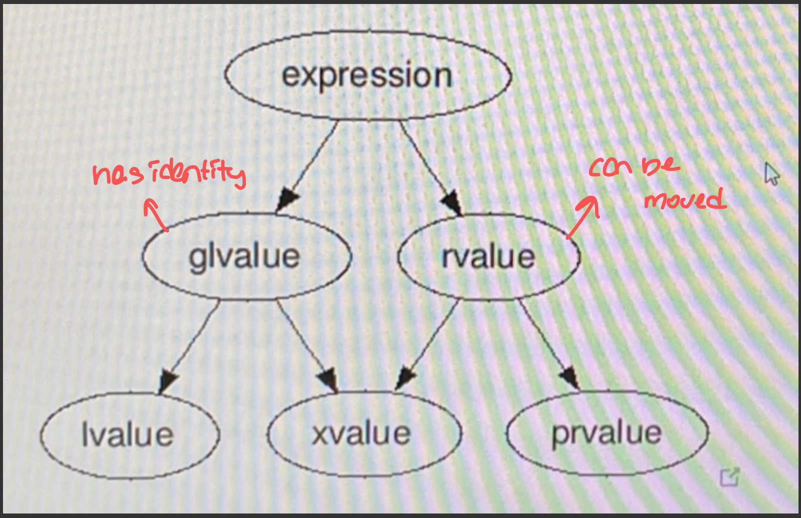
* Value Category expression ile ilgili iki önemli özelliği belirtir. → Expressions Identity
→ Implicitly moving its value legally

• Expression Identity:
- Değişken adı olan objelerin identity'si vardır.
- Expressionda belirtilmese bile, objenin olabilir.

- Implicitly Move:
- Expression bir fonksiyon parametresi olarak kullanılsaydı r-value parametreye bind edilebilir mi?

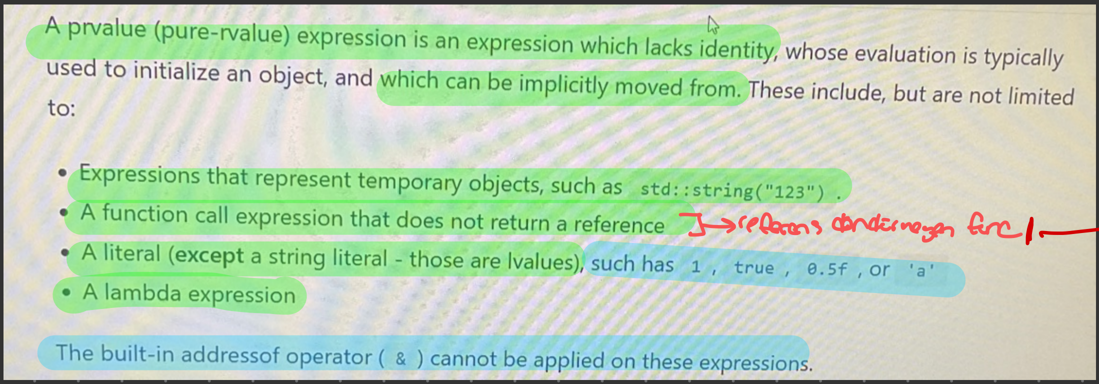
* Bu kategoriler L value → Expression with identity but not movable from
PR value → Expression without identity but movable from
X value → Expression with identity and movable from

*


→ Bunları kapsayan üst başlıklar var.

→ C++'da non-movable, non-identified expression olamaz!

