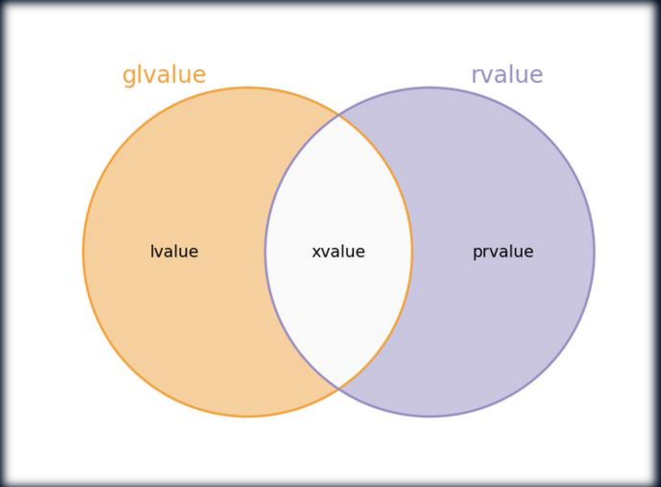
Effective C++ Programming

NIE-EPC (v. 2021):

VALUE CATEGORIES, TYPES OF REFERENCES, REFERENCE BINDING, OVERLOADING RULES

Value categories

- In C++, each *expression* is characterized by two properties:
  - *type*,
  - *value category*.
- The total number of value categories is 5, but for most purposes, only 2 of them matter:
  - *lvalues*,
  - *rvalues*.



- *Important rule* — each expression is (has category) either *lvalue* or *rvalue*.
- Precise definitions are complicated, but we will try to simplify them.
- *Cppreference* link:
  - https://en.cppreference.com/w/cpp/language/value_category.

Value categories — *lvalues*

- *Lvalues* — addressable entities.
  - Their addresses can be obtained by & operator.
- *Major cases:*
  - *Entities with name* — every expression that itself is a named entity is *lvalue* (*variables*, *references*, *function parameters*, *functions*,…).
  - *Array elements* — we can obtain their addresses.
  - *Dereferenced pointers* —addresses = pointer values.
  - *Function-call expressions* whose return type is *lvalue reference* (see later).
    - *Example* — `operator[]` of `std::vector` [link].

```
std::vector<int> v = { 1, 2, 3 };
std::cout << v[1];  // expression "v[1]" is lvalue
```

    - It returns *lvalue reference* to its element, which is addressable.
  - *String literals* — exception; other literals are *rvalues*.
  - *Casting to lvalue reference type* (see later).

Value categories — *rvalues*

- *Rvalues:*
  - Expressions that are not lvalues = *non-addressable* expressions.
- *Major cases:*
  - *Non-string literals* — `-1`, `false`, `nullptr`,…
  - *Temporary objects* — results of operations,…

```
int i = 1;
int a[3] = { i /* lvalue */, 1 /* rvalue */, i + 1 /* rvalue */ };
```

  - *Function-call expressions whose return type is non-reference.*

```
std::string s("hello world");
std::string s2 = s.substr(0, 5);  // red-emphasized expression is rvalue
```

  - *Function-call expressions whose return type is rvalue reference.*
    - *Example* — `std::move`:

```
std::string s("hello world");
std::stirng s_copy = s;            // purple-emphasized expression is lvalue
std::string s_move = std::move(s); // red-emphasized expression is rvalue
```

  - *Cast expression to non-reference type or to rvalue reference* (see later).

Value categories — *function overloading*

- Why are value categories important?
  - Because they, in practice, frequently decide which *overloaded variant of some function* will be called according to the value category of function argument.
- *Examples:*
  - Two overloaded variants of `std::vector::push_back`:
    - One is called for *lvalue arguments*, the other one *for rvalue arguments*.
  - *Overloaded constructors* (and *assignment operators*) for initialization expressions of the same type:
    - *copy constructor* is used for *lvalue "arguments"* — object initialization expressions,
    - *move constructor* (if exists) is used for *rvalue arguments*.
- How this overloading works?
  - Typically, it is based on passing objects by reference, where overloads have different reference types of parameters.

