**Value Category**

# Motivation

Compiler can steal data from temporaries

To explicitly steal data from object use std::move()

Wrong uses of std::move() may lead to performance overhead and may not result in what we intended

#include <iostream>

#include <string>

class LearnValueCategories {

std::string m\_data;

public:

~LearnValueCategories() {

std::cout << "Destructor\n";

}

LearnValueCategories(std::string data) : m\_data(data) {

std::cout << "Constructor\n";

}

LearnValueCategories(const LearnValueCategories & other)

: m\_data(other.m\_data) {

std::cout << "Copy Constructor\n";

}

LearnValueCategories(LearnValueCategories && other)

: m\_data(std::move(other.m\_data)) {

std::cout << "Move Constructor\n";

}

};

LearnValueCategories getValueCategories() {

return LearnValueCategories("returning LearnValueCategories value");

}

const LearnValueCategories getValueCategories\_Const() {

return LearnValueCategories("returning const LearnValueCategories value");

}

int main() {

std::cout << "=== Constructing and moving object ===\n";

{ LearnValueCategories lv("constructor"); } // 1 object created

std::cout << '\n';

{ LearnValueCategories lv1("this will be moved"); // 1 object created

LearnValueCategories lv2 = std::move(lv1); // 0 object created

}

std::cout << "=== Constructing object using return value ===\n";

{

LearnValueCategories lv = getValueCategories(); // 1 object created

}

std::cout << '\n';

{

LearnValueCategories lv = getValueCategories\_Const(); // 1 object created

}

std::cout << "=== Constructing object using return value and moving ===\n";

{

LearnValueCategories lv = std::move(getValueCategories()); // 1 object created

}

std::cout << '\n';

{

LearnValueCategories lv = std::move(getValueCategories\_Const()); // 2 object created

}

return 0;

}

Output:

=== Constructing and moving object ===

Constructor

Destructor

Constructor

Move Constructor

Destructor

Destructor

=== Constructing object using return value ===

Constructor

Destructor

Constructor

Destructor

=== Constructing object using return value and moving ===

Constructor

Move Constructor

Destructor

Destructor

Constructor

Copy Constructor

Destructor

Destructor

# Intro to Value Categories

## What are value Categories

Value Categories were inherited from C

C has "lvalue expression"

Originally it referred to the location of the expression in assignment statement

int val = 42;

**lvalue** (left-value) was on the left of the assignment

**rvalue** (right-value) was on the right of the assignment

Value Category of an entity defines:

1. Its lifetime:

* Can it be moved from
* Is it a temporary
* Is it observable after change, etc.

1. Its identity:

* Object has identity if its address can be taken and used safely

Value Categories affect two very important aspects relevant to libraries:

1. Performance
2. Overload resolution

Value Category is a **quality of an expression**

class Data {

int m\_val;

public:

Data(int val) : m\_val(val) { }

};

void fun(Data && data) {

data = 10;

}

int main() {

Data && data\_v = 11; // OK

fun(data\_v); // CE, data\_v is lvalue

fun(Data(12)); // OK

return 0;

}

What (misleadingly) looks like the value category, can in fact be the type

* data\_v’s Type: rvalue reference to Data
* data\_v’s Value Category: lvalue

What (misleadingly) looks like the same entity, is, in fact, not!

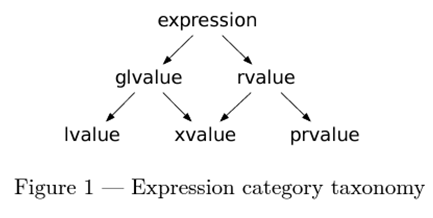
**The entity can have different VC in different contexts**

During a function call:

* Step I : Calls constructor, creates an unnamed temp Data(12)
* Step II : 'Data(12)' binds to the rvalue reference data
* Step III : The entity which used to be 'Data(12)' has a name data, therefore, in the scope of fun(), data is now an lvalue

Each expression has two properties:

1. A type (including CV qualifiers)
2. A value category



Introduced in: [N3055](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3055.pdf) : A Taxonomy of Expression Value Categories (William M. Miller, 2010)

Value Categories changed throughout C++ versions, affected by the rules defining references , move semantics and copy elision

## Evolution of value Categories

**C language**

Three types of expressions:

* lvalue expression
* Non lvalue object expression
* Function designator expression

**C++98: added lvalue references**

Expression is either an lvalue or an rvalue

* lvalue: Objects, Functions, References
* rvalue: Non-lvalue (can be bound by const lvalue reference)

