Moving to Modern C++:
Rvalue References, Moving, and Perfect Forwarding

Steve Dewhurst
Dan Saks

1

Modern C++

The C++ programming language is defined by a formal international standard specification. That standard was updated in 2011 and again in 2014. Modern C++ is the language as specified by these recent standards.

Compared to the earlier standards, Modern C++ introduces a significant number of new language and library features. This course focuses primarily on the language features of Modern C++ and programming techniques that use those features.

2

Saks & Associates
393 Leander Dr.
Springfield, OH 45504-4906 USA
+1-937-324-3601
dan@dansaks.com
www.dansaks.com

3

4

Legal Stuff

- These notes are Copyright © 2017 by Stephen C. Dewhurst and Daniel Saks.
- If you have attended this course, then:
  - You may make copies of these notes for your personal use, as well as backup copies as needed to protect against loss.
  - You must preserve the copyright notices in each copy of the notes that you make.
  - You must treat all copies of the notes — electronic and printed — as a single book. That is,
    - You may lend a copy to another person, as long as only one person at a time (including you) uses any of your copies.
    - You may transfer ownership of your copies to another person, as long as you destroy all copies you do not transfer.

5

More Legal Stuff

- If you have not attended this course, you may possess these notes provided you acquired them directly from Saks & Associates, or:
  - You have acquired them, either directly or indirectly, from someone who has (1) attended the course, or (2) paid to attend it at a conference, or (3) licensed the material from Saks & Associates.
  - The person from whom you acquired the notes no longer possesses any copies.
- If you would like permission to make additional copies of these notes, contact Saks & Associates.

6

About Steve Dewhurst

Steve Dewhurst is the cofounder and president of Semantics Consulting, Inc. He is the author of the critically-acclaimed books *C++ Common Knowledge* and *C++ Gotchas*, and the co-author of *Programming in C++*. He has written numerous technical articles on C++ programming techniques and compiler design.

As a Member of Technical Staff at AT&T Bell Laboratories, Steve worked with C++ designer Bjarne Stroustrup on the first public release of the C++ language and cfront compiler. He was lead designer and implementer of AT&T's first non-cfront C++ compiler. As a compiler architect at Glockenspiel, Ltd., he designed and implemented a second C++ compiler. He has also written C, COBOL, and Pascal compilers.

Steve served on both the ANSI/ISO C++ standardization committee and the ANSI/IEEE Pascal standardization committee.

7

About Steve Dewhurst

Steve has consulted for projects in areas such as compiler design, embedded telecommunications, e-commerce, and derivative securities trading. He has been a frequent and highly-rated speaker at industry conferences such as *Software Development* and *Embedded Systems*. He was a Visiting Scientist at CERT and a Visiting Professor of Computer Science at Jackson State University.

Steve was a contributing editor for *The C/C++ User's Journal*, an editorial board member for *The C++ Report*, and a cofounder and editorial board member of *The C++ Journal*.

Steve received an A.B. in Mathematics and an Sc.B. in Computer Science from Brown University in 1980 and an M.S. in Engineering/ Computer Science from Princeton University in 1982.

8

About Dan Saks

Dan Saks is the president of Saks & Associates, which offers training and consulting in C and C++ and their use in developing embedded systems.

Dan is a contributing editor for *embedded.com* online. He has written columns for numerous print publications including *The C/C++ Users Journal*, *The C++ Report*, *Software Development*, and *Embedded Systems Design*. With Thomas Plum, he wrote *C++ Programming Guidelines*, which won a *1992 Computer Language Magazine Productivity Award*. He has also been a Microsoft MVP.

Dan has taught C and C++ to thousands of programmers around the world. He has presented at conferences such as *Software Development*, *Embedded Systems*, and *C++ World*. He has served on the advisory boards of the *Embedded Systems* and *Software Development* conferences.

9

About Dan Saks

Dan served as secretary of the ANSI and ISO C++ standards committees and as a member of the ANSI C standards committee. More recently, he contributed to the *CERT Secure C Coding Standard* and the *CERT Secure C++ Coding Standard*.

Dan collaborated with Thomas Plum in writing and maintaining *Suite++™, the Plum Hall Validation Suite for C++*, which tests C++ compilers for conformance with the international standard. Previously, he was a Senior Software Engineer for Fischer and Porter (now ABB), where he designed languages and tools for distributed process control. He also worked as a programmer with Sperry Univac (now Unisys).

Dan earned an M.S.E. in Computer Science from the University of Pennsylvania, and a B.S. with Highest Honors in Mathematics/Information Science from Case Western Reserve University.

10

Past C++ Standards

- ***1998***: "C++98"
  - the first international C++ standard (ISO [1998])
- ***2003***: "C++03"
  - a revised international C++ standard (ISO [2003])
  - bug fixes
  - nothing else new
- ***2005***: "TR1"
  - Library "Technical Report 1" (ISO [2005])
  - proposals for library extensions
  - not a new standard

11

Modern C++ Standards

- ***2011***: "C++11"
  - a new international C++ standard (ISO [2011a])
  - significant new language features
  - most of TR1, plus more library components
- ***2014***: "C++14"
  - the latest international C++ standard (ISO [2014])
  - mostly improvements to C++11 features
  - a few new features, too

12

Rvalue References and Moving

13

Swapping Objects

- C++ code often moves the value of one object into another.
- For example, this is what the standard swap function template does.
- The standard swap function template looks something like:

```
template <typename T>
void swap(T &x, T &y) {
    T temp (x);
    x = y;
    y = temp;
}
```