References — *lvalue references*

- Types of references in C++ (since C++11):

1) *Lvalue references* — denoted by single & source code character.

```
template <typename T>
void f(
  int & lri,           // lvalue reference to int
  const int & clri,    // constant lvalue reference to int
  T & lrt              // lvalue reference to T (whatever it is resolved to)
  const T & clrt       // constant lvalue reference to T
) { ... }
```

- *Notes:*
  - "*Constant reference*" should be better worded as "*reference-to-const*".
  - Each reference is *constant* per se since it refers to the same object through its entire existence.
  - *Reference-to-const* — denoted by const modifier — indicates that referred-to object cannot be modified **through this reference**.
    - ⇒ It does not imply that the object itself is *constant* (*immutable*).
  - *For sake of simplicity*, we call "*references-to-const*" simply "*constant references*".

References — *rvalue and forwarding refs*

2) *Rvalue references* — denoted by double && source code characters in case these do not make forwarding references (see later).

```
template <typename T>
void f(
  int && rri,          // rvalue reference to int
  T && frt             // rvalue forwarding reference to T (whatever T resolves to)
) { ... }
```

- *Note* — constant rvalue references are almost never useful.
  - *Relevant discussion:* https://stackoverflow.com/q/4938875/580083.

3) *Forwarding references* — denoted by auto&&, or T&& where T is a function template parameter.

```
template <typename T>
void f(
  T && frt             // forwarding reference to T (whatever T resolves to)
) { ... }
```

- *Note* — forwarding reference type must be exactly *function template parameter + double ampersand &&*.
  - Any other form would make `frt` not a forwarding reference.

Reference binding

- A reference always refer to some entity, such as an object or a function.
- *Without the loss of generality*, we will consider mostly *references-to-objects*.
- *Standard wording* — reference that refers to some object is said to be "bound to" this object.

```
int i = 1;
int & ri = i;  // ri is bound to i
```

- *References-related rules:*
  - References cannot be "*rebound*" to another object.
  - When the lifetime of the bound object ends, the reference becomes *invalid* — so-called "*dangling*" – and it may not be used any more.
- Common *dangling-reference bug*:

```
int & f() {
  int i = 1;  // i does not exist outside of f...
  return i;   // ...where reference to i is passed
}
```

Reference binding *(cont.)*

- To what is reference bound is given by its *initialization expression*.

```
int & f() {
  int i = 1;
  return i;   // reference returned from f is bound to i
}
```

```
void g(int & par_ri) { ... }
```

```
int i = 1;
int & ri = i;  // ri is bound to i
g(i);          // par_ri is bound to i in this particular function call
```

```
std::vector<int> v = { 1, 2, 3 };
int & ri = v[1];  // ri is bound to second vector element
g(v[1]);          // par_ri is bound to second vector element in this function call
```

- References can sometimes prolong lifetime of temporaries:

```
int i = 1;
const int & cri = i + 1;
// cri is not dangling here, the temporary of type int with value 2 still exists
```

- Rules for temporary lifetime extension due to reference bounding are relatively complex ⇒ better is not to do this.

```
const int & f(const int & cri) {
  return cri;
}
```

```
const int & ref = f(1);
// lifetime of literal 1 is not extended
// => ref is dangling here
```

Reference binding and value categories

- Generally, references cannot be bound to objects of all categories.
- Basic rules:
  1) *Lvalue references* can be bound only to *lvalues*.

```
void f(int &) { }
```

```
int i;
std::vector<int> v = {1, 2, 3};

f(i);       // ok    - "i"     is lvalue (named entity)
f(v[1]);    // ok    - "v[1]"  is lvalue (operator[] returns lvalue reference)
f(1);       // error - "1"     is rvalue (non-string literal)
f(i + 1);   // error - "i + 1" is rvalue (temporary object)
```

  2) *Rvalue references* can be bound only to *rvalues*.

```
void g(int &&) { }
```

```
int i;
std::vector<int> v = {1, 2, 3};

g(i);       // error - "i"     is lvalue
g(v[1]);    // error - "v[1]"  is lvalue
g(1);       // ok    - "1"     is rvalue
g(i + 1);   // ok    - "i + 1" is rvalue
```

- *Note* — these rules justify names of “*lvalue*” and “*rvalue references*”.