**C++03**

* No significant change

**C++11**

* added rvalue references
* move semantics

|  |  |  |
| --- | --- | --- |
|  | Has Identity (glvalue) | Doesn’t have identity |
| Can’t be moved from | lvalue |  |
| Can be moved from (rvalue) | xvalue | prvalue |

**C++17**

Added guaranteed copy elision

The result of a prvalue is the value that the expression stores into its context

|  |  |  |
| --- | --- | --- |
|  | Has Identity (glvalue) | Doesn’t have identity |
| Can’t be moved from | lvalue |  |
| Can be moved from (rvalue) | xvalue | prvalue’s materialization |

**C++20**

[N4861] (March 2020)

* [P0527](https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0527r1.html): Implicitly move from rvalue references in return statements (David Stone)
* Editorial: Moved Value Categories section from [basic] to [expt]

[N4868] (Oct 2020)

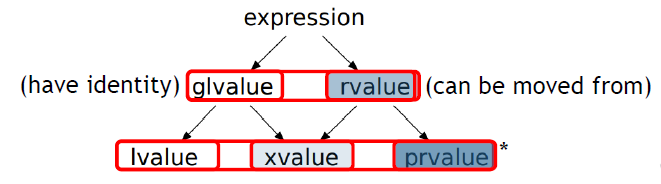
* Removed “bit field” from the value categories primary definitions

**C++23 draft (latest)**

[P0847](https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p0847r7.html): Deducing this Gašper Ažman, Sy Brand, Ben Deane, Barry Revzin

P0847 also added like\_t

[P2445](https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2445r0.pdf): std::forward\_like



**Main Categories (classification only)**

**glvalue:** expression whose evaluation determines the identity of an object or function

**rvalue:** a prvalue or an xvalue

**Subcategories**

**lvalue:** glvalue that is not an xvalue

**xvalue:** glvalue that denotes an object whose resources can be reused (usually because it is near the end of its lifetime)

**prvalue:** expression whose evaluation initializes an object, or computes the value of the operand of an operator, as specified by the context in which it appears, or an expression that has type cv void

### Examples

**Lvalue:** lvalue is something that have an identity

struct Data { int n; int pn = n; };

Data& getData (Data& d) { return d }

int a = 42; // a is lvalue

int b = a; // b is lvalue

int& iref = a; // iref is lvalue

int\* iptr = &a; // iptr is lvalue

int&& ra = 42;

a++;

++a; // ++a is lvalue

int arr[] = {1, 2, 3}; // arr[] is lvalue

arr[0] = 73; // arr[0] is lvalue

Data d; // d is lvalue

(&d)->n = 42; // (&d)->n is lvalue

d.n = 73; // d.n is lvalue

\*d.pn = 42; // \*d.pn is lvalue

string s ="Hello World"; // s is lvalue

a==b ? b : c; // is lvalue when b and c are lvalues

Data c = getData(d); // return value of function that creates c is lvalue

NOTE: ra has the type: rvalue reference to int, with the value category: lvalue

**prvalue**

struct Data {

int n;

int foo() { this->n = 4 ; } // this is prvalue, 4 is prvalue

};

int a = 42; // 42 is prvalue

int\* pa = &a; // &a is prvalue

pa = nullptr; // nullptr is prvalue

a++; // built-in post increment a++ is prvalue

++a;

auto l = [](){ return 2 ;}; // lambda [](){ return 2 ;} is prvalue

Data d;

Data\* dp = &d;

Data();

d->n = 6; // 6 is prvalue

d.n = 6; // 6 is prvalue

string s ="Hello World";

a==a ? throw 4 : throw 2; // throw is prvalue void

bool equals = a==42; // a==42 is prvalue

**xvalue**

struct Data { int n; int\* pn = &n; };

Data d1 = Data(42); // Data(42) is xvalue

d1.\*pn = 73;

Data d2 = std::move(d1); // std::move(d1) is xvalue

// Data().n is xvalue, because in order to get n we need to create the object even if its temporary

Data().n; // Data().n is xvalue

Data getData () {

return Data(73);

}

Data d3 = getData(); // function return value is xvalue

d1==d2 ? Data(42) : Data(73); // return value of ternary is xvalue

# Value Categories in Practice

## Intro to Overload resolution

Overload resolution of functions affected by value Categories

To Compile a function call, compiler creates list of candidates, then finds correct Overload

**Overload resolution**

* Picking the correct function from group of candidates
* Function candidates are picked according to function name (Name lookup)
* May include Template Argument Deduction (TAD) and Argument Dependent Lookup (ADL)

**Template Argument Deduction (TAD)**

* Value category of an expression affects type deduction
* Involves non-trivial deduction rules (out of scope)