14

Swapping Objects

- For many types, this `swap` function template is as efficient as it gets.
  - Certainly for scalar types.
- However, for class types with expensive copy constructors and copy assignments, this `swap` can be very inefficient.
- For example, an implementation of a string class could store the text of the string in a dynamically-allocated array…

15

Swapping Objects

```
class String {
public:
    String &operator=(String const &);
    ~~~
private:
    char *array;
    std::size_t size;
};

String operator+(String const &s1, String const &s2);
```

- Consider what happens when you `swap` non-empty Strings of different sizes…

16

Swapping Objects

```
String s ("hello");
String t ("goodbye");
~~~
swap(s, t);        // calls swap<String>(s, t)
```

- The values for `s` and `t` already exist.
- They're just not in the right place.
- The call to `swap` does a lot of allocating, copying, and deallocating.
- All of which is wasted effort…

17

Swapping Objects

```
void swap(String &x, String &y) {
    String temp (x);
```
  - *allocate uninitialized storage for `temp`'s array*
  - *construct each element of `temp`'s array by copying from `x`'s array*
```
    x = y;
```
  - *resize `x`'s array (possibly deallocating that array and allocating a new one)*
  - *copy the elements from `y`'s array to `x`'s array*
```
    y = temp;
```
  - *same effort as the previous assignment*
```
}
```
  - *deallocate `temp`'s array (as the function returns)*

18

Swapping Objects

- You could implement an efficient `swap` for `String` as a member function:

```
void String::swap(String &other) {
    std::swap(array, other.array);
    std::swap(size, other.size);
}
```

- This memberwise `swap` is much more efficient:
  - It copies just the `array` pointer, not the contents of the entire String.
- That's why the standard `string` includes its own `swap` member function.

19

Swapping Objects

- We'd like the generic `swap` to take advantage of these more efficient versions when available.
- We could overload `swap` for `String` to use the more efficient version, like this:

```
void swap(String &x, String &y) {
    x.swap(y);
}
```

- What about other types with their own `swap` member function?

20

Swapping Objects

- Alternatively, we could write the generic swap in terms of a swap member function, like this:

```
template <typename T>
void swap(T &x, T &y) {
    x.swap(y);
}
```

- Unfortunately, this version compiles only when T is a class type that provide its own swap member function.
- It won't compile when T is any non-class type.

21

Swapping Objects

- The generic swap is inefficient when it generates unnecessary copies.
- A type-specific swap can avoid generating these copies.
- Writing a generic swap that takes advantage of type-specific swap functions when they exist is difficult in C++03.
- C++11 makes it easier by introducing move semantics for types, using a new feature called ***rvalue references***.
- Before we can talk about rvalue references, we have to talk about lvalues and rvalues...

22

Lvalues vs. Rvalues

- Every expression is either an lvalue or an rvalue.
- In general:
  - An ***lvalue*** is an expression that refers to an object.
  - An ***rvalue*** is an expression that isn't an lvalue.
- In truth, as you'll see shortly:
  - An rvalue of ***non-class*** type ***doesn't*** refer to an object.
  - An rvalue of ***class*** type ***does*** refer to an object.

23

Rvalues

- Most literals are rvalues, including:
  - numeric literals, such as `3` and `3.14159`, and
  - character literals, such as `'a'`.
- Enumeration constants are also rvalues.
- In an assignment, an rvalue can appear only on the right.
- For example,

```
enum color { red, green, blue };
~~~
blue = green;   // error: blue is an rvalue
10 = 0;         // error: 10 is an rvalue
```

24

Lvalues

- An identifier that names an object is an lvalue.
- So is the result of a dereferencing operation such as unary * or [].
- For example,

```
enum color { red, green, blue };
color c;
int a[10];
int *p = new int (42);
~~~
c = green;      // OK: c is an lvalue
a[3] = 17;      // OK: a[3] is an lvalue
*p += 12;       // OK: *p is an lvalue
```

25

Temporary Objects

- A temporary object (resulting from a computation) is an rvalue.
- For example,

```
int i, j, k;
~~~
i + j = k;      // error: why?
```

- + has precedence over =.
- Thus, the assignment is equivalent to:

```
(i + j) = k;    // error: i + j is an rvalue
```

26

Lvalues vs. Rvalues

- In an assignment:
  - an rvalue can appear only on the right, but
  - an lvalue can appear on either side.
- When an lvalue appears on the right in an assignment, the compiler performs an ***lvalue-to-rvalue conversion***:

```
int m, n;
~~~
m = n;      // OK: n is converted to an rvalue
```

27

Non-modifiable Lvalues

- A const object is an object.
- Thus, an expression referring to a const object is an lvalue.
- But you can't assign to it.
- An expression referring to a const object is a ***non-modifiable lvalue***:

```
int const max = 15;
int n;
~~~
n = max;    // OK
max = n;    // error: max is non-modifiable
++n;        // OK
++max;      // error: max is non-modifiable
```

28

Non-modifiable Lvalues vs. Rvalues

- A non-modifiable lvalue is like an rvalue in that you can't modify it.
- How are they different?
  - You ***can*** take the address of a ***non-modifiable lvalue***.
  - You ***can't*** take the address of an ***rvalue***.
- For example,

```
int const *p;
int const max = 15;
p = &max;           // OK: max is a non-modifiable lvalue
enum { min = -16 };
p = &min;           // error: min is an rvalue
```

29

Class Rvalues

- Again, a temporary object is an rvalue.
- A program can create class rvalues in a few ways:
  - by a function call that returns a class object by value, or
  - by a cast expression that yields a class type.
- For example, suppose we have:

```
class T {
public:
    T(int);         // constructor
    int f(int);     // non-static member function
    ~~~
};

T g(int);           // function returning T by value
```