Reference binding and value categs. *(cont.)*

- *Special case*:
  3) *Constant lvalue references* can be bound *both lvalues and rvalues*.

```
void h(const int &) { }
```

```
int i;
std::vector<int> v = {1, 2, 3};

h(i);       // ok - "i"     is lvalue
h(v[1]);    // ok - "v[1]"  is lvalue
h(1);       // ok - "1"     is rvalue
h(i + 1);   // ok - "i + 1" is rvalue
```

- This rule allows writing a single function that can work with external objects regardless of their value category.
- For example, *equality-comparison* `operator==` typically does have both parameters constant lvalue references [*sample link*].

```
std::optional<int> lhs(1);
if (lhs == std::optional<int>(2)) ...  // compares lvalue with rvalue
```

  - It does not make any sense to distinguish lvalues and rvalues when comparing objects (their content).
  - When does it make sense? *See later…*

Reference binding and value categs. *(cont.)*

- Up to now, in *ad 1), 2), and 3)*, we worked with references related to particular object type:

```
void f(int &) { }
```

```
void g(int &&) { }
```

```
void h(const int &) { }
```

- What if we want to write a (*generic*) function that pass arguments by reference regardless of their type?
- *Lvalue reference — generalized case:*

```
template <typename T>
void f_gen(T &) { }
```

- Works as expected — only *lvalues* are accepted, and they may be of any type.

```
int i;
double d;

f_gen(i);    // ok - "i"      is lvalue (of type int)
f_gen(d);    // ok - "d"      is lvalue (of type double)
f_gen(1);    // error - "1"   is rvalue (of type int)
f_gen(1.0);  // error - "1.0" is rvalue (of type double)
```

Reference binding and value categs. *(cont.)*

- *Lvalue reference — generalized case (cont.)…*
- Exemplary library function template — `std::swap` for swapping content of two objects, both of the same type.

std::swap

Defined in header `<algorithm>`
Defined in header `<utility>`
Defined in header `<string_view>`

```
template< class T >
void swap( T& a, T& b );
```

- ⇒ `std::swap` can work only with lvalue arguments:

```
std::string s("hello");
std::swap(s, string("hi"));  // error
```

- *Rationale:*
  - Mostly, it does not make sense to swap content with *rvalue*.
  - For instance, rvalues are typically *temporaries* whose lifetime does not exceed the expression they emerge within.
  - ⇒ Assignment makes more sense then:

```
s = string("hi");  // makes more sense
```

Reference binding and value categs. *(cont.)*

- *Constant lvalue reference — generalized case:*

```
template <typename T>
void h_gen(const T &) { }
```

- Works as expected — both *lvalues* and *rvalues* are accepted, and they may be of any type.

```
int i;
double d;

h_gen(i);    // ok - "i"   is lvalue (of type int)
h_gen(d);    // ok - "d"   is lvalue (of type double)
h_gen(1);    // ok - "1"   is rvalue (of type int)
h_gen(1.0);  // ok - "1.0" is rvalue (of type double)
```

- *Rvalue? reference — generalized case:*

```
template <typename T>
void g_gen(T &&) { }
```

- Does not works as expected — both *lvalues* and *rvalues* are accepted (and they may be of any type).

```
int i; double d;

g_gen(i); g_gen(d); g_gen(1); g_gen(1.0);  // all ok
```