(e.g. deduction of T from multiple parm with different value categories)

## The details of binding

Expressions with different Value Categories "bind" to different types of References

The Reference type which binds the expression determines the permitted operations

Binding rules are important for

1. Initialization or assignment
2. Function call (including non-static class member function called on an object)
3. Return statement

int main() {

int val = 5;

int& ref = val;

ref = 65;

const int& cref = val;

cref = 67; // CE

int&& rref = val + 75;

rref = 68;

const int&& crref = val + 80;

crref = 75; // CE

}

### 1. Initialization or Assignment

Rule

|  |  |  |
| --- | --- | --- |
|  | Binds lvalues? | Binds rvalues? |
| lvalue reference | ✓ | ❌ |
| const lvalue reference | ✓ | ✓ |
| rvalue reference | ❌ | ✓ |
| const rvalue reference | ❌ | ✓ |

int main() {

int val = 5;

int& ref1 = val;

int& ref2 = 10; // CE

const int& cref1 = val;

const int& cref2 = 20;

int&& rref1 = val; // CE

int&& rref2 = val + 75;

const int&& crref1 = val; // CE

const int&& crref2 = val + 80;

The Lifetime of an object can be extended by binding to references

const lvalue reference extends lifetime of an object (not allowing modification)

rvalue reference extends lifetime of a temporary objects

### 2. Function Call

class Data {

int m\_val;

public:

Data(int val) : m\_val(val) { }

};

void fun(Data & ) { } // 1

void fun(const Data & ) { } // 2

void fun(Data && ) { } // 3

void fun(const Data && ) { } // 4

const Data getData(int x) {

return Data(x);

}

int main() {

Data data(5);

Data& lval\_ref = data;

const Data& clval\_ref = data;

fun(lval\_ref); // lvalue: 1, 2

fun(clval\_ref); // const lvalue: 2

Data&& rval\_ref = Data(55);

const Data&& crval\_ref = Data(55);

fun(rval\_ref); // lvalue: 1, 2

fun(crval\_ref); // const lvalue: 2

fun(Data(11)); // xvalue: 3, 4, 2

fun(getData(12)); // const xvalue: 4, 2

}

Limitations on the object in the context of the function are according to the binding function

|  |  |  |
| --- | --- | --- |
|  | Function can modify data? | Caller can observe (old) data? |
| lvalue reference | ✓ | ✓ |
| const lvalue reference | ❌ | ✓ |
| rvalue reference | ✓ | ❌ |
| const rvalue reference | ❌ | ❌ |

Non-static member functions behave similarly

**deleted const rvalue reference overload will block rvalue binding**

## Copy elision optimizations

### 3. Return Statement

Starting from C++17, the behavior of VCs is affected by: [P0135](https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0135r1.html): Guaranteed copy elision (…)"

There are two mandatory elisions of copy and move constructors:

1. Object initialization

Data d1 = Data(Data(42)); // 1 CTOR (avoids: CTOR, Copy CTOR)

1. Return statement:

An unnamed Return Value Optimization (RVO):

Data returnValue(int x) { return Data(x); }

Data d2 = returnValue(42); // 1 CTOR (avoids: CTOR, Move CTOR)

No change in non-mandatory Named Return Value Optimization (NRVO)

Data namedReturnValue(int x) { Data d(x); return d; }

Data d3 = namedReturnValue(42); // 1 CTOR (avoids: CTOR, Move CTOR)

#include <iostream>

class Data {

int m\_val;

public:

Data(int val) : m\_val(val) {

std::cout << "Constructor\n";

}

};

Data returnValue(int x) { return Data(x); }

Data namedReturnValue(int x) { Data d(x); return d; }

int main() {

Data d1 = Data(Data(42)); // 1 CTOR (avoids: CTOR, Copy CTOR)

Data d2 = returnValue(42); // 1 CTOR (avoids: CTOR, Move CTOR)

Data d3 = namedReturnValue(42); // 1 CTOR (avoids: CTOR, Move CTOR)

return 0;

}

Output:

Constructor

Constructor

Constructor

### Return Statement: Materialization

Temporary materialization conversion [[conv.rval]](https://eel.is/c++draft/conv.rval)

A prvalue of type T can be converted to an xvalue of type T.

This conversion initializes a temporary object of type T from the prvalue by evaluating the prvalue with the temporary object as its result object, and produces an xvalue denoting the temporary object

[in order to materialize] T shall be a complete type.