30

Class Rvalues

- Then these are all class rvalues:

```
g(3);               // T returned by value
T(0)                // "function-style" cast
(T)0                // "C-style cast"
static_cast<T>(0)   // "new-style cast"
```

- A class rvalue acts like other rvalues in that you cannot takes its address, as in:

```
T *p = &T(42);      // error: T(42) is an rvalue
```

✓ *A number of popular compilers allow taking the address in permissive mode. Even if you get away with it it's still illegal.*

31

Class Rvalues

- A class rvalue acts like an lvalue in one significant way:
  - You can apply a member function to a class rvalue.
- For example, this applies `T::f(int)` to a `T` rvalue:

```
T(7).f(0);
```

- `f` is a non-const member function, so it can modify that rvalue.
- However, that rvalue is temporary, so the program destroys it immediately after `f` returns.
- This use of class rvalues is not uncommon:

```
cout << for_each(begin(v), end(v), Count()).count();
```

32

References

- So how do rvalue references fit into all of this?
- As you well know, C++ has a feature called "references".
- What C++03 calls ***"references"***, C++11 calls ***"lvalue references"***.
  - This is to distinguish them from the new ***"rvalue references"***.
- Except for the name change, lvalue references in C++11 behave just like references in C++03.
- Let's start by reviewing an important aspect of lvalue references...

33

Lvalue References and Lvalues

- A "pointer to T" can point only to an lvalue of type T.
- Similarly, an "lvalue reference to T" binds only to an lvalue of type T.
- For example, these are both compile errors:

```
int *pi = &3;        // can't apply & to 3
int &ri = 3;         // can't bind this, either
```

- These are also compile errors:

```
int i;
~~~
double *pd = &i;     // can't convert int * into double *
double &rd = i;      // can't bind this, either
```

34

Lvalue References and Temporaries

- There's an exception to the preceding rule.
- An "lvalue reference to ***const*** T" can bind to e, even if e is an rvalue or has a type other than T:

```
T const &r = e;     // e need not be lvalue of type T
```

- However, it's valid ***only if*** there's a conversion from e's type to T.
- In this case, the compiler creates a temporary object holding a copy of the converted rvalue.
- The lvalue reference then binds to that temporary.

35

Lvalue References and Temporaries

- For example,

```
double const &rd = 1;
```

- When executed:
  - it converts 1 from int to double,
  - it creates a temporary to hold the result of the conversion, and then
  - it binds rd to the temporary.
- The program destroys the temporary when execution leaves the scope containing rd.
- Compilers can, and often do, optimize the above steps as compile-time computations.

36

Lvalue References and Temporaries

- Compilers don't bind references to temporaries unnecessarily.
- For example:

```
int i;
int const &ri = i;
```

- Here, `ri` binds directly to `i`.
- The compiler doesn't invent a temporary to hold a copy of `i`.
- Actually, the compiler isn't obligated to reserve storage for `ri` at all.

37

Lvalue References and Temporaries

- Again, a compiler will bind an "lvalue reference to `const`" to a temporary.
- Binding an "lvalue reference to (non-`const`) `T`" to anything except an lvalue of type `T` produces a compilation error:

```
int n;
double &rd = 1.0;       // error: 1.0 not an lvalue
double &rd = n;         // error: n is not double
double const &rcd = 1;  // OK: put 1 into a temporary
```

- So how do rvalue references differ from lvalue references?

38

Rvalue References

- Whereas an lvalue reference declaration uses the `&` operator, an ***rvalue reference*** uses the `&&` operator.
- For example, this declares `ri` to be an "rvalue reference to `int`":

```
int &&ri = 10;
```

- You can use rvalue references as function parameter and return types, as in:

```
double &&f(int &&ri);
```

- You *can* also have an "rvalue reference to `const`", as in:

```
int const &&rci = 20;
```

39

Rvalue References

- Rvalue references bind only to rvalues.
- This is true even for "rvalue reference to `const`".
- For example,

```
int n = 10;
int &&ri = n;       // error: n is an lvalue
int const &&rj = n; // error: n is an lvalue
```

40

Rvalue References

- When an rvalue reference binds to an rvalue, it behaves just like an “lvalue reference to const”.
- That is, the program creates a temporary object that holds a copy of the converted rvalue and binds the rvalue reference to it.
- For example,

```
int &&ri = 3;   // OK: binds ri to a temporary
```

- `ri` is a “reference to (non-const)”, so we can modify the temporary:

```
++ri;           // OK: ri acts like lvalue
```

- You’ll see why this is useful in a moment.

41

References and Overloading

- You can overload a function that accepts an lvalue reference parameter with a function that accepts an rvalue reference parameter, as in:

```
class String {
public:
    String(String const &s);
    String(String &&s) noexcept;
    ~~~
};

String operator+(String const &s1, String const &s2);
```

42

References and Overloading

- When you create a new String, the compiler selects the constructor based on whether the constructor argument is an lvalue or an rvalue, as in:

```
String s1;
String s2 (s1);         // calls String(String const &)
String s3 (s1 + s2);    // calls String(String &&)
```

- Remember:
  - `s1` is an lvalue.
  - `s1 + s2` is an rvalue
- You can use this new overloading capability to improve the efficiency of certain operations.
- Here's how…

43

Rvalue References and Temporaries

- Rvalues (that is, temporary objects) are destroyed at the end of the statement in which they're created.
- Suppose the compiler encounters this:

```
String s1, s2, s3;
s1 = s2 + s3;
```

- It actually generates something like this:

```
String s1, s2, s3;
String temp (s2 + s3);  // create a temporary
s1 = temp;              // copy the temporary to s1
temp.~String();         // destroy the temporary
```

44

Rvalue References and Temporaries

- It doesn't matter if an rvalue's value changes just before the rvalue is destroyed.
- Rather than create an expensive copy of a short-lived rvalue's value, you can just steal it from that rvalue.
- Stealing is cheaper than replicating.
- The rvalue won't mind.

45

Rvalue References and Temporaries

- For example, when constructing a new `String` from a `String` rvalue, you need not allocate a new array.
- You can take the array from the soon-to-be-destroyed source `String` and use it to construct the target `String`:

```
String::String(String &&s) noexcept {
    array = s.array;
    size = s.size;
    s.array = nullptr;  // don't forget to do this...
}
```

- If you don't set `s.array` to null, the `String` destructor will delete the array that was just transferred.

46

Move Constructors

- The preceding constructor is called a ***move constructor***.
- A copy constructor for a class `T` is typically declared as:

```
T(T const &x);
```

- A move constructor for a class `T` is typically declared as:

```
T(T &&x) noexcept;
```

- A ***copy*** constructor typically performs a ***non-destructive*** copy.
- A ***move*** constructor usually performs a ***destructive*** copy.
- The standard (but now deprecated) `auto_ptr` class template is a rarity whose copy constructor is destructive.

47

Move Constructors

- Why not write a move constructor with a plain old lvalue reference?

```
String::String(String &s) {
    array = s.array;
    size = s.size;
    s.array = nullptr;
}
```

- This function above can lead to problems very quickly...

48

Move Constructors

- Using a move constructor written with an lvalue reference parameter yields surprising behavior:

```
String s1 ("867-5309"); // non-copy/non-move constructor
String s2 (s1);         // moves resources from s1 to s2
cout << s1 << endl;     // prints nothing
cout << s2 << endl;     // prints "867-5309"
```

- The first output statement prints nothing because `s2`'s definition sets `s1`'s value to the empty String.
- Move constructors are often faster than copy constructors, but…
- For most types we still want to be able to copy an object without destroying it.

49

Move Constructors

- We can get the behavior we want by implementing both a copy constructor and a move constructor, like this:

```
class String {
public:
    String(String const &s);     // copy constructor
    String(String &&s) noexcept; // move constructor
    ~~~
};
```

- Overload resolution favors the copy constructor for lvalue arguments and the move constructor for rvalue arguments.
- That is, we get a speed boost from move semantics only when we can get it safely.

50

Move Assignment

- A class can have move assignment operators, as in:

```
class String {
public:
    String(String const &s);        // copy constructor
    String &operator=(String const &s);
                                    // ^ copy assignment
    String(String &&s) noexcept;    // move constructor
    String &operator=(String &&s) noexcept;
    ~~~                             // ^ move assignment
};
```

- Just as a class can have both copy and move constructors…
- A class can have both copy and move assignment operators.

51

Move Assignment

- In contrast to the move constructor…
- The move assignment's destination String might own a valid array, which should be deleted.

```
String s1 ("BE4-5789");
~~~
s1 = String("A date?  July 11, 1962.");
```

52

Move Assignment

- The definition for a move assignment often looks similar to the definition of its companion move constructor:

```
String &String::operator=(String &&s) noexcept {
    if (this != &s) {        // check for self-assignment
        delete [] array;     // discard old value
        array = s.array;
        size = s.size;
        s.array = nullptr;
    }
    return *this;
}
```

✓ *A move assignment should return an lvalue reference.*

53

Move Semantics

- A class has ***move semantics*** if it has both a move constructor and a move assignment operator.
- What about a class that has one but not both?
  - It's not clear, and we don't want to go there.

✓ *If you declare either move operation in a class, declare both.*

- What about built-in types, such as integers and pointers?
  - Built-in types don't have move semantics — they have only copy semantics.
  - For built-in types, a destructive move has no performance advantage over a non-destructive copy.
- For types with no externally-managed resources, move == copy.

54

The "Moved-From" State

- Stroustrup [2013] says the source of the move must be left in a "moved-from state."
- What's a "moved-from state"?
- It is a "valid but unspecified" state. [Hinnant 2014]

✓ *It must be destroyable.*
  - Of course.

✓ *It must be copy-assignable and move-assignable.*
  - The standard library requires this.

✓ *There is a consensus that it must be empty of external resources.*
  - For example, a container should be empty of elements.

55

The "Moved-From" State

- Here are the `String`'s destructor and copy assignment:

```
String::~String() {
    delete [] array;
}

String &String::operator =(String const &rhs) {
    if (size != rhs.size) { // re-use buffer if possible
        char *tmp = new char [rhs.size + 1];
        delete [] array;
        array = tmp;
        size = rhs.size;
    }
    strncpy(rhs.array, array, size);
}
```

56

The “Moved-From” State

- The move operations do leave the moved-from `String` in a destroyable state.
- However, a moved-from `String` is not copy-assignable.
- Fixing the move constructor is straightforward:

```
String::String(String &&s) noexcept {
    array = s.array;
    size = s.size;
    s.array = nullptr;
    s.size = 0;
}
```

- The move assignment requires the same fix.

57

Moves Should Not Throw

- Copy operations often allocate resources.
  - They may throw exceptions.
- Move operations typically transfer already allocated resources.
  - In general, they should not throw exceptions.

