Back To Basics
Value Categories

INBAL LEVI

Cppcon
The C++ Conference

2022
September 12th-16th

Who Am I?

- A C++ Developer, working at SolarEdge
- Active member of the ISO C++ work group (WG21):
  - Israeli NB Chair
  - Ranges SG Chair
  - Assistant LEWG Chair

- I love software design
- I also have a passion for cataloguing

Outline

- **Part 0:** Motivation
- **Part I:** Intro to value categories
- **Part II**: Value categories in practice
- **Part III**: Value categories In generic code
- **Part IV**: Tools for handling value categories

- **Disclaimers:**
  - I "cut corners", full definitions are in cppreference.com and in the standard
  - Examples will elegantly ignore function pointers and arrays, and focus on objects

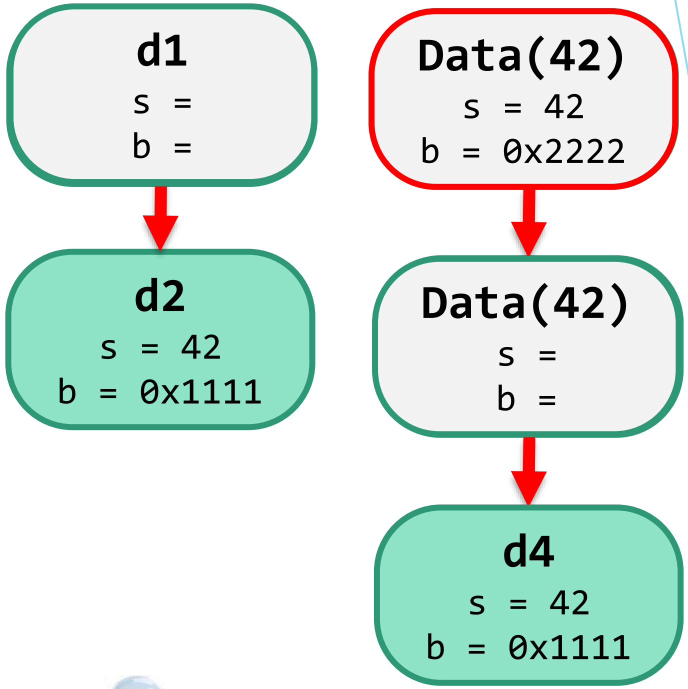
Part 0: Motivation

- Consider the following code:

```
struct Data {
   Data(size_t s);          // Constructor
   Data(const Data&);       // Copy Constructor
   Data(Data&&);            // Move Constructor

   size_t s;
   int* b;
};

const Data getData(size_t s) {
   return Data(s);
}
```

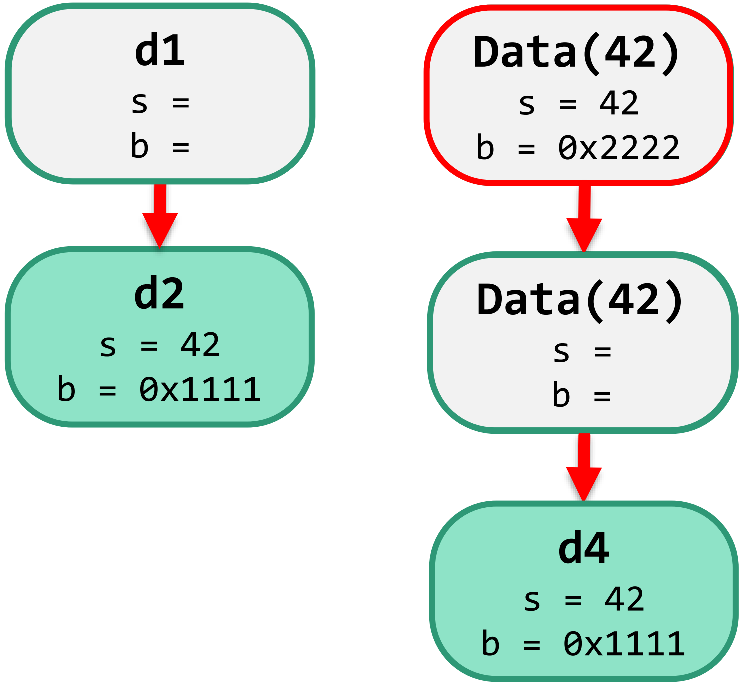
```
1 auto d1 = Data(42);                      // 1
2 auto d2 = std::move(d1);                 // 0

3 auto d3 = getData(42);                   // 1
4 auto d4 = std::move(getData(42));        // 2 ☹
```