Data returnValue(int x) { return Data(x); }

Data d2 = returnValue(42);

1. prvalue of type Data

2. Temp data evaluating prvalue

3. Produces an xvalue

Data namedReturnValue(int x) { Data d(x); return d; }

Data d3 = namedReturnValue(42);

1. lvalue of type Data

2. lvalue initializes temp data

3. Produces an xvalue

### Return Statement: const vs non const

#include <iostream>

class Data {

int m\_val;

public:

Data(int val) : m\_val(val) {

std::cout << "Constructor\n";

}

};

Data getNonConstData(int x){ return Data(x); }

const Data getConstData(int x){ return Data(x); }

int main() {

// for non-const return value

Data data = getNonConstData(13); // lval (no temporary, RVO)

Data& ref\_data = getNonConstData(13); // Error: xval -> non const lval ref

const Data& cref\_data = getNonConstData(13); // xval -> const lval ref (extends lifetime)

Data&& rref\_data = getNonConstData(13); // xval -> rval ref (extends lifetime)

{

// for const return value

Data data = getConstData(13); // lval (no temporary, RVO)

Data& ref\_data = getConstData(13); // Error: const xval -> non const lval ref

const Data& cref\_data = getConstData(13); // const xval -> const lval ref (extends lifetime)

Data&& rref\_data = getConstData(13); // Error: const xval -> rval ref

const Data&& crref\_data = getConstData(13); // const xval -> const rval ref (extends lifetime)

}

return 0;

}

# Vlaue Categories in Generic Code

**Template instantiation and overload resolution are affected by the value category of the expression**

When a function template participates in overload resolution, Template Argument Substitution Occurs

Template arguments can be affected by lvalue-to-rvalue, array-to-pointer, or function-to-pointer implicit conversion (as in auto)

## Reference collision

In case of concatenation of multiple '&' symbols:

* In generic code
* In code using type aliases

Compiler performs Reference Collision:

typedef int& lvalue\_ref\_t;

typedef int&& rvalue\_ref\_t;

template<typename T>

void fun(T t) { }

template<>

void fun(lvalue\_ref\_t t) { } // lvalue\_ref\_t -> int&

template<>

void fun(lvalue\_ref\_t& t) { } // lvalue\_ref\_t& -> int&& -> int&

int main() {

int val = 5;

lvalue\_ref\_t& lref = val; // lvalue\_ref\_t& -> int&& -> int&

lvalue\_ref\_t&& rref = val; // lvalue\_ref\_t&& -> int&&& -> int&

rvalue\_ref\_t& rref1 = val; // rvalue\_ref\_t& -> int&&& -> int&

rvalue\_ref\_t&& rref2 = 72; // rvalue\_ref\_t&& -> int&&&& -> int&&

return 0;

}

## Forwarding reference

Forwarding parameters inside a function template should consider Value Categories

The term for them was first suggested by Scott Myers, "universal reference"

Later formalized as "forwarding reference" [[temp.deduct.call]](https://timsong-cpp.github.io/cppwp/n4861/temp.deduct#def:forwarding_reference)

Due to TAD, "rvalue reference" has a special meaning in context of function template

T&& keeps the value category of the type the instantiation is based on

Type is deduced in order to keep the value category of the expression

#include <utility>

template<typename T>

void fun(T&& t) { }

int main() {

int val = 5;

const int & cref = val;

int&& rref = 13;

fun(val); // T&& -> int&& -> T = int&

fun(cref); // T&& -> const int&& -> T = const int&

fun(std::move(val)); // T&& -> int&& -> T = int

auto && uref = rref; // T&& -> int&& -> T = int&

return 0;

}

# Tools for handling value categories

when template function is considered for overload resolution, Template Argument Substitution occurs

Standard library and language provide different methods for querying and adjusting value category

Following type\_traits are example

std::is\_lvalue\_reference<T>

std::is\_rvalue\_reference<T>

Following section will use type traits, decltype and overload resolution to determine value category

* decltype specifier
* std::move
* std::forward
* std::decay
* std::remove\_reference
* std::declval
* std::is\_same

## decltype specifier

decltype(expression)

Evaluates an expression, yields its type + value category (a.k.a. the declared type)

**decltype (unlike auto) preserves value category**

For an expression of type T,

If expression is

xvalue, yields T&&

lvalue, yields T&

prvalue, yields T

Can be used instead of a type, as a placeholder which preserves value categories

#include <utility>

int fun(int i) { return std::move(i); }

int main() {

int i = 73;

auto a = fun(i); // Type: int | VC: lvalue

decltype(auto) b = fun(i); // Type: rvalue ref | VC: lvalue

return 0;

}

The fine print:

* The T prvalue doesn’t materialize, so T can be an incomplete type (C++17)
* If evaluation fails (entity is not found or overload resolution fails), program is ill formed

