Effective C++ Programming

NIE-EPC (v. 2021):

EMPLACE SEMANTICS, PERFECT FORWARDING, FUNCTION CALL DISPATCHING

© 2021 DANIEL LANGR, ČVUT (CTU) FIT

Implicit conversions

- Consider the following *example*:

```
struct X {
  X(int) { std::cout << "converting constructor\n"; }
  X(X&&) { std::cout << "move constructor\n"; }
};
```

```
std::vector< X > v;
v.push_back( X(1) );
```

- Here, `std::vector<X>::push_back(X&&)` is called where:
  - *type of parameter* is X&&,
  - *argument* is an *expression* that:
    - refers to an object of type X,
    - its value category is *rvalue*.
- *Modified example:*

```
std::vector< X > v;

v.push_back( 1 );
```

- What has changed?
  - Argument is an *expression* that refers to an object of type **other** than X.
  - (Namely, it refers to an object of type `int` and its category is *rvalue*.)

Implicit conversions *(cont.)*

- When:
  - initialization requires expression of some type `T1`, and
  - the type of the initialization expression is different than `T1` (say, `T2`),
- then a compiler will try to "*implicitly*" **convert** the initialization expression into an object of type `T1`.
- *In our case:*
  - *Parameter* of `push_back` (of type `X&&`) expects expression of type `X`.
  - *Initialization expression* (argument) `1` has type `int`.
- ⇒ Compiler will try to *implicitly* convert `1` to an object of type `X`.
- Is such conversion possible?
  - **Yes, it is**, due to the availability of matching non-explicit converting constructor (constructor with a parameter of type `int`):

```
X(int) { std::cout << "converting constructor\n"; }
```

- *Note* — `explicit` *specifier* does disallow such implicit conversions.
  - Conversions are then still possible but only explicitly written in code.

Emplacing-back into vector

- *Consequence* — both two options are *effectively equivalent*:

```
v.push_back( X(1) );  // option #1
v.push_back(   1  );  // option #2
```

  - In #2, the temporary object of type X created automatically due to an implicit conversion.
- What constructors of X are involved in…?

```
std::vector<X> v;
v.push_back(1);
```

1) A temporary of type X is created from 1 ⇒ *converting constructor*.
2) Then, push_back internally initializes a new element in its storage and move content into it from the temporary ⇒ *move constructor*.

- The program output agrees with that:

```
converting constructor
move constructor
```

- *Live demo* — https://godbolt.org/z/3Y3KaTnEx.

Emplacing-back into vector *(cont.)*

```
std::vector<X> v;
v.push_back(1);
```

```
converting constructor
move constructor
```

- There is something “wrong” with that from the perspective of *performance/efficiency*.
  - Why to create a *temporary* and then, immediately, *move content* from it?
  - Wouldn’t be better to create the *vector element* itself with the *converting constructor* “directly” *from the argument* `1`?
- Clearly, `push_back` itself cannot provide that — it accepts only arguments of type `X`.
- *Instead* — we want a vector member function that
  1) accepts argument(s) of *any type* and *any value category*,
  2) initializes a new vector element with *expression* that:
     - represents the same object(s) as the function argument(s),
     - has the same value category(ies) as the function argument(s).

Emplacing-back into vector *(cont.)*

- Such functionality is called "*emplacing*" and `std::vector` provides corresponding function called `emplace_back`.
- *Our first attempt* — *single-argument* implementation.
- We want to accept an argument of *any type*…
  - ⇒ the designed function actually needs to be a *function template*;
- …and *any value category*, which will be *recognizable*…
  - ⇒ the function parameter needs to be a *forwarding reference*.