✓ *Design move operations so that they don’t throw exceptions and declare them `noexcept`.*

```
class String {
public:
    String(String &&s) noexcept;
    String &operator=(String &&s) noexcept;
    void swap(String &) noexcept;
    ~~~
};
```

58

Swapping Objects Revisited

- We can now implement a more efficient swap that uses move semantics.
- Note that this version still uses copy semantics:

```
template <typename T>
void swap(T &x, T &y) {
    T temp (x);
    x = y;
    y = temp;
}
```

- It uses copy semantics even if the type substituted for T is a class with move semantics.
- Why is that?

59

Swapping Objects Revisited

- If T is a class type, then the definition:

```
T temp (x);
```

uses either:

```
T(T const &t);      // copy constructor
T(T &&t);           // move constructor
```

- In this case, x is an lvalue, so overload resolution selects the copy constructor.

60

Swapping Objects Revisited

- Similarly, each assignment statement:

```
x = y;
y = temp;
```

  uses either:

```
T &operator=(T const &t);   // copy assignment
T &operator=(T &&t);        // move assignment
```

- In each case, the right-hand operand (either `y` or `temp`) is an lvalue, so overload resolution selects the copy assignment.
- How can we make `swap` use move semantics instead?

61

The `std::move` Function

- We can get the compiler to favor move semantics over copy semantics by using the standard `move` function template:

```
template <typename T>
void swap(T &x, T &y) {
    T temp (std::move(x));  // favors T(T &&)
    x = std::move(y);       // favors T &operator=(T &&)
    y = std::move(temp);    // favors T &operator=(T &&)
}
```

- `std::move(x)` returns `x` as an rvalue by casting it to an rvalue reference.
- This version uses move semantics if `T` supports them and copy semantics if it doesn't.

62

Move Self-Assignment

- Our implementation of `String` move assignment handles the case for self assignment.
- It's not difficult to provoke a move self-assignment by treating an lvalue as an rvalue:

```
String s ("CAVE people");
s = move(s);        // contrived move self-assignment
String &t = s;
s = move(t);        // less contrived
```

- It is careless, though.
- Should we handle move self-assignment?

63

Move Self-Assignment

- One school of thought notes that the standard says an implementation may assume an rvalue reference function parameter is the only reference to the argument:

```
String &String::operator =(String &&rhs);
```

- For assignment, the implementation can therefore assume that `&rhs != this`.

✓ *If rhs refers to a temporary, then* `this != rhs`. *No need to check.*

✓ *If rhs refers to an explicitly-moved object (using std::move) then it's usually the responsibility of the code that called std::move to make sure self-assignment doesn't occur.*

- However, the consensus seems to be that move assignment should nevertheless be able to handle self assignment.

64

Not Considering Self-Assignment

- The "noalias" school would remove the check for self-assignment.

```
String &String::operator=(String &&s) noexcept {
    if (this != &s) {
        delete [] array;
        array = s.array;
        size = s.size;
        s.array = nullptr;
        s.size = 0;
    }
    return *this;
}
```

65

Considering Self-Assignment

- The "alias-possible" school might accept a modest additional cost for safety:

```
String &String::operator=(String &&s) noexcept {
    if (this != &s) {
        delete [] array;
        char *temp = array;
        array = s.array;
        size = s.size;
        s.array = nullptr;
        s.size = 0;
        delete[] temp;
    }
    return *this;
}
```

66

Move Self-Assignment

- But what does move self-assignment mean?

```
auto x = something;
x = move(x);
```

- We know that the source of a move assignment should be left in a valid but unspecified state.
- Therefore, after move self-assignment the object should be left in a valid but unspecified state.
  - The value of `x` may or may not be `something`.
- Our recommendation is…

✓ *Move self-assignment does not crash, but leaves the object in an unspecified, valid state.*

67

The State of “Moved From” Resources

- Consider a simple class that locks a mutex:

```
class R {
public:
    R(mutex &m) : lock_(m) {}
    R &operator =(R &&rhs) noexcept;
private:
    unique_lock<mutex> lock_;
};
```

- A simple “swap based” move assignment implementation:

```
R &R::operator =(R &&rhs) noexcept
    { swap(lock_, rhs.lock_); return *this; }
```

68

Not So Simple, After All

- This simple implementation is likely to surprise users:

```
mutex m1, m2;
~~~
R a (m1);    // a locks m1
R b (m2);    // b locks m2
~~~
a = move(b); // b not resource-free, has a's old lock
R c (m1);    // Exception: device or resource busy!
```

- The emerging consensus indicates that the moved-from state implies that all resources have been freed or transferred.

✓ *In general, do not implement move assignment by simply swapping resources.*

69

Proper Moves

- A better implementation of this move assignment would leave the source of the assignment without a locked mutex:

```
R &R::operator =(R &&rhs) noexcept {
    swap(lock_, rhs.lock_);
    lock_ = move(rhs.lock_);
    return *this;
}
~~~
R a (m1);    // a locks m1
R b (m2);    // b locks m2
~~~
a = move(b); // a has lock on m2, m1 unlocked
R c (m1);    // c locks m1
```

70

Swap Move Assignment

- We could simplify `String`'s move assignment operator using the standard `swap` function template:

```
String &String::operator=(String &&s) noexcept {
    std::swap(array, s.array);
    std::swap(size, s.size);
    return *this;
}
```

- This doesn't delete the destination `String`'s array explicitly.
- Rather, it hands that array to `s`, to be deleted when the destructor is called for `s`.
- However...