*** PR Value:**
- Lacks identity
- Used for initializing an object

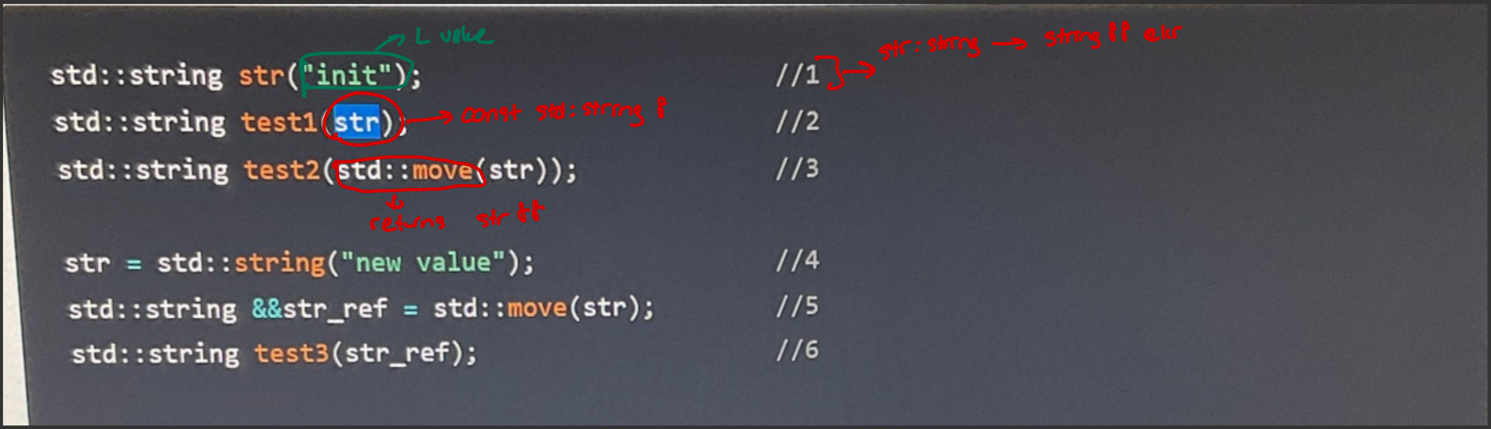

A prvalue (pure-rvalue) expression is an expression which lacks identity, whose evaluation is typically used to initialize an object, and which can be implicitly moved from. These include, but are not limited to:

- Expressions that represent temporary objects, such as `std::string("123")`.
- A function call expression that does not return a reference] → referans döndürmeyen func! → Referans dönen functionlar L valuedur.
- A literal (**except** a string literal - those are lvalues), such has `1`, `true`, `0.5f`, or `'a'`
- A lambda expression

The built-in addressof operator (`&`) cannot be applied on these expressions.

* R Value:

- Can be moved implicitly.
- R valuelar T&& (type ref ref) parametre alan bir fonksiyonun argümanı olabilir.
- Eğer bu fonksiyonlarda başka bir argüman kullanılırsa → func overload'a gider.
- PR value ve X value'nın üst kategorisi
- std::move() ile R value yaratılabilir ⟶ X value'ya dönüştürür.



```
std::string str("init");                    //1
std::string test1(str);                     //2
std::string test2(std::move(str));          //3

str = std::string("new value");             //4
std::string &&str_ref = std::move(str);     //5
std::string test3(str_ref);                 //6
```