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

template <typename U>
void emplace_back(U&& param) {
  if (size_ == capacity_)
    reserve(capacity_ ? 2 * capacity_ : 1);

  new (data_ + size_) T( ??? );

  size_++;
}
```

- *Remains to be resolved* — how *initialization expression* should look like?

Emplacing-back into vector *(cont.)*

- *Example:*

```
template <typename U>
void emplace_back(U&& param) {
  ...
  new (data_ + size_) T( ??? );
```

```
std::vector<X> v;
int i = 1;
v.emplace_back( i );    // (1)
v.emplace_back( i+1 );  // (2)
```

- In case (1), we need to initialize vector element with an expression that:
  a) refers to the same object as the argument ⇒ to variable `i`,
  b) has the same value category as the argument ⇒ *lvalue*.
- In case (2), we need to initialize vector element with an expression that:
  c) refers to the same object as the argument ⇒ to temporary `i+1`,
  d) has the same value category as the argument ⇒ *rvalue*.
- *Possible solutions?*

```
  new (data_ + size_) T( param );  // NO! param is always lvalue => breaks d)
```

```
  new (data_ + size_) T( std::move(param) );
                                   // NO! std::move(param) is always rvalue => breaks b)
```

Emplacing-back into vector *(cont.)*

```
new (data_ + size_) T( param );
```

```
new (data_ + size_) T( std::move(param) );
```

- *Problem:*
  - `param` is always *lvalue* (*named entity case*)
  - `std::move(param)` is always *rvalue*.
- Instead, we need an expression that:
  - represents the same object as param,
  - its value category is the same as the value category of the `emplace_back` argument.
- *Quick solution* — `std::forward`.

```
new (data_ + size_) T( std::forward<U>(param) );
```

- *Example:*

```
std::vector<X> v;
v.emplace_back(1);
```

```
converting constructor
```

Pushing- *vs* emplacing-back into vector

- *Benchmark:*
  - Insertion of elements with `push_back` and `emplace_back` into a *vector of strings* (`std::string`), where argument is a *string literal*.
- *Results:*
  - With *GCC/libstdc++*, using `emplace_back` was **2.3× faster**.
    - Link: https://quick-bench.com/q/bVc7e_EjenbAVWqAzSK22reeqYc.
  - With *Clang/libc++*, using `emplace_back` was **3.2× faster**.
    - Link: https://quick-bench.com/q/QtdeRHqkhARu-m9XOsJ4JeyxHr4.
- *Alternative benchmark:*
  - The same with a *vector of integers* (`int`) and *integer literal argument*.
  - *Results* — **same** measured runtime/performance.
- *Analysis:*
  - With `std::string`, *move constructors* are additionally involved with `push_back`.
  - On the contrary, `int` is a *non-class type* and additional initialization is effectively eliminated under optimizations.

Perfect forwarding

- How does `std::forward` function work?
- *Recall* — `param` is a forwarding reference that is bound to some object.

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

- This object is represented by either *lvalue* or *rvalue* expression.

```
Vector<X> v;
int i = 1;
v.emplace_back( i );
v.emplace_back( i+1 );
```

- We want to initialize new vector element with expression that:
  - represents the same object,
  - has the same value category.

Perfect forwarding *(cont.)*

```
template <typename U>
void emplace_back(U&& param) {
  if (size_ == capacity_)
    reserve(capacity_ ? 2 * capacity_ : 1);

  new (data_ + size_) T( ... );

  size_++;
}
```

```
Vector<X> v;

int i = 1;
v.emplace_back( i );    // (1)

v.emplace_back( i+1 );  // (2)
```

- *Recall:*
  - In the *first (lvalue) case*, `U` is deduced as `int&` (and type of param is `int&`).
  - In the *second (rvalue) case*, `U` is deduced as `int` (and the type of param is `int&&`).
- Generally, in case of
  - *lvalue* → `U` is a *(lvalue) reference* type,
  - *rvalue* → `U` is a *non-reference* type.
- ⇒ *Initialization expression* needs to be:
  - *lvalue* expression if `U` is a reference type,
  - *rvalue* expression otherwise.
- *Possible cast-based solution?*

```
new (data + size_) T( static_cast<U&&>(param) );
```

Perfect forwarding — casting