**((expression)) has a special meaning, and yields an lvalue expression**

int main() {

int&& a = 42;

decltype(a) b = 42; // Type: rvalue ref to int | VC: lvalue

decltype((a)) c = 73; // Error! Binding non const lvalue ref to prvalue

return 0;

}

### Use Case

1. When the type is unknown (syntax is available from C++14)

template <typename T, typename U>

decltype(auto) Add(T t, U u) {

return t + u;

};

template typename T>

decltype(auto) Wrapper(T&& t) {

// do something...

return std::forward<T>(t);

};

1. To preserve the value category of the expression

int main() {

int a = 42; // Type: rvalue ref to int | VC: lvalue

decltype(a) b = a; // Error! (binding rvalue ref to an lvalue ref a)

decltype(a) c = 73; // Type: rvalue ref to int | VC: lvalue

decltype((a)) d = a; // Type: lvalue ref to int | VC: lvalue

return 0;

}

## [std::move](https://github.com/toRatnesh/Learning_Code_Practice/blob/main/tech_talks/cppcon/cppcon2022/Back_to_Basics-Master_CPP_Value_Categories_With_Standard_Tools-Inbal_Levi/value_category.rst#id23)

std::move(expression);

Utility function, produces an xvalue expression T&&

Equivalent to static\_cast to a T rvalue reference type

static\_cast<typename std::remove\_reference <T>::type&&>(t);

std::move may not always do what you hoped

#include <iostream>

void fun(int& x) {

std::cout << "int& \n";

}

void fun(const int& x) {

std::cout << "const int& \n";

}

void fun(int&& x) {

std::cout << "int&& \n";

}

int main() {

int a = 73;

int b = a;

const int c = a;

const int d = 42;

fun(std::move(b)); // int&& -> fun(int&&)

fun(std::move(c)); // const int& -> fun(const int&)

fun(std::move(d)); // const int&& -> fun(const int&)

return 0;

}

Output:

int&&

const int&

const int&

## [std::forward](https://github.com/toRatnesh/Learning_Code_Practice/blob/main/tech_talks/cppcon/cppcon2022/Back_to_Basics-Master_CPP_Value_Categories_With_Standard_Tools-Inbal_Levi/value_category.rst#id24)

std::forward<T>(expression);

Suggested utility function, **preserves value category of the object passed to the template**

std::forward uses std::remove\_reference<T> to get the value type

Commonly used combined with forwarding reference

#include <iostream>

void fun(int& x) {

std::cout << "int& \n";

}

void fun(const int& x) {

std::cout << "const int& \n";

}

void fun(int&& x) {

std::cout << "int&& \n";

}

template<typename T>

void forward\_wrapper(T&& t) {

fun(std::forward<T>(t));

}

template<typename T>

void no\_forward\_wrapper(T&& t) {

fun(t);

}

int main() {

int a = 73;

const int & cref = a;

forward\_wrapper(a); // int&

no\_forward\_wrapper(a); // int&

forward\_wrapper(cref); // const int&

no\_forward\_wrapper(cref); // const int&

forward\_wrapper(5); // int&&

no\_forward\_wrapper(5); // int&

}

Output:

int&

int&

const int&

const int&

int&&

int&

## [std::decay](https://github.com/toRatnesh/Learning_Code_Practice/blob/main/tech_talks/cppcon/cppcon2022/Back_to_Basics-Master_CPP_Value_Categories_With_Standard_Tools-Inbal_Levi/value_category.rst#id25)

std::decay<T>::type;

std::decay\_t<T>;

Performs the following conversions:

1. Array to pointer
2. Function to function pointer
3. lvalue to rvalue (removes cv qualifiers, references) (issue for move only types)

It resembles "auto"s behavior ("auto" performs auto decay)

#include <iostream>

template<typename T, typename U>

struct decay\_is\_same

: std::is\_same<typename std::decay<T>::type, U>

{};

int main() {

std::cout << std::boolalpha;

std::cout << "decay\_is\_same<int&, int >::value "

<< decay\_is\_same<int&, int >::value << '\n'; // true

std::cout << "decay\_is\_same<const int&, int >::value "

<< decay\_is\_same<const int&, int >::value << '\n'; // true

std::cout << "decay\_is\_same<int&&, int >::value "

<< decay\_is\_same<int&&, int >::value << '\n'; // true

std::cout << "decay\_is\_same<const int&&, int >::value "

<< decay\_is\_same<const int&&, int >::value << '\n'; // true

return 0;

}

Output:

decay\_is\_same<int&, int >::value true

decay\_is\_same<const int&, int >::value true

decay\_is\_same<int&&, int >::value true

decay\_is\_same<const int&&, int >::value true

## [std::remove\_reference](https://github.com/toRatnesh/Learning_Code_Practice/blob/main/tech_talks/cppcon/cppcon2022/Back_to_Basics-Master_CPP_Value_Categories_With_Standard_Tools-Inbal_Levi/value_category.rst#id26)

std::remove\_reference<T>::type;

std::remove\_reference\_t<T>;

*If the type T is a reference type,*

provides the member typedef type which is the type referred to by T.