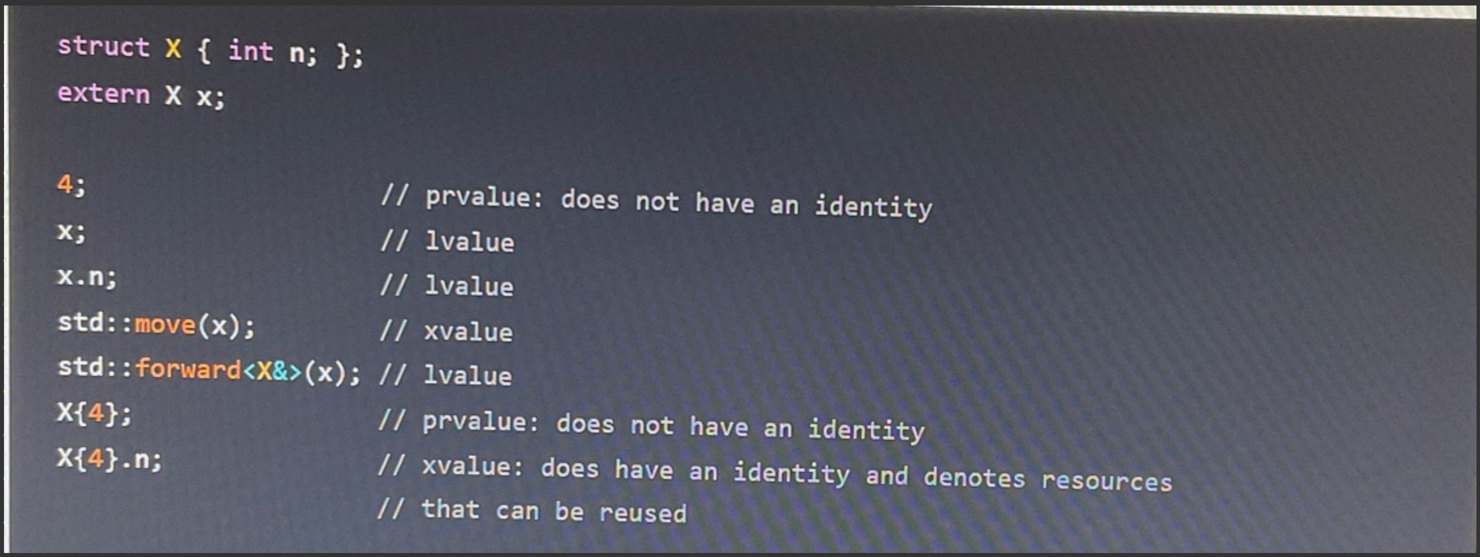
L value

str: string → string && alır

const std::string &

returns str &&

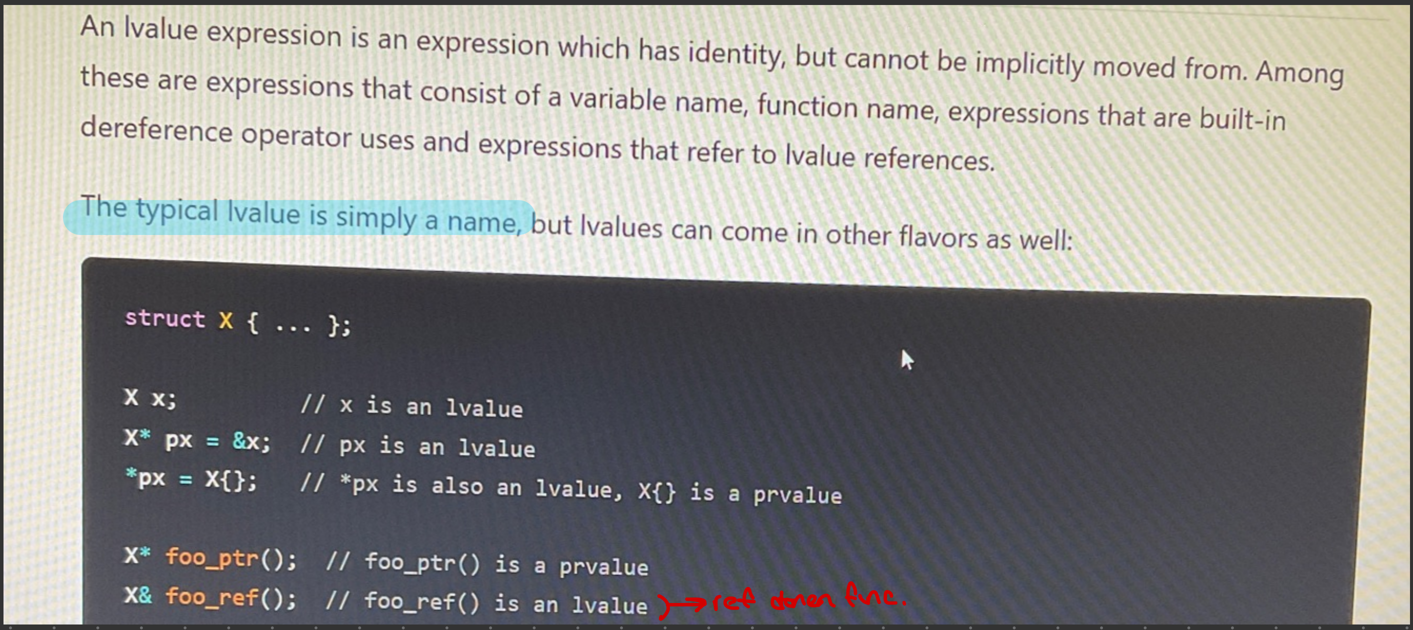
* X Value:



```
struct X { int n; };
extern X x;

4;                    // prvalue: does not have an identity
x;                    // lvalue
x.n;                  // lvalue
std::move(x);         // xvalue
std::forward<X&>(x);  // lvalue
X{4};                 // prvalue: does not have an identity
X{4}.n;               // xvalue: does have an identity and denotes resources
                      // that can be reused
```