```
template <typename U> void emplace_back(U&& param) {
  if (size_ == capacity_) reserve(capacity_ ? 2 * capacity_ : 1);
  new (data_ + size_) T( static_cast<U&&>(param) );
  size_++;
}
```

- *Lvalue-case example:*

```
Vector<X> v;
int i = 1;
v.emplace_back( i );
```

  - U is deduced as `int&` ⇒ after *substitution* and *reference collapsing*:

```
new (data_ + size_) T( static_cast< int& >(param) );  // cast expression is lvalue
```

  - ⇒ Initialization expression represents variable `i` and its category is *lvalue*.

- *Rvalue-case example:*

```
v.emplace_back( i+1 );
```

  - U is deduced as `int` ⇒ cast *turns* into:

```
new (data_ + size_) T( static_cast< int&& >(param) );  // cast expression is rvalue
```

  - ⇒ Initialization expression represents temporary `i+1` and its category is *rvalue*.

Perfect forwarding — `std::forward`

- Delegation of cast to a separate function:

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

- *Application* to `std::vector<T>::emplace_back`:

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

- *Lvalue-case example:*

```
Vector<X> v;  int i = 1;  v.emplace_back( i );
```

  - ⇒ T in forward is `int&` ⇒ after *substitution* and *collapsing*:

```
int& forward(int& param) { return static_cast<int&>(param); }  // its call is lvalue
```

  - ⇒ forward call expression represents variable `i` and is *lvalue*.

- *Rvalue-case example:*

```
Vector<X> v;  int i = 1;  v.emplace_back( i+1 );
```

  - ⇒ T in forward is `int` ⇒ after *substitution*:

```
int&& forward(int& param) { return static_cast<int&&>(param); }  // its call is rvalue
```

  - ⇒ forward call expression represents temporary `i+1` and is *rvalue*.

Perfect forwarding — `std::forward` *(cont.)*

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

- *Note* — our forward function (template) accepts only *lvalue arguments*.
- This was ok for our `emplace_back`.
  - Its parameter `param` used as an argument of `forward` is always *lvalue* (named-entity case).

```
new (data_ + size_) T( forward<U>(param) );  // param expression is always lvalue
```

- Generally, we need to "cover" cases where `forward`:
  - *function argument* is either *lvalue* or *rvalue*,
  - *template argument* is either *reference* or *non-reference* type.
- ⇒ This makes 4 different cases, which cannot be resolved with a single definition of `forward` function template.
- An additional overload would need to be added for *rvalues*:

```
template <template T>
T&& forward(typename remove_reference<T>::type&& param)  // overload for rvalues
{  return static_cast<T&&>(param);  }
```

Perfect forwarding — `std::forward` *(cont.)*

- Such two overloads are provided by the C++ standard library as a function template `std::forward`.
- ⇒ Explanation of the *original solution*:

```
new (data_ + size_) T( std::forward<U>(param) );
```

- *Relevant notes:*
  - Template argument for `(std::)forward` call must be explicitly specified.
  - Template argument deduction rules wouldn't work here with the desired functionality.
  - ⇒ Implementations of `std::forward` are more "*robust*" (than our `forward`), and enforce explicit template argument provision.
    - *Note* — in application to our problem, there is effectively no difference.
  - Relevant *Stack Overflow* discussions:
    - Why does std::forward have two overloads?
    - The implementation of std::forward

Perfect forwarding *(cont.)*

- Generalization:
  - We have a function (template) that has a *forwarding reference parameter*.
  - This function is called with some *function argument (= expression)*.
  - This argument represents some object and has some value category.
  - Inside the function, there is another expression created that:
    - represents the same object as the function argument,
    - has the same value category as the function argument.
- Such technique is generally called "***perfect forwarding***".
- It is effectively used for passing/"forwarding" of arguments of "outer" function call into some internal another function call (for instance, constructor in case of initialization).
- "*Perfect*" = it preserves all properties of arguments (representation of particular object, its type, and value category).
- *Note* — "*forwarding*" gave rise of term "*forwarding reference*"

`std::forward` *vs* `std::move`

- *Perfect forwarding* is typically implemented with the help of library utility function (template) `std::forward`.
- ⇒ This is the reason of its name.
- `std::forward` and `std::move`:

1) `std::move` — *recall*:
   - It does not "move" anything.
   - Instead, its call creates an expression that represents the same objects as its argument and has category *rvalue*.
2) `std::forward` — *similarly*:
   - It does not "forward" anything.
   - Instead, its call creates an expression that represents the same object as its argument (*forwarding reference*) and has a desired value category (*same as the expression "bound" to that forwarding reference*).

- ⇒ Both functions are technically similar; they generally differ only in the value category they "*produce*".
- ⇒ However, they both have very different use cases.

Emplacing back — multiple arguments

- Final solution:

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

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_++;
}
```