Otherwise

type is T.

#include <iostream>

#include <type\_traits>

int main() {

std::cout << std::boolalpha;

std::cout << "std::is\_same\_v<int, std::remove\_reference\_t<int>> "

<< std::is\_same\_v<int, std::remove\_reference\_t<int>> << '\n';

std::cout << "std::is\_same\_v<int, std::remove\_reference\_t<int&>> "

<< std::is\_same\_v<int, std::remove\_reference\_t<int&>> << '\n';

std::cout << "std::is\_same\_v<int, std::remove\_reference\_t<int&&>> "

<< std::is\_same\_v<int, std::remove\_reference\_t<int&&>> << '\n';

std::cout << "std::is\_same\_v<const int, std::remove\_reference\_t<int>> "

<< std::is\_same\_v<const int, std::remove\_reference\_t<const int&>> << '\n';

}

Output:

std::is\_same\_v<int, std::remove\_reference\_t<int>> true

std::is\_same\_v<int, std::remove\_reference\_t<int&>> true

std::is\_same\_v<int, std::remove\_reference\_t<int&&>> true

std::is\_same\_v<const int, std::remove\_reference\_t<int>> true

## [std::declval](https://github.com/toRatnesh/Learning_Code_Practice/blob/main/tech_talks/cppcon/cppcon2022/Back_to_Basics-Master_CPP_Value_Categories_With_Standard_Tools-Inbal_Levi/value_category.rst" \l "id27)

std::declval<T>;

Utility function, produces:

* xvalue expression T&&
* If T is void, returns T

Can be used with expression to return the expression's reference type

**Can return a non constructible or incomplete type**

#include <iostream>

struct Type {

int a;

int Foo() { return 42; }

private:

Type() {}

};

int main() {

Type t; // CE

typeid(std::declval<Type>()).name(); // type

std::cout << "typeid(std::declval<Type>()).name() "

<< typeid(std::declval<Type>()).name() << '\n';

return 0;

}

Output:

(after commention 'Type t;' compilation error)

typeid(std::declval<Type>()).name() 4Type

// NOTE: printed type is mangled name

Combined with decltype, can get the type of a member (even when Type is non constructible)

decltype(declval<Type>().a) b = 73;

Shouldn’t be used in an evaluated context (evaluating std::decltype is an error)

**std::declval allows us to access T members, in a way preserves value categories**

#include <utility>

struct Type {

int a;

int ra = a;

int getA() { return int(73); }

int getRefA() { return ra; }

private:

Type(int i) : a(int (i)) {}

};

int main() {

std::declval<Type>().a; // xvalue

std::declval<Type>().ra; // lvalue

std::declval<Type>().getA(); // prvalue

std::declval<Type>().getRefA(); // lvalue

}

decltype and declval are often used to transform between type and instance, for example:

template<typename Derived>

using begin\_adaptor\_t = detail::decay\_t<(decltype(

range\_access::begin\_adaptor(std::declval<Derived &>()))>;

## [std::is\_same](https://github.com/toRatnesh/Learning_Code_Practice/blob/main/tech_talks/cppcon/cppcon2022/Back_to_Basics-Master_CPP_Value_Categories_With_Standard_Tools-Inbal_Levi/value_category.rst" \l "id28)

std::is\_same<T, U>::value;

std::is\_same\_v<T, U>;

Can be used to evaluate equality, including the value category of the expression

#include <iostream>

template <typename T, typename U>

struct remove\_reference\_is\_same

: std::is\_same<typename std::remove\_reference\_t<T>, U>

{ };

int main() {

std::cout << std::boolalpha;

int a = 7;

std::cout << "std::is\_same\_v<int, int> "

<< std::is\_same\_v<int, int> << '\n'; // true

std::cout << "std::is\_same\_v<int&, int> "

<< std::is\_same\_v<int&, int> << '\n'; // false

std::cout << "std::is\_same\_v<decltype(a), int> "

<< std::is\_same\_v<decltype(a), int> << '\n'; // true

std::cout << '\n';

std::cout << "remove\_reference\_is\_same<int&, int >::value "