* L Value:



An lvalue expression is an expression which has identity, but cannot be implicitly moved from. Among these are expressions that consist of a variable name, function name, expressions that are built-in dereference operator uses and expressions that refer to lvalue references.

The typical lvalue is simply a name, but lvalues can come in other flavors as well:

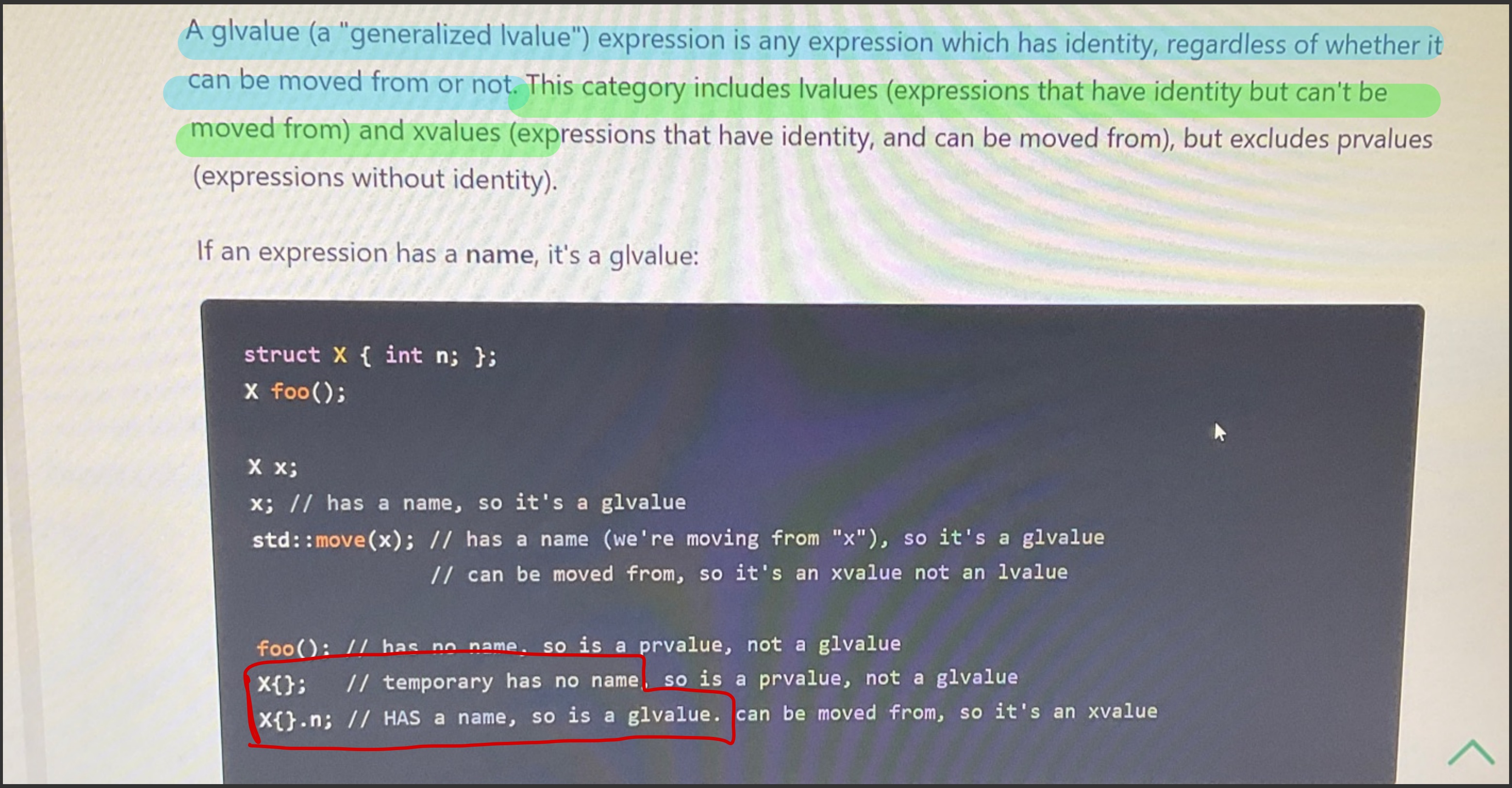
```
struct X { ... };

X x;         // x is an lvalue
X* px = &x;  // px is an lvalue
*px = X{};   // *px is also an lvalue, X{} is a prvalue

X* foo_ptr();  // foo_ptr() is a prvalue
X& foo_ref();  // foo_ref() is an lvalue
```

