# Value categories



Every C++ expression belongs to exactly one of the following classifications, known as value categories: lvalue, xvalue, prvalue. There's an overlap between these, so a higher level of classification can be thought of as just rvalues and glvalues *(generalized lvalues)*.

Knowing the value category of an expression will allow you to make informed decisions about the lifetime of the expression, thus preventing common pitfalls which introduce undefined behavior and compilation errors.

```
              expression
              /        \
             /          \
         glvalue       rvalue
         /     \       /    \
        /       \     /      \
    lvalue      xvalue     prvalue
```

## PRvalues

prvalues are rvalues which are "pure," meaning they've never been bound to a name. They're often just called temporaries and are the result of any function which returns a non-reference value type, as well as most literals. prvalues can have their lifetime prolonged by binding to another reference type. The lifetime of a prvalue is the extent of the full expression.

| | |
|---|---|
| `42 // prvalue`<br>`true // prvalue`<br>valid | All literals, aside from string literals, are prvalues. String literals are lvalues. |
| `int foo();`<br>`foo(); // prvalue`<br>valid | Any function call returning a non-reference value type, including pointers, yields a prvalue. In the call expression, the value has not been given a name, which makes it pure. |
| `int a{}, b{}; // both lvalues`<br>`a + b; // prvalue`<br>valid | Like any function returning a non-reference value type, the result of arithmetic, when not using compound operators such as +=, is a prvalue. |
| `int a{}; // lvalue`<br>`&a; // prvalue`<br>valid | The address of any lvalue, is prvalue. Note that you can't take the address of prvalues. |
| `int a{}; // lvalue`<br>`static_cast<double>(a); // prvalue`<br>valid | The result of casting an lvalue to a non-reference type is a prvalue. This is no different with non-trivial types, too. |
| `[](int const a)`<br>`{ return a * a; }; // prvalue`<br><br>`int a{}; // lvalue`<br>`[&]{ return a * a; }; // prvalue`<br>valid | Anonymous functions, regardless of their capture, are prvalues like other literals. |
| `int a{}; // lvalue`<br>`a++; // prvalue`<br>valid | Postfix operators return a copy of the old value, which is a non-reference value type, so it's a prvalue. |
| `double{}; // prvalue`<br>`std::vector<database>{}; // prvalue`<br>valid | The construction of any type, using uniform initialization, which isn't a variable or member definition, is a prvalue. This is the same for both trivial and non-trivial types. |
| `void foo(std::string const &s);`<br>`foo("kitty"); // argument is a prvalue`<br>`foo(std::string{ "kitty" }); // same`<br>valid | Arguments passed to functions, including constructors, which are implicitly converted, are prvalues. This is commonly seen with std::string and various smart pointer types. |
| `int &a{ 42 }; // invalid`<br>invalid | An rvalue cannot be bound to an lvalue reference-to-non-const. |

## Lvalues

lvalues are glvalues which are bound to a name; typically, they appear on the left hand side of expressions (such as `a = 5`). lvalues may exist as a local, global, parameter, member, etc. The lifetime of an lvalue is the extent of the current scope.

| | |
|---|---|
| `"Meow!" // lvalue`<br>valid | Unlike all other literals, the string literal is an lvalue. This originated in C, since string literals are arrays and arrays in C can only exist in expressions as lvalues. |

| | |
|---|---|
| `int a{}; // lvalue`<br>`int& get()`<br>`{ return a; }`<br><br>`get(); // lvalue`<br>valid | A function call is an lvalue if the function returns a reference to an object, const or non-const. |
| `int a{}; // lvalue`<br>`++a; // lvalue`<br>valid | Prefix operators return a reference to the object, which is an lvalue. |
| `std::cout << 42; // lvalue`<br>valid | Even though the insertion operator is taking the prvalue `42`, the operator returns a reference to the ostream, so it's an lvalue. |
| `int a{}; // lvalue`<br>`int *p{ &a }; // lvalue`<br>`(p + 1); // prvalue`<br>`*(p + 1); // lvalue`<br>valid | While pointer arithmetic yields a prvalue, the indirection operator on a pointer results in an lvalue. |
| `int foo();`<br>`int &&a{ foo() }; // lvalue`<br>valid | Though `a` is an rvalue reference, it's named, so it's an lvalue. In order to get it back to an rvalue, in an expression, `std::move` or similar will be needed. |
| `struct foo`<br>`{ int a; };`<br><br>`foo f; // lvalue`<br>`f.a; // lvalue`<br>valid | A non-static data member of an lvalue is also an lvalue. |
| `int &&a{ 77 }; // lvalue`<br>`int &b{ a }; // lvalue`<br>valid | Though `a` is initialized with a prvalue, it becomes an lvalue. Since it's an lvalue, a normal lvalue reference can be taken from it. |
| `int a{ -7 }; // lvalue`<br>`int &&b{ a }; // invalid`<br>invalid | An lvalue cannot be bound to an rvalue reference without the usage of `std::move`. |