71

Moving and Resources

- This swap-based implementation of `String`'s move assignment leaves the source of the move in possession of a memory resource:

```
String a ("867-5309");
String b ("BE4-5789");
~~~
a = move(b); // b not resource-free, has a's old memory
```

- This is a less drastic situation than deadlock, but `b`'s memory will not be recovered until `b` is destroyed at the end of its scope.
- Worse, `b` may be used after the move with an expectation that it has a well-defined value.

✓ *Generally: The source of a move should hold no resources after the move.*

72

Moving "Rules"

- ✓ *A moved-from object must be destroyable.*
- ✓ *A moved-from object should be assignable.*
- ✓ *A moved-from object should hold no external resources.*
- ✓ *Move operations should generally be noexcept.*
- ✓ *Move self-assignment is allowed and leaves the moved-from object in a valid but unspecified state.*

73

Aside: noexcept

- We'd also like our swap to be noexcept, if possible:

```
template <typename T>
void swap(T &x, T &y) noexcept {
    T temp (std::move(x));
    x = std::move(y);
    y = std::move(temp);
}
```

- However, for some types T, the move constructor or move assignment might throw.
- You should declare the function noexcept only if those functions won't throw…

74

Aside: noexcept

- The noexcept specification may be followed by a parenthesized boolean constant-expression:

```
template <typename T>
void swap(T &x, T &y) noexcept (
    std::is_nothrow_move_constructible<T>::value
    && std::is_nothrow_move_assignable<T>::value
) {
    ~~~
}
```

- The noexcept specification applies only if the condition is true.
- noexcept by itself is short for noexcept (true).

75

Aside: Conditional Moving

- The std::move function template performs an unconditional cast to an rvalue reference.
- A lightly-used variant of std::move is std::move_if_noexcept, that will conditionally cast to an rvalue reference or an lvalue reference to const.
- If the var has type T, then
  - move_if_noexcept(var) will cast to T const & if T does not have a nothrow move constructor but does have a copy constructor.
  - Otherwise, it will cast to T &&, as move does.
- The intent is to ease the task of writing strongly exception safe template code in situations where we'd like the code to be move-enabled, if possible.

76

Motivation for Conditional Moving

- The motivation for this feature was originally outlined in Abrahams [2010]:
- "...it's a backward-compatibility/code evolution issue that only arises when all these conditions are satisfied:
  - An existing operation today gives the strong guarantee
  - The operation is being move-enabled (altered to use move operations)
  - An existing type that the operation manipulates acquires a move constructor
  - That move constructor can throw
  - The particular move-enabled operation can only offer the basic guarantee if a move constructor throws"

77

```
template <typename T>
T *resize(T *a, size_t n, size_t newsize) {
    T *temp =
       static_cast<T *>(operator new(sizeof(T)*newsize));
    size_t i = 0;
    try {
        for (; i != n; ++i)
            new (static_cast<void *>(&temp[i]))
                T(move_if_noexcept(a[i]));
    }
    catch (...) {
        while (i > 0) temp[--i].~T();
        operator delete(temp);
        throw;
    }
    return temp;
}
```

78

Temporaries and Return By Value

- Some functions should return their result by value.
- For example, the built-in binary + operator yields an rvalue.
- A user-defined operator + should, too, by using return by value:

```
String operator +(String const &a, String const &b)  {
    String temp (a);
    temp += b;
    return temp;    // invokes copy or move constructor
}
```

- Conceptually, the compiler creates a temporary String object in the caller to hold the value returned from operator +.
- The return statement in operator + initializes that temporary.

79

Temporaries and Return By Value

- The return value initialization is typically implemented (something) like this:

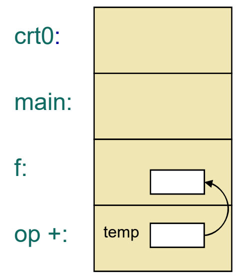
```
void operator +(
    String &ret, String const &a, String const &b
) {
    String temp (a);
    temp += b;
    ret.String(temp);   // apply constructor (compiler
                        // can do this; we can't!)
}
```

- Note: Defining operator + with three parameters is something the compiler can do, but we can't.

80

Return By Value

```
void f(String const &a, String const &b) {
    ~~~
    String c (a + b);
    ~~~
}
String operator +(String const &a, String const &b) {
    String temp (a);
    temp += b;
    return temp;  // copy ctor
}
```



81

The Return Value Optimization

- The compiler is allowed to eliminate the returned local variable, its copy and destruction, and substitute the return location.

```
X f(int a) {
    X local (a);
    // do stuff with local...
    return local;
}
```

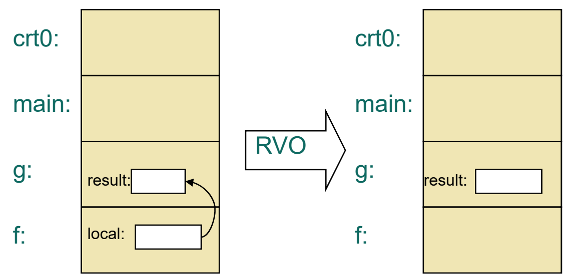
may be modified by the compiler to

```
void f(X &result, int a) {
    result.X::X(a);
    // do stuff with result...
}
```

82

RVO

```
void g() {
    ~~~
    X result (f(12));
   ~~~
}
X f(int a) {
    X local (a);
    // do stuff with local...
    return local;
}
```



83

Prefer Initialization to Assignment

- Note that the RVO can't copy directly into the target of an assignment.
- Consider this assignment:

```
c = a + b;
```

- The compiler will probably translate it into something like:

```
String tmp;
tmp.String(a.s_, b.s_);
c = tmp;
tmp.~String();
```

- Initialization is generally more efficient than assignment.