1



Part 0: Motivation

- This is (roughly) how your compiler see your program!
- You can use std::move to **explicitly** color objects in “Gray” (= temporary)
- Some conditions can mark objects in Gray **implicitly**
- “Gray” tells your compiler that it can “steal” from the object



Outline

- **Part 0:** Motivation
- **Part I:** Intro to value categories
- **Part II**: Value categories in practice
- **Part III**: Value categories In generic code
- **Part IV**: Tools for handling value categories

Part I: Intro To Value Categories

- What are value categories
- Evolution of value categories

Part I: What Are Value Categories

- Value Categories were inherited from C, with the porting of “lvalue expression”

- Originally referred to the location of expression with regards to assignment:

```cpp
auto a = int(42);
```

  - lvalue (left-value) was on the left of the assignment
  - rvalue (right-value) was on the right of the assignment

Part I: What Are Value Categories

```
auto a = int(42);
```

- Value Category of an entity defines:
  1. Its **lifetime**:
     - Can it be moved from
     - Is it a temporary
     - Is it observable after change, etc.
  2. Its **identity:**
     - Object has identity if its address can be taken and used safely

- Value Categories affect two very important aspects:
  1. Performance
  2. Overload resolution

Part I: (Detour) Terminology of References

- A brief reminder of References' terminology

```
struct Data {
   Data(int x);
   int x_;
};
```

```
Data a = 42;
Data& lval_ref_a = &a;  // lvalue ref
Data&& rval_ref_a = &a; // rvalue ref (Fail!)
Data&& rval_ref_a = 42; // rvalue ref (OK)
```

Part I: What Are Value Categories

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

```
struct Data {
    Data(int x);
    int x_;
};
void foo(Data&& x) {
    x = 42;
}
```

```
Data&& a = 42;
foo(a);              // Fail! lvalue!
foo(Data(73));       // OK
```

- What (misleadingly) looks like the value category, **can in fact be the type**
  - **a's Type:** *rvalue reference* to Data
  - **a's Value Category:** lvalue

Part I: What Are Value Categories

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

```
struct Data {
   Data(int x);
   int x_;
};
void foo(Data&& x) {
   x = 42;
}
```

```
Data&& a = 42;
foo(a);              // Fail! lvalue!
foo(Data(73));       // OK
```

- 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(73)`
  - **Step II**: `Data(73)` binds to the *rvalue reference* x
  - **Step III**: The entity which *used to be* `Data(73)` has a name - x, therefore, in the scope of foo, x is now an lvalue

Part I: What Are Value Categories

- Each **expression** has two properties:
  1. A type (including CV qualifiers)
  2. A value category
- Value Category is a quality of an **expression**

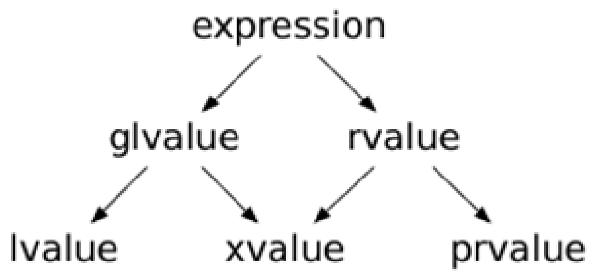


Figure 1 — Expression category taxonomy (*)

(*) Introduced in: N3055: A Taxonomy of Expression Value Categories (William M. Miller, 2010)

- VCs changed dramatically throughout the lifetime of C++ versions, affected by the rules defining **references**, **move semantics** and **copy elision**.