## Xvalues

xvalues are rvalues which are also glvalues, such as lvalues which have been casted to an rvalue reference. xvalues cannnot have their life prolonged by binding to another reference. You cannot take the address of an xvalue. The lifetime of an xvalue is the extent of the full expression.

| | |
|---|---|
| `bool b{ true }; // lvalue`<br>`std::move(b); // xvalue`<br>`static_cast<bool&&>(b); // xvalue`<br>valid | An lvalue that's moved will yield an xvalue. The same can be achieved by casting. |
| `int&& foo();`<br>`foo(); // xvalue`<br>valid | A function call which returns an rvalue reference yields an xvalue. |
| `int &&a{ 5 }; // lvalue`<br>`std::move(a); // xvalue`<br>`int &&b{ std::move(a) }; // lvalue`<br>`int const &c{ std::move(b) }; // lvalue`<br>valid | Like prvalues, xvalues can be bound to rvalue references and lvalue references-to-const. They cannot, however, have their lifetime prolonged. |
| `struct foo`<br>`{ int a; };`<br><br>`foo f; // lvalue`<br>`std::move(f).a; // xvalue`<br>`foo{}.a; // xvalue`<br>valid | A non-static data member of any rvalue is an xvalue. |

## Lifetime extension

prvalues can have their lifetime prolonged to be the lifetime of a reference to which they're bound. glvalues, meaning both lvalues and xvalues, don't have this same benefit, though it is still possible to bind them to other references.

```
struct T{};
T foo();

T const &ref{ foo() }; // lvalue
```

valid

A prvalue can be bound to an lvalue reference-to-const, which will prolong its lifetime to be the lifetime of the reference.

```
struct T{};
T foo();

T &&ref{ foo() }; // lvalue
```

valid

A prvalue can be bound to an rvalue reference, which will prolong its lifetime to be the lifetime of the reference.

```
struct T{};
T foo();

T &&ref{ std::move(foo()) }; // lvalue
T const &ref{ std::move(foo()) }; // lvalue
```

undefined-behavior

Moving a prvalue yields an xvalue. While that can be bound to an rvalue reference or an lvalue reference-to-const, both cases are undefined behavior, since neither will prolong the lifetime of an xvalue.

```
int &&a{ 5 }; // lvalue
int const &b{ std::move(a) }; // lvalue
```

valid

While it's well-defined to bind an xvalue to an lvalue reference-to-const, no lifetimes will be prolonged, so it must be done with care.

## Common patterns and mistakes

*Returning reference to const local*

```
int const& foo()
{
  int ret{}; // lvalue
  return ret; // rvalue
}
```

undefined-behavior

```
int foo()
{
  int ret{}; // lvalue
  return ret; // rvalue
}
```

§12.8 (32): When the criteria for elision of a copy operation are met or would be met save for the fact that the source object is a function parameter, and the object to be copied is designated by an lvalue, overload resolution to select the constructor for the copy is first performed as if the object were designated by an rvalue.

valid

*Returning reference to const parameter*

```
template <typename T>
T const& get(std::string const &key,
             T const &fallback)
{
  auto const &found(find(key)); // lvalue
  if(found) // lvalue
  { return *found; } // lvalue
  return fallback; // lvalue
}

int a{}; // lvalue
get("meow", a); // lvalue, well-defined
get("meow", 0); // lvalue, undefined
```

undefined-behavior

```
template <typename T>
T get(std::string const &key,
      T const &fallback)
{
  auto const &found(find(key)); // lvalue
  if(found) // lvalue
  { return *found; } // lvalue
  return fallback; // lvalue
}

int a{}; // lvalue
get("meow", a); // prvalue
get("meow", 0); // prvalue
```

An lvalue reference-to-const parameter may be bound to an outside lvalue, or it may be prolonging the lifetime of a prvalue. Thus, it's not well-defined to return an lvalue reference-to-const bound by that parameter. In this case, return a non-reference type.

valid

*Moving an object properly*

```
{
  std::vector<int> a{ calculate_things() };

  // done with a, so move it
  use_results(std::move(a)); // move gives an xvalue

  // a is now moved-from
}

// can be a non-reference parameter
void use_results(std::vector<int> v);

// can explicitly require an rvalue, to
// prevent accidental copies
void use_results(std::vector<int> &&v);
```

valid

You should use `std::move` to tag objects as xvalues so that they can be transferred optimally.