<< remove\_reference\_is\_same<int&, int>::value << '\n'; // true

std::cout << "remove\_reference\_is\_same<const int&, int >::value "

<< remove\_reference\_is\_same<const int&, int>::value << '\n'; // false

std::cout << "remove\_reference\_is\_same<const int&, const int >::value "

<< remove\_reference\_is\_same<const int&, const int>::value << '\n'; // true

std::cout << '\n';

std::cout << "remove\_reference\_is\_same<int&&, int >::value "

<< remove\_reference\_is\_same<int&&, int>::value << '\n'; // true

std::cout << "remove\_reference\_is\_same<const int&&, int >::value "

<< remove\_reference\_is\_same<const int&&, int>::value << '\n'; // false

std::cout << "remove\_reference\_is\_same<const int&&, const int >::value "

<< remove\_reference\_is\_same<const int&&, const int>::value << '\n'; // true

return 0;

}

Output:

std::is\_same\_v<int, int> true

std::is\_same\_v<int&, int> false

std::is\_same\_v<decltype(a), int> true

remove\_reference\_is\_same<int&, int >::value true

remove\_reference\_is\_same<const int&, int >::value false

remove\_reference\_is\_same<const int&, const int >::value true

remove\_reference\_is\_same<int&&, int >::value true

remove\_reference\_is\_same<const int&&, int >::value false

remove\_reference\_is\_same<const int&&, const int >::value true

# [Advanced Utilities](https://github.com/toRatnesh/Learning_Code_Practice/blob/main/tech_talks/cppcon/cppcon2022/Back_to_Basics-Master_CPP_Value_Categories_With_Standard_Tools-Inbal_Levi/value_category.rst" \l "id29)

* std::reference\_wrapper
* std::ref, std::cref
* std::unwrap\_reference (since C++20)
* Concepts (since C++20)
* auto cast (since C++23)
* Deducing this (since C++23)

## [std::reference\_wrapper](https://github.com/toRatnesh/Learning_Code_Practice/blob/main/tech_talks/cppcon/cppcon2022/Back_to_Basics-Master_CPP_Value_Categories_With_Standard_Tools-Inbal_Levi/value_category.rst" \l "id30)

std::reference\_wrapper<T>;

Functional helper, produces lvalue expression which can be used in same places as a reference

std::reference\_wrapper provides extra safety of avoiding dangling reference

can be used to create a container of objects of non-constructible or incomplete types

std::ref and std::cref are often used to generate std::reference\_wrapper objects

std::reference\_wrapper is also used to pass objects by reference

#include <iostream>

#include <type\_traits>

#include <array>

#include <vector>

class Type {

int a;

Type(int val) : a(val) {}

public:

friend Type getType(int i);

friend std::ostream & operator<<(std::ostream & os, const Type & t) {

std::cout << t.a;

return os;

}

};

Type getType(int i) {

return Type(i);

}

int main() {

std::array larr = {getType(1), getType(2), getType(3), getType(4), getType(5)};

std::vector<std::reference\_wrapper<Type>> vec(larr.begin(), larr.end());

std::cout << "vec.at(0): " << vec.at(0) << '\n';

std::cout << "vec.at(4): " << vec.at(4) << '\n';

}

Output:

vec.at(0): 1

vec.at(4): 5

## [std::ref, std::cref](https://github.com/toRatnesh/Learning_Code_Practice/blob/main/tech_talks/cppcon/cppcon2022/Back_to_Basics-Master_CPP_Value_Categories_With_Standard_Tools-Inbal_Levi/value_category.rst" \l "id31)

Function helper, takes an lvalue reference to object (overload (const) rvalue reference is deleted)

Produces an object of type std::reference\_type

**Since C++20, T may be incomplete type**

#include <iostream>

#include <type\_traits>

#include <array>

#include <vector>

class Type {

int a;

Type(int val) : a(val) {}

public:

friend Type getType(int i);

friend std::ostream & operator<<(std::ostream & os, const Type & t) {

std::cout << t.a;

return os;

}

};

Type getType(int i) {

return Type(i);

}

template<typename T>

void require\_ref(T t) {

std::cout << "Inside require\_ref: " << t << '\n';

}

int main() {

std::array larr = {getType(1), getType(2), getType(3), getType(4), getType(5)};

std::vector<std::reference\_wrapper<Type>> vec;

vec.push\_back(std::ref(larr.at(0)));

vec.push\_back(std::ref(larr.at(1)));

vec.push\_back(std::ref(larr.at(2)));

vec.push\_back(std::ref(larr.at(3)));

vec.push\_back(std::ref(larr.at(4)));

std::cout << "vec.at(0): " << vec.at(0) << '\n';

std::cout << "vec.at(4): " << vec.at(4) << '\n';

int a = 15;

require\_ref(a);

require\_ref(std::ref(a));

require\_ref(std::cref(a));