Template argument deduction

- When function template is called and:
  - some *template argument* is not *explicitly provided*,
  - and *template parameter* appears in some function parameter type,
- …then this template argument is deduced from the type of that function argument (*simplified wording*).
- *Example — lvalue reference case*:

```
template <typename T> void f_gen(T & param) { }
```

```
int i; double d;

f_gen(i);  // in this call, T is deduced as int => type of param is int&
f_gen(d);  // in this call, T is deduced as double => type of param is double&
```

- This deduction is "*intuitive*", as for *constant lvalue reference case*:

```
template <typename T> void h_gen(const T &) { }
```

```
int i; double d;

h_gen(i);    // in this call, T is deduced as int => type of param is const int&
h_gen(d);    // in this call, T is deduced as double => type of param is const double&
h_gen(1);    // in this call, T is deduced as int => type of param is const int&
h_gen(1.0);  // in this call, T is deduced as double => type of param is const double&
```

Template argument deduction *(cont.)*

- However, the *seemingly rvalue reference case* is different:

```
template <typename T> void g_gen(T && param) { }
```

- In case of *rvalue* arguments, it "works" as expected:

```
int i; double d;

g_gen(1);    // in this call, T is deduced as int => type of param is int&&
g_gen(1.0);  // in this call, T is deduced as double => type of param is double&&
```

- ⇒ Parameters indeed are rvalue references.
- However, in case of *lvalue arguments*, something weird happens:

```
int i; double d;

g_gen(i);  // in this call, T is deduced as int& => type of param is int& (???)
g_gen(d);  // in this call, T is deduced as double& => type of param double& (???)
```

- ⇒ Parameters are lvalue references.
- ⇒ They can bind lvalue arguments.
- *Short explanation* — T&& here is not an *rvalue reference* but it is a *forwarding reference*.
  - Forwarding references have special template argument deduction rules.
  - Why?

Reference collapsing

```
template <typename T> void g_gen(T && param) { }
```

```
int i; double d;

g_gen(i);  // in this call, T is deduced as int& => type of param is int& && = int&
g_gen(d);  // in this call, T is deduced as double& => type of param double& && = double&
```

- *First puzzle:*
  - When
    - `T` is *deduced* as `int&`,
    - and is "*substituted*" into `T&&`,
  - how is possible that the resulting type of param is `int&`?
  - Shouldn't it be `int& &&` instead?
- *Explanation:*
  - Technically, after *substitution*, the type of param is `int & &&`.
  - However, then this `int & &&` "*collapses*" into `int &`…
  - …according to so-called *reference collapsing rules*:
    - `& &&`, `&& &`, and `& &` *collapse* to `&`.
    - Only `&& &&` *collapses* to `&&`.
  - *See, for example:* https://en.cppreference.com/w/cpp/language/reference.

Forwarding references

- *Need:*
  - passing by reference argument of *any type* and *any value category*,
  - plus *finding out the value category* of the passed argument inside the function.
- *Why?*
  - Mostly used for so-called "*perfect forwarding*" — will be explained later.
- Only *forwarding reference* can do that:
  - If passed argument is an lvalue of type X, the corresponding template argument is deduced as X&,
  - otherwise (rvalue argument of type X), it is deduced as X.
  - ⇒ If the *template parameter is reference*, passed argument was *lvalue*, and *vice versa*.
  - *Counter-example* — generic constant lvalue references:
    - For both *lvalue* and *rvalue arguments*, the template argument is deduced equally.
    - ⇒ There is no information about value category of passed argument available in the function.

```
// template <typename T> void h_gen(const T &) { }

int i;

h_gen(i);  // lvalue argument => T is deduced as int, type of param is const int &
h_gen(1);  // rvalue argument => T is deduced as int, type of param is const int &
```

Forwarding references — *example*

- *Example* —template argument deduction rules for forwarding references allows us to write a function that tell whether the argument expression is *lvalue* or *rvalue*:

```
template <typename T>
bool is_lvalue(T&& param) {  // forwarding reference parameter
  return ???  // return true if T is reference, false if it is non-reference type
}
```

- In case of *lvalue argument*, T is *lvalue reference* to argument's types, otherwise (*rvalue argument*), T is *non-reference*.
- How to check this?
- *Possible solution* = *template metafunction* `std::is_reference`.
- *Recall — template metafunctions / C++ library type traits*:
  - take some type or number as template argument,
  - produce some type or number as its "result" (member type/variable):
  - We have seen `std::is_trivially_copyable<T>` that maps type to a Boolean value (static member value) which is *true* if T is *trivially-copyable*.