84

Return by Value

- Again, a binary operator that creates a new value should return its result by value:

```
String operator +(String const &a, String const &b);
```

- Although the returned object is an rvalue, it's modifiable.
- It may be misused:

```
String a ("Hello,"), b (" World!"), c ("reset");
~~~
a + b = c;          // creative! But unfortunate...

operator +(a, b).operator =(c); // same thing...
```

85

Return by Const Value

- Declaring the return type `const` avoids this particular problem:

```
String const
operator +(String const &a, String const &b);
~~~
a + b = c;       // compile error! Can't assign to const
```

- This used to be recommended practice.
- Now it isn't recommended.

86

Don’t Return Const Values

- Unfortunately, declaring the return type const prevents efficient move assignment in modern C++.
- Recall that the ***move*** assignment’s parameter is an “rvalue reference to ***non-const***”, whereas the ***copy*** assignment’s parameter is an “lvalue reference to ***const***”:

```
String &String::operator =(String &&rhs) noexcept;
String &String::operator =(String const &rhs);
```

- When `operator +` has a const-qualified return type, this expression uses the copy assignment, not the move assignment:

```
a = b + c;  // inefficient! chooses copy, not move
```

✓ *When returning by value, return non-const.*

87

Value Return of Locals

- You should not use `std::move` in cases such as:

```
T operator +(T const &a, T const &b) {
    T tmp (a);
    tmp += b;
    return std::move(tmp);  // unnecessary, and might
}                           // be counterproductive
```

- Using `std::move` is unnecessary here because a local variable that’s returned by value is already treated as an rvalue.
- Using `std::move` could be counterproductive because it prevents consideration of an even more effective return value optimization.

88

Value Return of Locals

```
T operator +(T const &a, T const &b) {
    T tmp (a);
    tmp += b;
    return tmp;              // better
}
```

- In this case, the compiler will consider, in order, the following return strategies:
  1. the return value optimization
  2. move constructing the return value
  3. copy constructing the return value

✓ *Do not std::move value returns of locals.*

89

By-Value Parameters and RVO

- A pass-by-value parameter is also a local variable of a function, but RVO does not apply to return of a value parameter:

```
T operator +(T a, T const &b) {
    a += b;
    return a;              // RVO doesn't apply
}
```

- While it may be tempting to `std::move` the return value, it's not necessary.
- The compiler will first attempt to move the return value, and copy only if that fails.

✓ *Do not std::move returns of by-value parameters.*

90

Reference Parameters and the RVO

- Consider an overload to our binary operator:

```
T operator +(T const &a, T const &b); // #1
T operator +(T &&a, T const &b);      // #2
```

- The second version of `operator +` is intended to optimize the case where the left argument of the addition expression is an rvalue.

```
T a, b, c;
T d = a * b + c; // add c to rvalue, use #2
T e = b + c;     // add c to lvalue, use #1
```

91

Move Rvalue Reference Parameters

- A straightforward implementation is sub-optimal:

```
T operator +(T &&a, T const &b) {
    return a += b;            // no RVO, copy result
}
```

- As a result, `a` is copied to the return location.
- In this situation it makes sense to move the return value:

```
T operator +(T &&a, T const &b) {
    return std::move(a += b);  // move result
}
```

92

You May Move But Once

- Once an object has been moved, the source of the move is good only for destruction or as the target of an assignment.

```
bool is_palindrome(string a);
bool is_perfect_pangram(string a);
~~~
void game(string &&s) {
    if (is_perfect_pangram(move(s))
        && is_palindrome(move(s)))
        cout << "impossible!!!" << endl;
}
```

- After the first move s is likely to have changed.
- In case of a string, the moved-from state is possibly a null string, which is a (degenerate) palindrome. Impossible!

93

You May Move But Once

- If you plan to use the value of an object multiple times, you should move it only on its last use:

```
void game(string &&s) {
    if (is_perfect_pangram(s)
        && is_palindrome(move(s)))
        cout << "impossible!!!" << endl;
}
~~~
game("Stop! Murder us not, tonsured rumpots!");
game("Quartz glyph job vex'd cwm finks.");
```

94

A Variable is an Lvalue

- Note that a variable is always an lvalue, even if it has an rvalue reference type.

```
void func(String &&s);        // #1
void func(String const &s);   // #2
~~~
String &&getAString();
String &&tmp = getAString();

func(tmp);                    // call #2
func(move(tmp));              // call #1
func(getAString());           // call #1
func((String &&)tmp);         // call #1
```

95

A Named Variable is an Lvalue

- How would you write this game if s were an rvalue?

```
bool is_palindrome(string a);
bool is_perfect_pangram(string a);
~~~
void game_prime(string &&s) { // If s were an rvalue
    if (is_perfect_pangram(s) // this would be a move
        && is_palindrome(s))  // and this would be bad.
        cout << "impossible!!!" << endl;
}
```

- Fortunately, the argument s is an lvalue.

96

Addendum: Modern C++ Expressions

expression

glvalue *rvalue*

lvalue *xvalue* *prvalue*

- Modern C++ introduces a more complex categorization of types of expressions.
- It expands the traditional lvalue/rvalue dichotomy to five overlapping categories.