- It works as required, but…
- …only for a single argument.

```
struct X {
  X(int) { }  // converting constructor
};
```

```
Vector<X> v;

v.emplace_back( 1 );  // ok
```

- ⇒ This version of `emplace_back` cannot perfectly-forward multiple arguments to the vector elements initialization.
  - This might be required for constructors with multiple parameters.

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

```
Vector<Y> v;

int i = 1;
v.emplace_back( i, i+1 );  // error
```

Emplacing back — multiple args *(cont.)*

- We would like to "perfectly-forward" — in our case to the initialization expression of the added vector element — not only a *single argument*, but *any number of arguments*.
- *Quick solution* — making emplace_back a **variadic template**:

```
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_++;
}
```

```
Vector<Y> v;
int i = 1;
v.emplace_back( i, i+1 );  // ok
```

- *Effect:*
  - Each argument is (separately) perfectly-forwarded to the initialization expression of the constructed vector element.
    - The first constructor argument represents variable `i` and its category is *lvalue*.
    - The second constructor argument represents temporary `i+1` and its category is *rvalue*.

Emplacing back — multiple args *(cont.)*

- *Resolution:*

```
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_++;
}
```

```
Vector<Y> v;
int i = 1;
v.emplace_back( i, i+1 );
```

- `emplace_back` has 2 arguments ⇒ effectively equivalent with…

```
template <typename U, typename V> void emplace_back(U&& param1, V&& param2) {
  if (size_ == capacity_) reserve(capacity_ ? 2 * capacity_ : 1);
  new (data_ + size_) T( std::forward<U>(param1), std::forward<V>(param2) );
  size_++;
}
```

- …which is then "resolved" (*instantiated*) as:

```
void emplace_back(int& param1, int&& param2) {
  if (size_ == capacity_) reserve(capacity_ ? 2 * capacity_ : 1);
  new (data_ + size_) Y( std::forward<int&>(param1), std::forward<int>(param2) );
  size_++;
}
```

Emplacing back — no argument

```
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_++;
}
```

- This variadic template-based solution even allows to forward **no argument** at all:

```
Vector< std::string > v;
v.push_back();     // error - push_back requires an argument
v.emplace_back();  // ok - inserts empty = default-constructed string object in the vector
```

- emplace_back is here *instantiated* as:

```
void emplace_back() {
  if (size_ == capacity_) reserve(capacity_ ? 2 * capacity_ : 1);
  new (data_ + size_) std::string( /* nothing here */ );
  size_++;
}
```

push_back *vs* emplace_back

- push_back requires — as its input — an object of vector's value type T.
  - Recall, there are two overloads for lvalues and rvalues.

```
// template <typename T> class Vector { ...
void push_back(const T& param) {                                   // overload for lvalues
  if (size_ == capacity_) reserve(capacity_ ? 2 * capacity_ : 1);
  new (data_ + size_) T( param );
  size_++;
}
void push_back(T&& param) {                                        // overload for rvalues
  if (size_ == capacity_) reserve(capacity_ ? 2 * capacity_ : 1);
  new (data_ + size_) T( std::move( param ) );
  size_++;
}
```

- On the contrary, emplace_back can accept *any argument*:

```
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_++;
}
```