Part I: Evolution Of Value Categories

- **C language**:
  - Three types of expressions:
    - lvalue expression
    - Non-lvalue object expression
    - (Function designator expression)

- **C++98** [ISO/IEV 14882:1998]: 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** [ISO/IEC 14882:2003]
  - No significant change

Part I: Evolution Of Value Categories

- **C++11** [N3242]: added rvalue references, move semantics:
  - By: Howard E. Hinnant, Peter Dimov, Dave Abrahams
  - N1377: A Proposal to Add Move Semantics Support to the C++ Language (2002)
  - N1385: The Forwarding Problem: Arguments (2002)
  - N2118: A Proposal to Add an Rvalue Reference to the C++ Language (2006)

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

- **C++17** [N4659]: Added guaranteed copy elision:
  - P0135: Guaranteed copy elision through simplified value categories (Richard Smith)

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

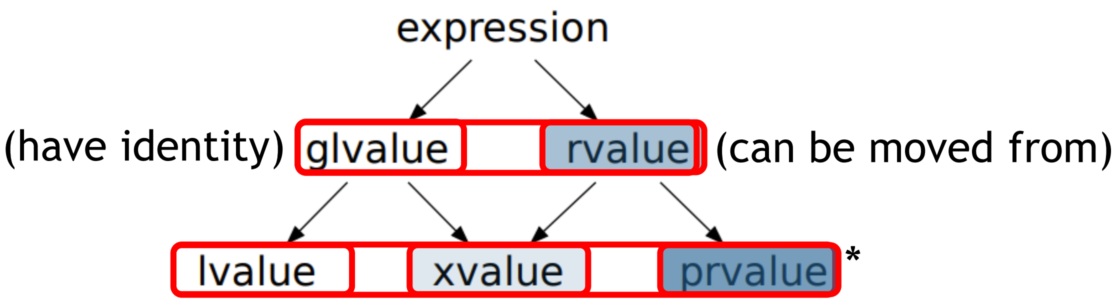
"The ***result of a prvalue*** is the value that the expression stores into its context"

Part I: Evolution Of Value Categories

- **C++20:**
  - [N4861] (March 2020):
    - P0527: 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

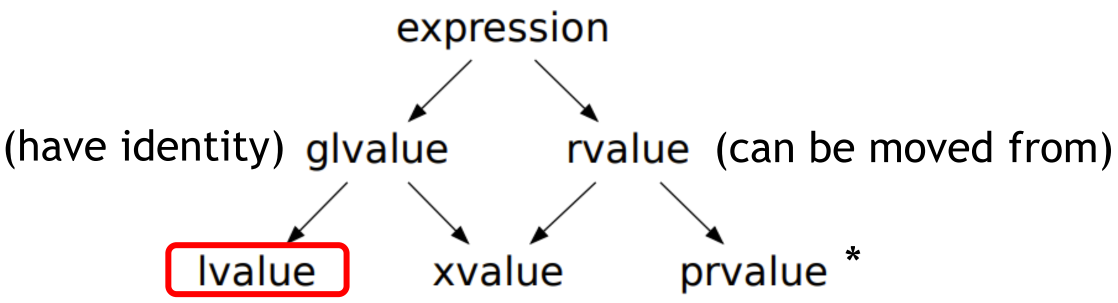
- **C++23 draft (latest):**
  - P0847: Deducing this (Gašper Ažman, Sy Brand, Ben Deane, Barry Revzin)
  - P0847 also added like_t
  - P2445: std::forward_like