Forwarding references — *example (cont.)*

- Library metafunction (type trait) `std::is_reference<T>`:
  - Maps type to a Boolean value which is
    - *true* if the input type is a *reference type*,
    - *false* otherwise.
  - "Output" Boolean value has a form of a static member constant `value`.
  - *Example:*

```
std::cout << std::is_reference< int   >::value;  // prints out "0"
std::cout << std::is_reference< int&  >::value;  // prints out "1"
std::cout << std::is_reference< int&& >::value;  // prints out "1"
```

- Application to our problem is straightforward:

```
template <typename T>
bool is_lvalue(T&& param) {
  return std::is_reference<T>::value;
}
```

- *Demo:*

```
int i;
std::cout << is_lvalue( i );                // prints out "1" (named entity - variable)
std::cout << is_lvalue( 1 );                // prints out "0" (non-string literal)
std::cout << is_lvalue( i+1 );              // prints out "0" (temporary)
std::cout << is_lvalue( "string literal" ); // prints out "1" (string literal)
```

Value categories + copying/moving content

- Assume that we want to create a function `f` that:
  - takes some object of type `X` as its argument,
  - and, inside, either *copy* or *move* content from it.
- How to design such a function?
- *Recall:*
  - *Lvalues* are *addressable* and, typically, their lifetime is longer than the lifetime of the expression where they emerge:

```
X x;
f( x );  // expression "x" has category lvalue (variable - named entity)
```

    - Here, the object referred to by lvalue expression “x” existed before the function-call expression and will exist after it as well.
  - *Rvalues* are typically short-lived objects (such as *temporaries*) whose lifetime is limited to the expression where they emerge:

```
f( X{} );  // expression "X{}" has category rvalue (temporary)
```

    - Here, the object referred to by rvalue expression “X{}” exists only within the function-call expression.

Value categories + copying/moving *(cont.)*

- *Consequences:*
  - *Lvalues* are natural candidates for *copying content* from.
  - *Rvalues* are natural candidates for *moving content* from.
    - Why to copy content from an object that will be then soon destructed?
- *Idiomatic solution:*
  - Two function overloads:
    - *Frist for lvalue arguments* that will *copy content* from them inside.
    - *Second for rvalue arguments* that will *move content* from them inside.

```
void f(X&  param) { /* copy from argument bound to param */ }  // accepts lvalue arguments
void f(X&& param) { /* move from argument bound to param */ }  // accepts rvalue argumetns
```

- *More idiomatic solution:*
  - The content of the *copied-from object* is usually preserved.
  - ⇒ *Constant lvalue reference* indicates that explicitly:

```
void f(const X& param)  { ... }
void f(X&& param)       { ... }
```

Value categories + copying/moving *(cont.)*

- *Problem?*

```
void f(const X& param)  { ... }  // (1) accepts lvalue and rvalue arguments
void f(X&& param)       { ... }  // (2) accepts rvalue argumetns
```

  - For *lvalue arguments*, only overload (1) can be called.
  - For *rvalue arguments*, both overloads are viable:
    - *Rvalue* can be bound both to *constant lvalue reference* as well as to *rvalue reference*.
  - Fortunately, C++ overloading rules will prefer the second overload (2).
    - ⇒ There is no overloading *ambiguity*:

```
X x;

f( x );    // will call overload (1)
f( X{} );  // will call overload (2)
```

- *Example* — `std::vector::push_back` [link]:
  - Two overloaded variants:

1) One for *lvalues* — *copies content* from arguments.
2) Another one for *rvalues* — *moves content* from arguments.