→ ref dönen func.

* GL Value:



A glvalue (a "generalized lvalue") expression is any expression which has identity, regardless of whether it can be moved from or not. This category includes lvalues (expressions that have identity but can't be moved from) and xvalues (expressions that have identity, and can be moved from), but excludes prvalues (expressions without identity).

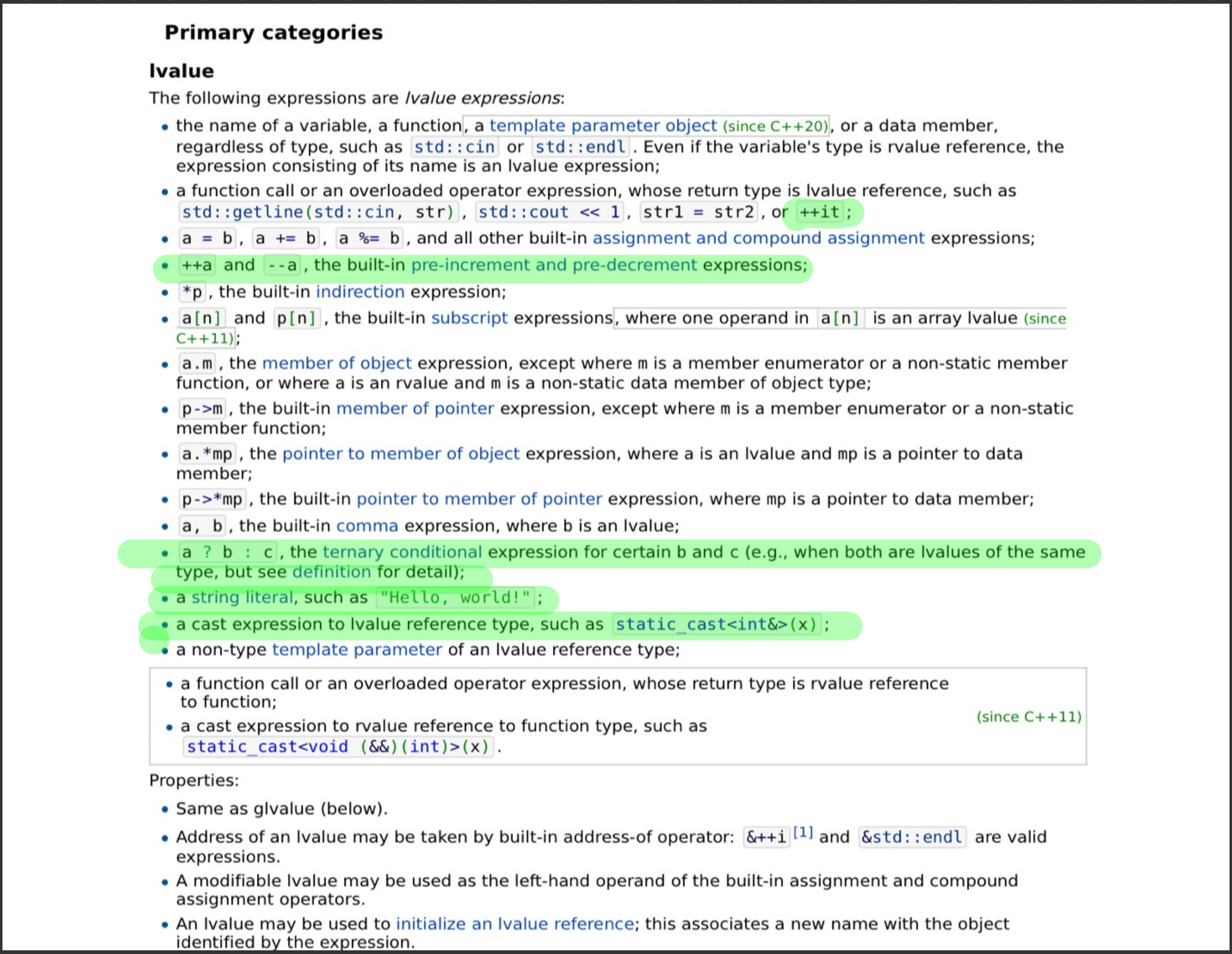
If an expression has a **name**, it's a glvalue:

```
struct X { int n; };
X foo();

X x;
x; // has a name, so it's a glvalue
std::move(x); // has a name (we're moving from "x"), so it's a glvalue
              // can be moved from, so it's an xvalue not an lvalue

foo(); // has no name, so is a prvalue, not a glvalue
X{};   // temporary has no name, so is a prvalue, not a glvalue
X{}.n; // HAS a name, so is a glvalue. can be moved from, so it's an xvalue
```