Part I: Evolution Of Value Categories



- 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

Part I: Evolution Of Value Categories



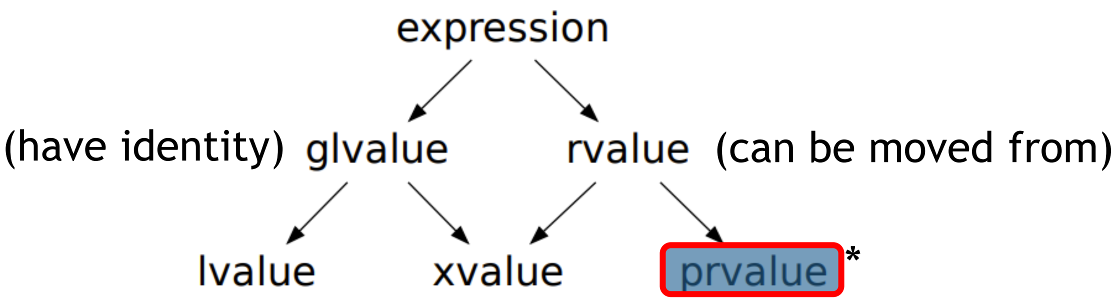
- Examples:

```
struct Data { int n; int* pn = &n; };
Data& getData(Data& d) { return d }
int a = 42;
int b = a;
int& la = a;    // ra = a = b;
int* pa = &a;
int&& ra = 42;
a++;
++a;
```

```
int arr[] = {1, 2, 3};
arr[0] = 73;
Data d;
(&d)->n = 42;
d.n = 73;
*d.pn = 42;
string s = "Hello World";
a==b ? b : c; // when b, c are lvalues
Data c = getData(d);
```

(*) ra has the **type: rvalue reference to int**, with the **value category: lvalue**

Part I: Evolution Of Value Categories



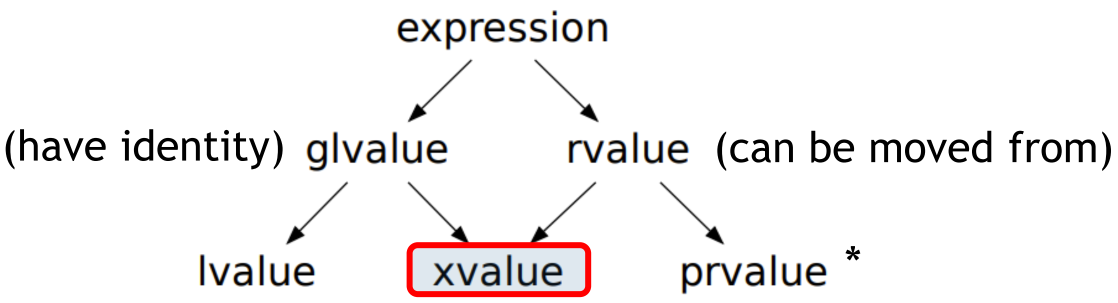
- Examples:

```
struct Data {
    int n;
    int foo() { this->n = 4; }
};
int a = 42;
int* pa = &a;
pa = nullptr;
a++;
++a;
```

```
auto l = []() { return 2; };
Data d;
Data* dp = &d;
Data();
d->n = 6;
d.n = 6;
string s = "Hello World";
a==a ? throw 4 : throw 2;    // void
bool equals = a==42;
```

(*) **Built-in** post increment

Part I: Evolution Of Value Categories



- Examples:

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


Data d1 = Data(42);
d1.*pn = 73;
Data d2 = std::move(d1);
Data().n;
```

```
Data getData() {
    return Data(73);
}

Data d3 = getData();
d1==d2 ? Data(42) : Data(73);
```

Outline

- **Part 0:** Motivation
- **Part I:** Intro to value categories
- **Part II**: Value categories in practice
- **Part III**: Value categories In generic code
- **Part IV**: Tools for handling value categories

Part II: Value Categories In Practice

- The details of binding
- Copy elision optimizations

Part II: 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

```
int a = 42;
int& la = a;
const int& cla = a;
int&& ra = a + 73;
const int&& cra = 42;
```

```
la = 73;        // OK, a = la = 73
cla = 42;       // Error!
ra = 73;        // OK
cra = 42;       // Error!
```

- Binding rules are important as part of the following "events":
  1. Initialization or assignment
  2. Function call (including non-static class member function called on an object)
  3. Return statement

Part II: The Details Of Binding

1. Initialization Or Assignment:

```
int a = 42;
int& la1 = a;                   // OK, la = 42
int& la2 = 73;                  // Error
const int& cla1 = a;            // OK
const int& cla2 = 73;           // OK
int&& ra1 = a;                  // Error
int&& ra2 = a + 42;             // OK
const int&& cra1 = a;           // Error
const int&& cra2 = a + 42;      // OK
```

1

	Binds lvalues?	Binds rvalues?
lvalue reference	V	X
Const lvalue reference	V	V
rvalue reference	X	V
Const rvalue reference	X	V

- 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

Part II: The Details Of Binding

2. Function Call:

```
struct Data {
   Data(int n) : _n(n) {}
   int _n;
};
const Data getData(int x) {
   return Data(x);
}
void foo(Data& x) {}          // 1
void foo(const Data& x) {}    // 2
void foo(Data &&x) {}         // 3
void foo(const Data &&x) {} // 4
```

```
Data d = 42;
Data &lval_ref_d = d;
const Data& c_lval_ref_d = d;
Data&& rval_ref_d = Data(73);
const Data&& c_rval_ref_d = Data(42);
foo(lval_ref_d);              // lvalue: 1,2
foo(c_lval_ref_d);            // const lvalue: 2
foo(rval_ref_d);              // lvalue: 1,2
foo(c_rval_ref_d);            // const lvalue: 2
foo(Data(73));                // xvalue: 3,4,2
foo(getData(42));             // const xvalue: 4,2(!)
```

1

(*)
Example scheme by Boris Kolpackov

- Limitations in the context of the function are according to the **binding function**:

	Function can modify data?	Caller can observe (old) data?
lvalue reference	V	V
Const lvalue reference	X	V
rvalue reference	V	X
Const rvalue reference	X	X

2

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

Part II: Copy Elision Optimizations

3. Return Statement:
   - Starting from C++17, the behavior of VCs is affected by: "P0135: Guaranteed copy elision (…)"
   - There are two mandatory elisions of copy and move constructors:
     1. Object initialization

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

1

     2. Return statement:

        An un-named Return Value Optimization (RVO):

```
Data getData(int x){
   return Data(x);
}
```

```
Data d = getData(42); // 1 CTOR (avoids: CTOR, Move CTOR)
```

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

Part II: Copy Elision Optimizations

3. Return Statement: **Materialization**

Temporary materialization conversion [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.

Part II: The Details Of Binding

- To summarize:
  Binding rules apply in the following “events”:
  1. Initialization or assignment
  2. Function call (including non-static class member function called on an object)
  3. Return statement

- Behavior of the entity:
  1. Initialization: limits are according to the reference which binds it
  2. Function call: limits inside the function are according to the overload which binds it
  3. Return statement: limits as in initialization, with additional rules due to optimizations and const

Outline

- **Part 0:** Motivation
- **Part I:** Intro to value categories
- **Part II**: Value categories in practice
- ➡ **Part III**: Value categories In generic code
- **Part IV**: Tools for handling value categories

Part III: Value Categories In Generic Code

- Reference collision
- Forwarding reference

Part III: Reference Collision

- In case of concatenation of multiple ‘&’ symbols:
  - In generic code
  - In code using type aliases

- Compiler performs Reference Collision:

```
typedef int&  lr;
typedef int&& rr;
```

```
int a;
int&& b = a;        // int&
int&&& c = a;       // int&
int&&& d = a;       // int&
int&&&& e = 73;     // int&&
```

Part III: 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/3])

- Due to TAD, “rvalue reference” has a special meaning in context of function template:

```
Template <typename T>
void foo(T&& t) {
   // Type of T here
}
```

```
int a = 42;
const int& cla = a;
int&& b = 73;
foo(a);               // int       &&  T=int&
foo(cla);             // const int&&  T=const int&
foo(std::move(a));    // int       &&  T=int
```

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

Outline

- **Part 0:** Motivation
- **Part I:** Intro to value categories
- **Part II**: Value categories in practice
- **Part III**: Value categories In generic code
- **Part IV**: Tools for handling value categories

Part IV: Tools for handling value categories

- std::move
- std::forward
- std::decay
- decltype specifier
- std::declval
- Deducing this (C++23)

Part IV: (1): std::move

```
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)
```

- Notice that std::move may not always do what you hoped:

```
void foo(int& x) {
    cout << "int&";
}
void foo(const int& x) {
    cout << "const int&";
}
void foo(int &&x) {
    cout << "int&&";
}
```

```
int a = 73;
int &b = a;
const int& c = a;
const int&& d = 42;
foo(std::move(b));    // int&        -> foo(int&&)
foo(std::move(c));    // const int&  -> foo(const int&)
foo(std::move(d));    // const int&& -> foo(const int&)
```

1

2

Part IV: (2): std::forward

```
std::forward<T>( expression );
```

- N1385: The forwarding problem: Arguments (Peter Dimov, Howard E. Hinnant, Dave Abrahams) (2002)
  Presented two issues: forwarding params, and returning result
- 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

```cpp
void Foo(int& x) {
    cout << "int&";
}
void Foo(const int& x) {
    cout << "const int&";
}
void Foo(int &&x) {
    cout << "int&&";
}
```

```cpp
template<class T>
void Wrapper(T&& t) {
   Foo(std::forward<T>(t));
}

template<class T>
void NFWrapper(T&& t) {
   Foo(t);
}
```

```cpp
int a = 73;
const int& lca = a;
Wrapper(a);          // int&
NFWrapper(a);        // int&
Wrapper(lca);        // const int&
NFWrapper(lca);      // const int&
Wrapper(6);          // int&&
NFWrapper(6);        // int&
```

1

Part IV: (3): std::decay

```
std::decay<T>::type
```

- Type trait, result is accessible through `_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)

```cpp
template <typename T, typename U>
struct decay_is_same :
    std::is_same<typename std::decay<T>::type, U>
{};
```

1

```cpp
decay_is_same<int&, int>::value;        // True(1)
```

- This behavior should be familiar to you, as it resembles “auto”s behavior (“auto” performs auto-decay)

Part IV: (4): decltype specifier

```
decltype( expression )
```

- Evaluates an *expression*, yields its **type** + **value category** (AKA the declared type)
- decltype (unlike auto) preserves value category. For an *expression* of type T:
  - If *expression* is xvalue, yields T&&
  - If *expression* is lvalue, yields T&
  - If *expression* is prvalue, yields T
- Can be used instead of a type, as a placeholder which preserves value categories

```
int&& foo(int& i) {
    return std::move(i);
}
```

```
int i = 73;
auto a = foo(i);             // Type: int        | VC: lvalue
decltype(auto) b = foo(i);   // Type: rvalue ref | VC: lvalue
```

1

(*)

(*) Utility evaluating VC in example is based on Barry Revzin's blog: Value Categories in C++17

Part IV: (4): decltype specifier

```
decltype( expression )
```

- 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&& 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
```

Part IV: (4): decltype specifier

```
decltype( expression )
```

- To summarize:
  decltype() main use cases:
  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&& 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
```

Part IV: (5): std::declval

```
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

```
struct Type {
   int a;
   int Foo() { return 42; }
private:
   Type(){}
};
```

```
Type t;                                  // Fails
typeid(std::declval<Type>()).name();     // Type
```

1

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

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

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

Part IV: (5): std::declval

```
std::declval<T>( )
```

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

```
struct Type {
   int a;
   int &ra = a;
   int getA() { return int(73); };
   int& getRefA() { return ra; };
private:
   Type(int i) : a(int(i)) {}
};
```

```
std::declval<Type>().a;          // xvalue
std::declval<Type>().ra;         // lvalue
std::declval<Type>().getA();     // prvalue
std::declval<Type>().getRefA();  // lvalue
```

1

2

- 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 &>()))>;
```

Part IV: (6): Deducing This (C++23)

```
template <typename T>
void Foo(this T&& t) {}
```

- P0847: Deducing this (Gasper Azman, Sy Brand, Ben Deane, Barry Revzin) - voted into C++23
- Allows specifying from within a member function the value category of the expression it's invoked on

```
struct Type {
    auto Foo() const &;
    auto Foo() &;
    auto Foo() &&;
};
```