  - *Note* — if *move operation* is not available, ad 2) falls back to copying content (*see later*).

Value categories + copying/moving *(cont.)*

- Another example — *copy/move constructors/assignment operators*.
  - *Copy constructor* copies content (and preserve content of the source object) and content is typically copied from lvalues.
    - $\Rightarrow$ Parameter of copy constructor is usually *constant lvalue reference*.
  - *Move constructor* moves content and content is typically moved from rvalues.
    - $\Rightarrow$ Parameter of move constructor is *rvalue reference*.
  - The same holds for assignment operators.

```
struct X {
  X(cosnt X&);                // copy constructor
  X(X&&);                     // move constructor

  X& operator=(const X&);     // copy assignment operator
  X& operator=(X&&);          // move assignment operators
```

- *Note* — this explains forms of parameter types of *copy/move constructors/assignment operators*.

Moving content from lvalues

- *Recall:*
  - *Lvalues* are “natural candidates” for *copying* content from.
  - *Rvalues* are “natural candidates” for *moving* content from.
- However, this is not always the case.
- Sometimes, content needs to be *moved* from *lvalues*!
- *Example* — vector’s reallocation:

```
for (size_t i = 0; i < size_; i++)
  new (data + i) T( ??? );  // want to move content from original elemetns
```

- Expression `*(data_+i)` itself is *lvalue* (dereferenced pointer case).
  - ⇒ Move constructor cannot be called (rvalue reference parameter).
- When we want to prefer content moving (from the original elements), we need to:
  - make an expression that refers to the same object as the original (*lvalue*) expression,
  - but its category will be *rvalue* instead of *lvalue*.

Moving content from lvalues *(cont.)*

- *First option* — *casting* to rvalue reference:
  - *Cast-to-rvalue-reference expression* has category *rvalue*:

```
int i;
std::cout << is_lvalue( i );                              // prints out "1"
std::cout << is_lvalue( static_cast< int&& >( i ) );  // prints out "0"
```

- Application to vector's reallocation problem:

```
for (size_t i = 0; i < size_; i++)
  new (data + i) T( static_cast< T&& >( *(data_ + i) ) );
```

- *Another option* — function call that returns rvalue reference.
  - This is the case of `std::move` function:

```
int i;
std::cout << is_lvalue( std::move( i ) );  // prints out "0"
```

```
for (size_t i = 0; i < size_; i++)
  new (data + i) T( std::move( *(data_ + i) ) );
```

- The second solution is preferred in practice:
  - It is effectively equivalent, but it is more "*descriptive*" and *idiomatic*.
- *Note* — in both cases, expressions refer to the very same objects, but their category is *rvalue*.

Moving content from lvalues *(cont.)*

```
for (size_t i = 0; i < size_; i++)
  new (data + i) T( std::move( *(data_ + i) ) );
```

- *Universality of this solution* — it works even for types where there is no *content-moving* operation:
- *Trivially-copyable types:*
  - There is no difference between initialization from *lvalue* or *rvalue* expressions:

```
int i = 1;

int* pi1 = new int(           i  );
int* pi2 = new int( std::move(i) );  // effectively the very same as above
```

- *Non-trivially-copyable types with no move constructor:*
  - Copy constructor *parameter* = *constant lvalue reference* ⇒ can be bound to both *lvalue* and *rvalue arguments*.

```
struct X {
  X();
  X(const X&);
  // no move constructor available
};
```

```
X x;

X* px1 = new X(           x  );  // (1)
X* px2 = new X( std::move(x) );  // (2)
// (1) and (2) are effectively equivalent
```

`std::move`

- How to write an `std::move`-like function?
- It needs to take any argument (of any type and any value category) and return *rvalue reference* to it:

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

- This will work for *rvalue* arguments, but not for *lvalue* ones:
  - For lvalue argument of type X, T is deduced to X&.
  - ⇒ *Substitution* to T&& yields X& &&.
  - ⇒ Due to *reference collapsing* rules, the return type becomes X&.
  - ⇒ Return type is *lvalue reference*.
  - *Recall* — function call expression where return type is lvalue reference is *lvalue*.
- *Solution* — in case where T is reference, we need to “remove” it:

```
template <typename T>
typename std::remove_reference<T>::type && move(T&& param) {
  return static_cast< typename std::remove_reference<T>::type && > param;
}
```

std::remove_reference

```
template <typename T>
typename std::remove_reference<T>::type && move(T&& param) {
  return static_cast< typename std::remove_reference<T>::type && > param;
}
```

- `std::remove_reference` is a *type trait* that *“removes” reference from a type*.
- Up to now, we have met two *type traits / template metafunctions*:
  - `std::is_trivially_copyable`:
    - Maps type to a Boolean value, which is *true* if the type is *trivially-copyable*.
  - `std::is_reference`:
    - Maps type to a Boolean value, which is *true* if the type is *reference*.
- `std::remove_reference`:
  - Maps type to a type, which is T if the input type is a *reference-to-T*.
  - *Otherwise*, behaves as *identity* (maps type to itself).
  - The “output” type has a form of a member type called type.

```
std::remove_reference< int&     >::type i = 1;     // type of i is int
std::remove_reference< double&& >::type d = 1.0;   // type of d is double
std::remove_reference< bool     >::type b = true;  // type of b is bool
```

Keyword/specifier typename

```
template <typename T>
typename std::remove_reference<T>::type && move(T&& param) {
  return static_cast< typename std::remove_reference<T>::type && > param;
}
```

- Why is there that `typename` specifier?
- If we write `Y::identifier`, where `Y` is a *class name*, `identifier` may refer to:
  - either *static member variable*,
  - or *member type*.

```
struct Y {
  static const bool value = true;  // non-static member variable
  using type = bool;               // member type ( alternative: typedef bool type; )
};
Y::type b = Y::value;  // Y::type refers to member type, Y::value refers to member variable
```

- *Problem* — if we write `Z<A>::identifier`, where `Z` is a *class template name*, then whether `identifier` refers to
  - either *static member variable*
  - or *member type*,
- may, *generally*, depend on the provided template argument `A`.
- *Possible cause* — *template specialization* (see further lectures).

Keyword/specifier typename *(cont.)*

- In other words:
  1) For some template argument A1, `Z<A1>::identifier` may refer to a *static member variable*,
  2) while for some other template argument A2, `Z<A2>::identifier` may refer to a *member type*.
- If the template argument is "known", a compiler "sees" which case — ad 1) or ad 2) — applies:

```
Z<bool>::identifier b;  // can be resolved, ok if Z<bool>::identifier refers to a type
```

- But the same does not hold if the template argument is "unknown":

```
template <typename T> void f() {
  Z<T>::identifier b;  // Is Z<T>::ident type or a static member variable?
}                      // This cannot be resolved when this code is analyzed.
```

- A compiler needs to distinguish between ad 1) and ad 2) to parse the source code, even if it does not see what T is.
- *Solution:*
  - By default, it assumes that case 1) applies (*static member variable*).
  - Case 2) (*member type*) needs to be explicitly indicated by the typename specifier.

std::move — *example*

```
template <typename T>
typename std::remove_reference<T>::type && move(T&& param) {
  return static_cast< typename std::remove_reference<T>::type && > param;
}
```

- *Illustrative example:*

```
void f(const std::vector<int>&); // lvalue overload
void f(std::vector<int>&&);      // rvalue overload
```

```
std::vector<int> v = { 1, 2, 3 };
f( std::move( v ) );
```