*Move in as rvalue, return as non-reference*

```
std::vector<int> add_some(std::vector<int> &&v) // lvalue
{
  v.push_back(42);
  return std::move(v); // xvalue
}

std::vector<int> v; // lvalue
v = add_some(std::move(v)); // sends in xvalue
```

```cpp
std::vector<int> add_some(std::vector<int> &&v) // lvalue
{
  v.push_back(42);
  return v; // lvalue -- non-idiomatic
}

std::vector<int> v; // lvalue
v = add_some(std::move(v)); // sends in xvalue
```

Parameters of a reference-type will not be automatically candidates for return value optimization, as they could be referring to objects outside the scope of the function. In order to avoid deep copying here, use `std::move` to coerce the parameter to an xvalue when returning.

Note, do not use this technique when returning non-reference parameters or objects local to the function's scope; they will automatically be returned as rvalues, if possible.

valid

*Hanging onto an xvalue member*

non-idiomatic

```cpp
struct foo
{ int a; };

foo get();

int const &b{ get().a }; // a is an xvalue
```

undefined-behavior

```cpp
struct foo
{ int a; };

foo get();

int const b{ get().a }; // copy the xvalue
int const c{ std::move(get().a) }; // move the xvalue
```

Members of rvalue objects are xvalues; xvalues cannot have their lifetime extended by binding to a reference-to-non-const or rvalue-reference, though the binding is valid and will compile. When pulling a member out of an rvalue object, prefer to copy or move the member.

valid

*Hanging onto an rvalue container element*

```cpp
std::vector<int> get();

get().at(0); // lvalue
int const &a{ get().at(0) }; // undefined
```

undefined-behavior

```cpp
std::vector<int> get();

int const a{ get().at(0) }; // copy
int const b{ std::move(get().at(0)) }; // move
```

A container, returned as an rvalue, does not have its lifetime extended by binding a reference to one of its members or elements. At the end of the expression, the container will be destroyed and the reference will be dangling.

valid

*Hanging onto an lvalue container element*

```cpp
std::vector<int> const& get();

get().at(0); // lvalue
int const &a{ get().at(0) }; // lvalue
```

valid

A container returned as an lvalue doesn't need its lifetime extended, so binding a member or element from it to an lvalue reference is well-defined.

*Hanging onto an lvalue member of an rvalue*

```cpp
struct foo
{
  int a{};
  int const& get_a() // lvalue
  { return a; }
};

foo{}; // prvalue
foo{}.get_a(); // lvalue

int const &a{ foo{}.get_a() }; // undefined
```

undefined-behavior

```cpp
struct foo
{
  int a{};
  int const& get_a() // lvalue
  { return a; }
};

int const a{ foo{}.get_a() }; // copy
```

A function returning an lvalue reference always results in an lvalue reference, even when it's called on an rvalue object. In this case, `foo{}` is a prvalue, but calling `get_a()` on it yields an lvalue. As shown, just because a function returns an lvalue member doesn't make it safe to bind to another reference.

valid

*Binding an rvalue to a string_view*

```cpp
boost::string_view s{ std::string{ "foo" } }; // undefined

std::string get();
boost::string_view s{ get() }; // undefined
```

undefined-behavior

```cpp
std::string s{ "meow" }; // lvalue

std::string get();
std::string const &s{ get() }; // lvalue
```

A string_view is like an lvalue reference to a `std::string`, or C string. It doesn't extend the string's lifetime and it should be thought of as just holding onto members of the string: begin and end.

valid

*Binding an rvalue to a string_view parameter*

```cpp
void foo(boost::string_view const &s) // s is an lvalue
{ }

foo("meow"); // From lvalue literal
foo(std::string{ "meow" }); // From prvalue
```

valid

Binding an rvalue string to a string_view isn't always undefined behavior. In the case of parameters, the rvalue will live as long as the full expression, which is the duration of the function call. In this manner, string_views can provide a type-agnostic way of serving `std::string`, C strings, and other string_views.