Cpp Reference



Primary categories

lvalue

The following expressions are *lvalue expressions*:

- the name of a variable, a function, a template parameter object (since C++20), or a data member, regardless of type, such as `std::cin` or `std::endl`. Even if the variable's type is rvalue reference, the expression consisting of its name is an lvalue expression;
- a function call or an overloaded operator expression, whose return type is lvalue reference, such as `std::getline(std::cin, str)`, `std::cout << 1`, `str1 = str2`, or `++it`;
- `a = b`, `a += b`, `a %= b`, and all other built-in assignment and compound assignment expressions;
- `++a` and `--a`, the built-in pre-increment and pre-decrement expressions;
- `*p`, the built-in indirection expression;
- `a[n]` and `p[n]`, the built-in subscript expressions, where one operand in `a[n]` is an array lvalue (since C++11);
- `a.m`, the member of object expression, except where `m` is a member enumerator or a non-static member function, or where a is an rvalue and `m` is a non-static data member of object type;
- `p->m`, the built-in member of pointer expression, except where `m` is a member enumerator or a non-static member function;
- `a.*mp`, the pointer to member of object expression, where a is an lvalue and `mp` is a pointer to data member;
- `p->*mp`, the built-in pointer to member of pointer expression, where `mp` is a pointer to data member;
- `a, b`, the built-in comma expression, where b is an lvalue;
- `a ? b : c`, the ternary conditional expression for certain b and c (e.g., when both are lvalues of the same type, but see definition for detail);
- a string literal, such as `"Hello, world!"`;
- a cast expression to lvalue reference type, such as `static_cast<int&>(x)`;
- a non-type template parameter of an lvalue reference type;

> - a function call or an overloaded operator expression, whose return type is rvalue reference to function;
> - a cast expression to rvalue reference to function type, such as `static_cast<void (&&)(int)>(x)`.
>
> (since C++11)

Properties:

- Same as glvalue (below).
- Address of an lvalue may be taken by built-in address-of operator: `&++i`[1] and `&std::endl` are valid expressions.
- A modifiable lvalue may be used as the left-hand operand of the built-in assignment and compound assignment operators.
- An lvalue may be used to initialize an lvalue reference; this associates a new name with the object identified by the expression.

prvalue

The following expressions are *prvalue expressions*:

- a literal (except for string literal), such as `42`, `true` or `nullptr`;
- a function call or an overloaded operator expression, whose return type is non-reference, such as `str.substr(1, 2)`, `str1 + str2`, or `it++`;
- `a++` and `a--`, the built-in post-increment and post-decrement expressions;
- `a + b`, `a % b`, `a & b`, `a << b`, and all other built-in arithmetic expressions;
- `a && b`, `a || b`, `!a`, the built-in logical expressions;
- `a < b`, `a == b`, `a >= b`, and all other built-in comparison expressions;
- `&a`, the built-in address-of expression;
- `a.m`, the member of object expression, where `m` is a member enumerator or a non-static member function[2];
- `p->m`, the built-in member of pointer expression, where `m` is a member enumerator or a non-static member function[2];
- `a.*mp`, the pointer to member of object expression, where `mp` is a pointer to member function[2];
- `p->*mp`, the built-in pointer to member of pointer expression, where `mp` is a pointer to member function[2];
- `a, b`, the built-in comma expression, where `b` is an rvalue;
- `a ? b : c`, the ternary conditional expression for certain b and c (see definition for detail);
- a cast expression to non-reference type, such as `static_cast<double>(x)`, `std::string{}`, or `(int)42`;
- the `this` pointer;
- an enumerator;
- a non-type template parameter of a scalar type;

• a lambda expression, such as `[](int x){ return x * x; }`;	(since C++11)
• a requires-expression, such as `requires (T i) { typename T::type; }`; • a specialization of a concept, such as `std::equality_comparable<int>`.	(since C++20)

Properties:

- Same as rvalue (below).
- A prvalue cannot be polymorphic: the dynamic type of the object it denotes is always the type of the expression.
- A non-class non-array prvalue cannot be cv-qualified, unless it is materialized in order to be bound to a reference to a cv-qualified type (since C++17). (Note: a function call or cast expression may result in a prvalue of non-class cv-qualified type, but the cv-qualifier is generally immediately stripped out.)
- A prvalue cannot have incomplete type (except for type `void`, see below, or when used in `decltype` specifier)
- A prvalue cannot have abstract class type or an array thereof.

xvalue

The following expressions are *xvalue expressions*:

- `a.m`, the member of object expression, where a is an rvalue and `m` is a non-static data member of an object type;
- `a.*mp`, the pointer to member of object expression, where a is an rvalue and `mp` is a pointer to data member;
- `a ? b : c`, the ternary conditional expression for certain b and c (see definition for detail);

• a function call or an overloaded operator expression, whose return type is rvalue reference to object, such as `std::move(x)`; • `a[n]`, the built-in subscript expression, where one operand is an array rvalue; • a cast expression to rvalue reference to object type, such as `static_cast<char&&>(x)`;	(since C++11)
• any expression that designates a temporary object, after temporary materialization.	(since C++17)

Properties:

- Same as rvalue (below).
- Same as glvalue (below).

In particular, like all rvalues, xvalues bind to rvalue references, and like all glvalues, xvalues may be polymorphic, and non-class xvalues may be cv-qualified.

Mixed categories

glvalue

A *glvalue expression* is either lvalue or xvalue.

Properties:

- A glvalue may be implicitly converted to a prvalue with lvalue-to-rvalue, array-to-pointer, or function-to-pointer implicit conversion.
- A glvalue may be polymorphic: the dynamic type of the object it identifies is not necessarily the static type of the expression.
- A glvalue can have incomplete type, where permitted by the expression.

rvalue

An *rvalue expression* is either prvalue or xvalue.

Properties:

- Address of an rvalue cannot be taken by built-in address-of operator: `&int()`, `&i++`[3], `&42`, and `&std::move(x)` are invalid.
- An rvalue can't be used as the left-hand operand of the built-in assignment or compound assignment operators.
- An rvalue may be used to initialize a const lvalue reference, in which case the lifetime of the object identified by the rvalue is extended until the scope of the reference ends.

- An rvalue may be used to initialize an rvalue reference, in which case the lifetime of the object identified by the rvalue is extended until the scope of the reference ends.
- When used as a function argument and when two overloads of the function are available, one taking rvalue reference parameter and the other taking lvalue reference to const parameter, an rvalue binds to the rvalue reference overload (thus, if both copy and move constructors are available, an rvalue argument invokes the move constructor, and likewise with copy and move assignment operators).

(since C++11)