}

Output:

vec.at(0): 1

vec.at(4): 5

Inside require\_ref: 15

Inside require\_ref: 15

Inside require\_ref: 15

## [std::unwrap\_reference (since C++20)](https://github.com/toRatnesh/Learning_Code_Practice/blob/main/tech_talks/cppcon/cppcon2022/Back_to_Basics-Master_CPP_Value_Categories_With_Standard_Tools-Inbal_Levi/value_category.rst" \l "id32)

std::unwrap\_reference<T>::type;

std::unwrap\_reference\_t<T>;

Produces Type

*If T is std::reference\_wrapper<U> for some type U,*

type is U&;

*otherwise,*

type is T

std::unwrap\_ref\_decay<T>::type;

std::unwrap\_ref\_decay<T>;

*If T is std::reference\_wrapper<U> (possibly CV qualified) for some type U,*

type is U&;

*otherwise,*

type is std::decay\_t<T>

## [Concepts (since C++20)](https://github.com/toRatnesh/Learning_Code_Practice/blob/main/tech_talks/cppcon/cppcon2022/Back_to_Basics-Master_CPP_Value_Categories_With_Standard_Tools-Inbal_Levi/value_category.rst" \l "id33)

template<Concept T>

auto fun(T t);

Concepts can be used to constraint the value category of the entity acceptable in an algorithm

Example Library Concepts:

**std::move\_constructible**

Concept is satisfied if:

* T is a reference type
* T is an object type which can be constructed from a T rvalue

**std::movable**

Concept is satisfied if:

T is an object, which also satisfies

std::move\_constructible<T>

assignable\_from<T& T> (using std::is\_lvalue\_reference, std::forward etc.)

std::swappable<T>

These as well as the <type\_traits> are available for constructing our own concepts

## [auto cast (since C++23)](https://github.com/toRatnesh/Learning_Code_Practice/blob/main/tech_talks/cppcon/cppcon2022/Back_to_Basics-Master_CPP_Value_Categories_With_Standard_Tools-Inbal_Levi/value_category.rst" \l "id34)

Result is a prvalue of an object type(discards CV qualifier)

can also be used combined with "decltype"

int a = 42; // lvalue (int)

auto & b = a; // lvalue (int&)

auto c(b); // lvalue (int&)

auto(c); // prvalue

## [Deducing this (since C++23)](https://github.com/toRatnesh/Learning_Code_Practice/blob/main/tech_talks/cppcon/cppcon2022/Back_to_Basics-Master_CPP_Value_Categories_With_Standard_Tools-Inbal_Levi/value_category.rst" \l "id35)

tempplate<typename T>

auto fun(this T&& t);

Allows specifying from within a member function the value category of the expression it’s invoked on

struct Type {

// pre C++23

auto fun() const &;

auto fun() &;

auto fun() &&;

// since C++23

auto fun(this const Type&);

auto fun(this Type&);

auto fun(this Type&&);

};

Combined with the forwarding reference, we can now write all these in a single template function

struct Type {

template<typename Self>

auto fun(this Self self);

};

"Deducing this" feature introduced two new utilities: 'like\_t' and 'forward\_like <T>(u)'

**like\_t <T,U>**

Applies CV and ref qualifiers of T onto U (introduced in [P0847](https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p0847r6.html))

like\_t<double&, int> // int

like\_t<const double&, int> // const int&

like\_t<double &&, int> // int&&

like\_t<const double &&, int> // const int

forward\_like<T>(u) -> forward<like\_t<T, decltype(u)>>(u)

Forwards instance of type U with CV and ref qualifiers of T (introduced in [P2445](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2445r0.pdf))

int a = 5;

forward\_like<double &>(a); // int&

forward\_like<const double &&>(a); // const int&

# [References](https://github.com/toRatnesh/Learning_Code_Practice/blob/main/tech_talks/cppcon/cppcon2022/Back_to_Basics-Master_CPP_Value_Categories_With_Standard_Tools-Inbal_Levi/value_category.rst" \l "id36)

[Back to Basics: Master C++ Value Categories with Standard Tools - Inbal Levi - CppCon 2022](https://www.youtube.com/watch?v=tH0Z2OvHAd8)

[Master Value Categories with Standard Tools - Inbal Levi - CppNow 2022](https://www.youtube.com/watch?v=D52fZsQ9j4o)