- In `f(std::move(v))` call:
  - Argument `v` of `std::move` is expression of type `std::vector<int>` and category *lvalue*.
  - Parameter `param` of `std::move` is *forwarding reference*.
  - ⇒ `T` is deduced as `std::vector<int>&`.
  - Type of `param` is — due to *ref. collapsing* — `std::vector<int>&` as well.
  - Expression `typename std::remove_reference<T>::type` removes reference from `std::vector<int>&` ⇒ it results in `std::vector<int>`.
  - ⇒ `std::move` is *instantiated* effectively as:

```
std::vector<int>&& move( std::vector<int>& param ) {
  return static_cast< std::vector<int>&& > param;
}
```

Pushing-back into vector

- How to implement `push_back` for our `Vector` class?
- *Relevant questions:*
  - Does push_back insert object "into" the vector (at its "end")? **Yes**.
  - Does push_back insert object passed as its argument into the vector? **No!**
- *Analysis:*
  - The object passed as an argument exists outside of the vector itself.
  - *Recall* — there is no way how to "get" an object from one storage (*external*) to another (*vector's*).
- ⇒ push_back needs to:
  - construct a new object (*element*) in its storage, and
  - either *copy* or *move* content from the argument into this new element.
- *Solution* — two overloaded push_back variants:
  1) one for *lvalue* arguments,
  2) another one for *rvalue* arguments.

Pushing-back into vector *(cont.)*

- Ad 1) Initializes new element with the *reference-to-argument*:

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

- Initialization expression param is *lvalue* (named entity).
  - ⇒ *Copy constructor* will be called for *non-trivially-copyable* types.
- Ad 2) The same approach?

```
  void push_back(T&& param) {  // overload for rvalues
    if (size_ == capacity_) reserve(capacity_ ? 2 * capacity_ : 1);
    new (data_ + size_) T( param );
    size_++;
  }
```

- **Wrong!**

Pushing-back into vector *(cont.)*

```
void push_back(T&& param) {  // overload for rvalues
  if (size_ == capacity_) reserve(capacity_ ? 2 * capacity_ : 1);
  new (data_ + size_) T( param );
  size_++;
}
```

- Initialization expression param:
  - its *type* is rvalue reference to T,
  - but its *value category* is *lvalue* (named entity; addressable).
- *Consequence:*
  - Inserted element would be initialized with *copy constructor*.
- *Solution* — expression that refers to the same object (push_back argument) but its category is *rvalue*:

```
void push_back(T&& param) {  // overload for rvalues
  if (size_ == capacity_) reserve(capacity_ ? 2 * capacity_ : 1);
  new (data_ + size_) T( std::move( param ) );
  size_++;
}
```

- Now, *move constructor* will be called if it is available.

Pushing-back into vector *(cont.)*

- Classes with copy constructor only (no resource owners, pre-C++11 classes,…)
  - It will work as expected = new element will be initialized with copy constructor.
  - *Recall — rvalues* may be bound to *constant lvalue references*.

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

```
struct X {
  X(int) { }
  X(const X&) { }  // copy constructor
                   // no move constructor
};
```

```
Vector<X> v;
v.push_back( X(1) );  // rvalue argument - inserted element initialized by copy constructor
```

- *Live demo* with `std::vector`: https://godbolt.org/z/MsE85fsGz.
- *Note — custom definition* of *copy constructor* suppresses *automatic generation* of *move constructor*.

Pushing-back into vector *(cont.)*

- What if we explicitly disable (delete) move constructor?

```
struct X {
  X(int) { }
  X(const X&) { }   // copy constructor
  X(X&&) = delete;  // explicitly deleted move constructor
};
```

```
Vector<X> v;
v.push_back( X(1) );  // error: use of deleted function 'X::X(X&&)'
```

  - *Live demo* with `std::vector`: https://godbolt.org/z/boqMEKjna.
- *Explanation:*
  - *Deleted member functions* **do participate** in overload resolution.
  - For *rvalue arguments*, both *copy* and *move* constructors are viable candidates.
  - But move constructor has higher priority (*rvalue reference* parameter for *rvalue* argument).
- *Conclusion:*
  - If there is custom-defined copy constructor, move constructor should be:
    - either *custom-defined* as well,
    - or *undefined* (and *undeclared*) completely (but not defined as deleted)