```
struct Type {
    auto Foo(this const Type&);
    auto Foo(this Type&);
    auto Foo(this Type&&);
};
```

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

```
struct Type {
    template <typename Self>
    auto Foo(this Self&& self);
};
```

Part IV: (6): Deducing This (C++23)

```
template <typename T>
void Foo(this T&& t) {}
```

- “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)

```
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)

```
int a = 5;
forward_like<double&>(a)          // int&
forward_like<const double&>(a)    // const int&
```

Summary

- **Part I**: What are value categories
  - What are value categories
  - Evolution of value categories
- **Part II**: Value categories in practice
  - The details of binding
  - Copy elision optimizations
- **Part III**: Value categories in generic code
  - Reference collision
  - Forwarding reference
- **Part IV**: Manipulating value categories
  - std::move
  - std::forward
  - std::decay
  - decltype specifier
  - std::declval
  - Deducing This (C++23)

- **More Tools (not in this talk):**
  - Type Traits:
    - std::is_lvalue_reference<T>
    - std::is_rvalue_reference<T>
    - std::is_same
    - std::common_type
    - std::common_reference (C++20)
    - std::common_reference_with (C++20)
  - Library Utils:
    - std::remove_reference
    - std::reference_wrapper
    - std::ref
    - std::unwrap_reference (C++20)
    - Library provided Concepts (C++20)
  - Language Utils:
    - auto cast (C++23)
    - User defined Concepts (C++20)
  - …

Thanks!

Thank you for listening ☺

Special thanks to:

→ Keren Or Curtis
→ Dvir Yizhaki
→ Andrei Zissu

- Linkedin.com/inballevi
- Twitter.com/Inbal_l
- sinbal2lextra@gmail.com

Would love to get your input!

References (☺)

- Proposals:
  - P0849: auto(x): decay-copy in the language (Zhihao Yuan) (C++23)
  - P2255: A type trait to detect reference binding to temporary (Tim Song) (C++23)
  - P2446: views::as_rvalue (C++23?)
  - P0792: function_ref: a non-owning reference to a Callable (Romeo, Yuan, Waterloo) (C++23?)
  - P2266: Simpler implicit move (Arthur O'Dwyer) (C++23?)
  - P1663: Supporting return-value-optimisation in coroutines (Lewis Baker)
  - P1906: Provided operator= return lvalue-ref on rvalue (Peter Sommerlad)
  - P2307: Lvalue closures (Jens Gustedt)
  - P1946: Allow defaulting comparisons by value (Barry Revzin, Casey Carter)
  - P2027: Moved-from objects need not be valid (Geoff Romer)
  - P2226: A function template to move from an object and reset it (…) (Giusepper D'Angelo)
  - P2012: Fix the range-based for loop (Various Authors)

References (☺)

- Books, Talks & Blogs:
  - Book: David Vandevoorde, Nicolai Josuttis, Douglas Gregor: C++ Templates: The Complete Guide
  - Paper: Bjarne Stroustrup: “New” Value Terminology
  - Talk: Ben Deane: Deducing this Patterns (CppCon 2021)
  - Talk: Peter Sommerlad: Reducing immediate dangling? (lightning talk) (CppCon 2019)
  - Talk(s): Walter E.Brown: Modern Template Metaprogramming: A Compendium (CppCon 2014)
  - Blog: Barry Revzin: xvalues and prvalues: The Next Generation (Discussing Identity function safely)
  - Blog: Barry Revzin: Value Categories in C++17
  - Blog: Tristan Brindle: Rvalue Ranges and Views in C++20
  - Blog: Thomas Becker: C++ Rvalue References Explained
  - StackOverflow: Do rvalue references to const have any use
  - Manual: Rvalue references in Chromium

"The best number is 73. Why? 73 is the 21st prime number. Its mirror, 37, is the 12th and its mirror, 21, is the product of multiplying 7 and 3."
"We get it, 73 is the Chuck Norris of numbers!"
"Chuck Norris wishes. In binary 73 is a palindrome, 1001001, which backwards is 1001001. All Chuck Norris backwards gets you is Sirron Kcuhc!"