = true; };
template<> struct is_floating_point< double      > { static const bool value = true; };
template<> struct is_floating_point< long double > { static const bool value = true; };
```

- …from the matching specialization resulting in:

```
struct is_floating_point<double> { static const bool value = true; }
```

- ⇒ Template argument of `enable_if` is `false`.
- ⇒ Compiler needs to instantiate `enable_if<false>`.

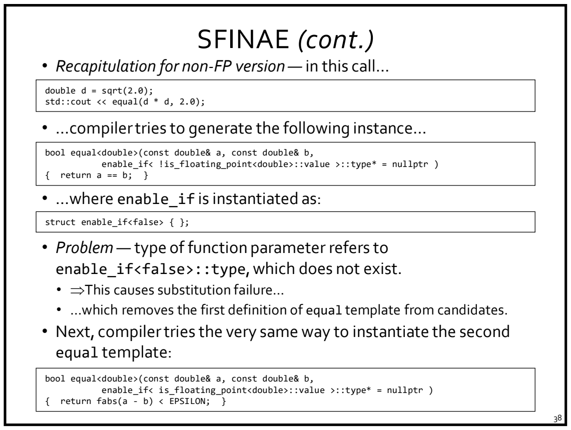
```
template <bool B> struct enable_if {                     };
template <> struct enable_if<true> { using type = void; };
```

- The instance does not match the specialization ⇒ it is instantiated from primary template as follows:

```
struct enable_if<false> { };
```

37

37



SFINAE *(cont.)*

- *Recapitulation for non-FP version* — in this call…

```
double d = sqrt(2.0);
std::cout << equal(d * d, 2.0);
```

- …compiler tries to generate the following instance…

```
bool equal<double>(const double& a, const double& b,
          enable_if< !is_floating_point<double>::value >::type* = nullptr )
{  return a == b;  }
```

- …where `enable_if` is instantiated as:

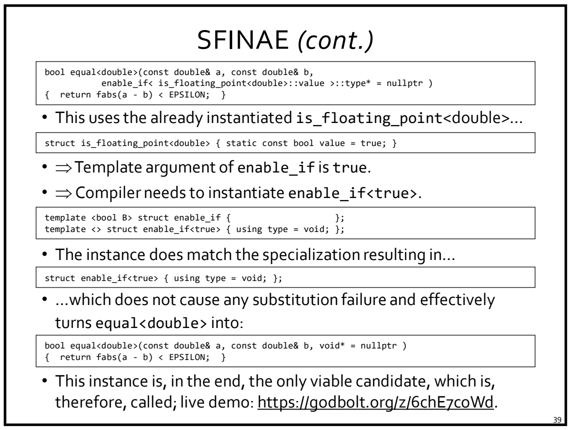
```
struct enable_if<false> { };
```

- *Problem* — type of function parameter refers to `enable_if<false>::type`, which does not exist.
  - ⇒This causes substitution failure…
  - …which removes the first definition of `equal` template from candidates.
- Next, compiler tries the very same way to instantiate the second `equal` template:

```
bool equal<double>(const double& a, const double& b,
          enable_if< is_floating_point<double>::value >::type* = nullptr )
{  return fabs(a - b) < EPSILON;  }
```

38

38



SFINAE *(cont.)*

```
bool equal<double>(const double& a, const double& b,
          enable_if< is_floating_point<double>::value >::type* = nullptr )
{  return fabs(a - b) < EPSILON;  }
```

- This uses the already instantiated `is_floating_point`<double>…

```
struct is_floating_point<double> { static const bool value = true; }
```

- ⇒ Template argument of `enable_if` is `true`.
- ⇒ Compiler needs to instantiate `enable_if<true>`.

```
template <bool B> struct enable_if {                     };
template <> struct enable_if<true> { using type = void; };
```

- The instance does match the specialization resulting in…

```
struct enable_if<true> { using type = void; };
```

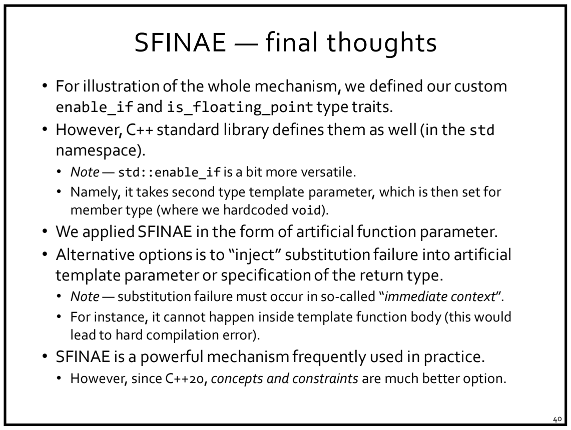
- …which does not cause any substitution failure and effectively turns `equal<double>` into:

```
bool equal<double>(const double& a, const double& b, void* = nullptr )
{  return fabs(a - b) < EPSILON;  }
```

- This instance is, in the end, the only viable candidate, which is, therefore, called; live demo: https://godbolt.org/z/6chE7coWd.

39

39



SFINAE — final thoughts

- For illustration of the whole mechanism, we defined our custom `enable_if` and `is_floating_point` type traits.
- However, C++ standard library defines them as well (in the `std` namespace).
  - *Note* — `std::enable_if` is a bit more versatile.
  - Namely, it takes second type template parameter, which is then set for member type (where we hardcoded `void`).
- We applied SFINAE in the form of artificial function parameter.
- Alternative options is to "inject" substitution failure into artificial template parameter or specification of the return type.
  - *Note* — substitution failure must occur in so-called "*immediate context*".
  - For instance, it cannot happen inside template function body (this would lead to hard compilation error).
- SFINAE is a powerful mechanism frequently used in practice.
  - However, since C++20, *concepts and constraints* are much better option.

40

40