- ⇒ These arguments may — of course — be of type T as well.

push_back *vs* emplace_back *(cont.)*

- When argument of emplace_back is of type T and its category is *lvalue*, emplace_back behaves exactly as the *lvalue* overload of push_back.
  - ⇒ We can write this overload of push_back in terms of emplace_back:

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

- Similarly, when argument of emplace_back is of type T and its category is *rvalue*, emplace_back behaves exactly as the *rvalue* overload of push_back.
  - ⇒ We can also write this overload of push_back in terms of emplace_back:

```
void push_back(T&& param) {                                             // overload for rvalues
  emplace_back( std::move(param) );  // need to make rvalue from param
}
```

- This way — for example — is the push_back implemented in *Microsoft STL* (with a bit different internal syntax, but the same effect).
  - Link: https://github.com/microsoft/STL/blob/main/stl/inc/vector#L633.

Perfect forwarding — use cases

- *Perfect forwarding* is one of the building blocks of modern C++.
- It was introduced in C++11 (same as, for example, *content moving semantics*).
- It is commonly used in C++ standard library.
- *Use case I.* — `std::construct_at` (seen in lecture 4):
  - *Recall* — this function has the functionality of *placement new*.
  - ⇒ It explicitly initializes an object of a *give type* in the *location* specified by its first function argument…
  - …while all other arguments are *perfectly forwarded* to the *object initialization expression*.

```
new (ptr) T(arg1, arg2, arg3);                    // option #1 - placement new
std::construct_at<T>(ptr, arg1, arg2, arg3);  // option #2 - equivalent
```

- *Possible implementation:*

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

Perfect forwarding — use cases *(cont.)*

- *Use case II.* — constructor of `std::optional` (seen in lecture 4):
  - *Recall* — *optional class* can optionally own/store an object of its *value type* (template argument) into its *included storage*.
  - It has a constructor that immediately initializes the owned object and perfectly forwards any arguments to its constructor.
- *Possible implementation* of such a constructor:

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

template <typename Ts...>
optional(std::in_place_t, Ts&&... args) : exists_(true) {
  new (buffer_) T(std::forward<Ts>(args)...)
}
```

- *Alternative implementation* with `std::construct_at` (⇒ *double perfect forwarding*)

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

template <typename Ts...>
optional(std::in_place_t, Ts&&... args) : exists_(true) {
  std::construct_at<T>(buffer_, std::forward<Ts>(args)...);
}
```

- What about that (unnamed) parameter of type `std::in_place_t`?

Perfect forwarding — use cases *(cont.)*

- Why isn't the constructor defined simply as follows?

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

template <typename Ts...>
optional(Ts&&... args) : exists_(true) {
  new (buffer_) T(std::forward<Ts>(args)...)
}
```

- Consider the following two options when initializing an optional object:
  - *First*, we want to create an *"empty" optional owner object*, which does not own any object of its *value type*.
    - For this purpose, there is a *default constructor*:

```
std::optional< std::string > o1;  // default constructor => no owned object
```

  - *Second*, we want to initialize an optional owner that owns an empty = *default-constructed string object*.
    - ⇒ We would like to invoke the "*forwarding-constructor*" and ***forward nothing***.
    - With the above definition, this would require **no constructor argument as well**.
- ⇒ We need some mechanism to tell the compiler that it should call the forwarding constructor in this case (*discussion:* [link]).
- Mechanism used by `std::optional` is called *tag dispatching*.

Function call dispatching

- *Situation* — *function call* which may correspond to *multiple versions (variants/definitions)* of that function.
- ***Dispatching** of such function call* = resolution of which of the versions will be called.
- C++ has several different dispatching mechanisms suitable for different situations.
- *First* dispatching option — *function overloading*.
  - The dispatch is resolved according to the *function argument expressions* (their types and value categories).
  - *Example* — *type-based* overload-dispatching:

```
f(int);      // overload #1
f(double);   // overload #2
```

```
f(  1  );  // overload #1 called
f( 1.0 );  // overload #2 called
```

  - *Another example* — value *category-based* overload-dispatching:

```
struct X {
  X();            // default constructor
  X(const X&);    // copy constructor
  X(X&&);         // move constructor
};
```

```
X x1;

X x2(      x1       );  // copy ctor called
X x3( std::move(x1) );  // move ctor called
```

Function call dispatching *(cont.)*

- *Second* dispatching option — *dynamic polymorphism*.
  - The dispatch is resolved according to the *actual (dynamic) type of the pointed-to/referenced object*.
  - *Example*:

```
struct Base {
  virtual void f() = 0;
  virtual ~Base() = default;
};

struct Derived1 : Base {
  virtual void f() { std::cout << "1"; }
};

struct Derived2 : Base {
  virtual void f() { std::cout << "2"; }
};

void f(Base& obj) {
  obj.f();  // dispatch required
}
```

```
std::unique_ptr<Base> p_obj;

int i;
std::cin >> i;

p_obj = (i == 1) ?
  new Derived1 : new Derived2;

f(*p_obj);

// => calls either
//    Derived1::f()
// or
//    Derived2::f()
// based on the type of created object
```

- Dispatch resolution is based on *virtual functions.*
  - It is sometimes referred to as "*virtual dispatch*".

Function call dispatching *(cont.)*

- *Third* dispatching option — *static polymorphism*.
  - The dispatch is resolved according to the *actual type of the object* that is specified by a template parameter.
  - *Example*:

```
template <typename T>
void f(T& obj) {
  obj.f();  // dispatch required
}
struct A {  void f() { std::cout << "A"; }  };
struct B {  void f() { std::cout << "B"; }  };
```

```
A obj;
f(obj);  // calls A::f()
```

- Dispatch may be resolved
  1) either *at compile time* — then, it is generally called "*static dispatch*",
  2) or *at runtime* — then, it is generally called "*dynamic dispatch*".
- *Examples:*
  - *Overload-* and *static polymorphism*-based dispatches are *static*.
  - *Dynamic polymorphism (virtual)*-based dispatch is *dynamic*.

Function call dispatching *(cont.)*

- When class object is *initialized*, constructor is dispatched according to the *initialization expression* by the *overloading mechanism*:

```
struct X {
  X();
  X(const X&);
  X(X&&);
};
```

```
X x1;                  // no argument => default constructor

X x2(x1);              // lvalue argument of type X => copy constructor
X x3(std::move(x1));   // rvalue argument of type X => move constructor
```

- *Problem* with the following version of the forwarding constructor of the `optional` class is…

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

optional();                                        // default constructor

template <typename Ts...> optional(Ts&&... args);  // forwarding constructor (attempt)
```

- …that the overloading rules cannot distinguish between the described two initialization cases:
  1) initialization of *empty* `optional` object,
  2) initialization with *no-argument forwarding*.
- Both syntactically correspond with initialization with no argument:

```
std::optional< std::string > o1;  // both default and forwarding constructor match
```

Overloading

```
optional();                                      // default constructor
template <typename Ts...> optional(Ts&&... args);  // forwarding constructor (attempt)
```

```
std::optional< std::string > o1;  // both default and forwarding constructor match
```

- In this case, *both constructor* match the initialization form (*no/empty initialization expression*).
  - They both can be called ⇒ are so-called *viable "candidates"* for overload resolution.
- Which of them will be finally called?
- *Generally*, when they are multiple viable overloading candidates, the one with the *highest priority* is selected.
  - These priorities are specified by (relatively complex rules of) C++ standard.
- In our case, the *default constructor* will be selected according to these rules (*non-templates* typically have priority over *templates*).
- *Note:*
  - Multiple candidates with equal highest priority results in compilation error ("*ambiguous call of…*").

Tag dispatching

```
optional();                                     // default constructor
template <typename Ts...> optional(Ts&&... args);  // forwarding constructor (attempt)
```

- *Overloading rules* cannot distinguish between initialization with these two constructors in case of *no-argument forwarding*.
- ⇒ We need some other dispatching mechanism how to select forwarding constructor.
- One such mechanism is so-called "***tag dispatching***".
  - It works such that we introduce into a function a parameter (typically unnamed) of a *special type* created only for tag dispatching purposes.
  - *Example:*

```
void f() { std::cout << "1"; }

struct tag_t { };  // special new tagging-purpose type
template <typename... Ts> void f(tag_t, Ts&&...) { std::cout << "2"; }
int main() {
  f();            // prints out "1"
  f( tag_t{} );   // prints out "2"
```

- *Note* — tag dispatching is an "*explicit overloading mechanism*".

Tag dispatching *(cont.)*

- *Tag dispatching* is used for selection of *forwarding constructor* of `std::optional`.
- The special *selection "tag" type* is `std::in_place_t`:

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

template <typename Ts...>
optional(std::in_place_t, Ts&&... args);
```

- Forwarding constructor is then effectively selected by providing the first argument of this type during initialization:

```
std::optional< std::string > o1;                      // default constructor called
std::optional< std::string > o2( std::in_place_t{} );  // forwarding constructor called
```

- *Resolution:*
  - *First initialization* creates an empty optional object.
  - *Second one* creates an optional object that owns an empty string object.
- *Note:*
  - Tag argument needs to be an expression/object of type `std::in_place_t`.
  - Standard library defines such an object for us:

```
std::optional< std::string > o2( std::in_place );
```

Perfect forwarding *(cont.)*

- *Use case III.* — smart pointer "*makers*".
  - Smart pointers do not have forwarding constructors.
  - Instead, we can initialize them by creating owned objects explicitly:

```
std::unique_ptr<X> upy = new X(1);
```

- *Drawbacks:*
  - Type needs to be written twice in the source code (*duplication*).
  - Its "*ugly*" — using new explicitly is considered bad practice in modern C++ when programming at a high level of abstraction.
- *Alternative solution…*

```
std::unique_ptr<X> upy = std::make_unique<X>(1);  // eliminates explicit new
```

- …or, even better:

```
auto upy = std::make_unique<X>(1);                // eliminates both drawbacks
```

- The same approach — `std::shared_ptr` + `std::make_shared`.
- *Caveat:*
  - Smart pointers support so-called *custom deleters*, which cannot be supplied via `make_` functions.

Perfect forwarding *(cont.)*

- Function (template) `std::make_unique`:
  - It creates a unique pointer that owns a constructed object of a desired type…
  - …and perfectly forwards all function arguments to the object initialization expression.
- ⟹ *Possible implementation:*

```
template <typename T, typename... Ts>
std::unique_ptr<T> make_unique(Ts&&... params) {
  return std::unique_ptr<T>( new T( std::forward<Ts>(params)... ) );
}
```

- Does `make_unique` have any *runtime overhead*?

```
std::unique_ptr<X> upy1 = new X(1); // converting constructor of std::unique_ptr called only
auto upy2 = std::make_unique<X>(1); // which constructors called here?
```

  - *Since C++17*, it is *guaranteed* to be equivalent ⟹ no overhead.
  - Until C++17, it is *very likely* equivalent ⟹ (very) likely no overhead.
  - *Reason* = *copy elision* optimization technique.

Perfect forwarding *(cont.)*

- *Use case IV. —emplacing functions* of library containers.
  - We have seen `std::vector::emplace_back`.
  - Other containers (and container adapters) also support *emplacing semantics* for elements insertion operations.
  - They differ from "traditional" insertion functions such that they construct/initialize new container elements while perfectly forwarding arguments to its initialization expression.
  - *Examples:*

```
std::set<std::string> m;
m.insert("string");   // converting + move
m.emplace("string");  // converting only
```

```
std::list<std::string> l;
l.push_front("string");     // converting + move
l.emplace_front("string");  // converting only
```

```
std::stack<std::string> s;
s.push("string");     // converting + move
s.emplace("string");  // converting only
```