97

Value Categories

expression

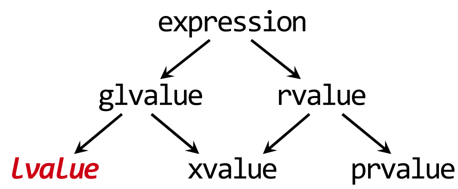
glvalue rvalue

lvalue *xvalue* *prvalue*

- Every expression belongs to exactly one of the categories lvalue, xvalue, or prvalue.
- This is the expression's ***value category***.

98

Lvalues

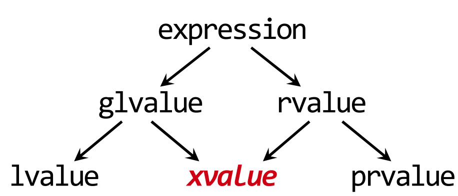


- As before, an ***lvalue*** designates a function or an object:

```
T var;
var = aT;
T *p = &aT;
*p = aT;
T &func();
func() = aT;
```

99

Xvalues

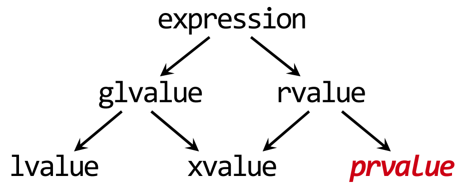


- An ***xvalue*** designates an “expiring” value.
- It refers to an object that’s (typically) near the end of its lifetime.
- Expressions involving rvalue references yield xvalues:

```
Product &&factory();
Product p = factory();
Product q = move(p);
```

100

Prvalues

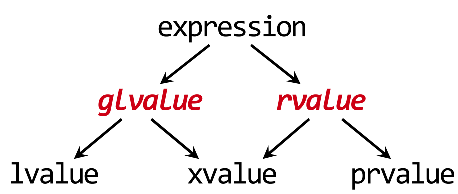


- A ***prvalue*** is a "pure rvalue" — an rvalue that's not an xvalue:

```
int const &r = 12;
T func();
T t = func();
T *p = nullptr;
```

101

Glvalues and Rvalues



- A ***glvalue*** is a "generalized lvalue".
- Think of it as an expression that refers to an object in memory or to a function.
- Think of an ***rvalue*** as an expression that can be used initialize an rvalue reference.

102

Collected Advice

- ✓ *A move assignment should return an lvalue reference.*
- ✓ *If you declare either move operation in a class, declare both.*
- ✓ *Resource transfer that leaves a resource-free source will become the conventional meaning of move.*
- ✓ *Design move operations so that they don't throw exceptions and declare them* `noexcept`*.*
- ✓ *Move self-assignment does not crash, but leaves the object in an unspecified, valid state.*
- ✓ *Do not automatically implement move assignment by swapping resources.*
- ✓ *Generally: The source of a move should hold no resources after the move.*
- ✓ *When returning by value, return non-const.*
- ✓ *Do not* `std::move` *value returns of locals.*

103

104

Universal References and Perfect Forwarding

105

Reference Collapsing

- You can't declare a reference to a reference directly:

```
int a = 12;
int & &ri = a;  // error! reference to reference
```

- However, references to references can occur indirectly by using a typedef:

```
typedef int &RI;
RI &ri = a;     // OK: ri is int &
```

- In this case, the compiler will perform ***reference collapsing***.
- A sequence of lvalue reference modifiers "collapse" to a single lvalue reference modifier.

106

Reference Collapsing

- Reference collapsing also occurs in template specialization:

```
template <typename T>
void f(T &arg);
~~~
f<int const &>(12);         // OK, arg is int const &
```

- It also occurs with `decltype`:

```
int &ri = anint;
decltype(ri) ri2 = anint;   // ri2 is int &
decltype(ri) &ri3 = anint;  // ri3 is also int &
```

107

Rvalue Reference Collapsing

- Rvalue references can also collapse:

```
int &&rri = 12;
decltype(rri) rri2 = 12;    // rri2 is int &&
decltype(rri) &&rri3 = 12;  // ri3 is also int &&
```

- However, any sequence of both lvalue and rvalue references always collapses to an lvalue reference:

```
int &ri = anint;
decltype(ri) &&ri4 = 12;    // error! ri4 is int &
```

- The number and order of reference modifiers is immaterial.

108

Rvalue References and Type Deduction

- Again, an rvalue reference must refer to an rvalue:

```
int &&r1 = 12;      // OK
int a = 12;
int &&r2 = a;       // error! a is an lvalue
```

- With type deduction, a declared rvalue reference may actually be an lvalue reference:

```
auto &&r1 = 12;     // OK: r1 is int &&
int a = 12;
auto &&r2 = a;      // OK: r2 is int &!
```

109

Rvalue References and Type Deduction

- Use of auto with an initializer is a type deduction context.
- The compiler deduces the declared type from the initializer.
  - It's *almost* identical to template argument deduction.
- The type deduction rules for declared rvalue references differ when the initializer is an lvalue or an rvalue!
  - When it's an ***lvalue*** of type T, the deduced type is ***T &***.
  - When it's an ***rvalue*** of type T, the deduced type is ***T***.

```
auto &&r1 = 12; // rvalue initializer: r1 is int &&
int a = 12;
auto &&r2 = a;  // lvalue initializer: r2 is int & &&
                // after collapsing:   r2 is int &
```

110

Rvalue References and Type Deduction

- Things are particularly interesting and useful in the context of template argument deduction:

```
template <typename T>
void f(T &&r) { ~~~ }
```

- The effective function parameter type depends on the function call argument's properties:

```
int a = 10, b = 12, c = 0;
f(12);      // rvalue argument: T is int,   r is int &&
f(a);       // lvalue argument: T is int &, r is int &
f(a+b);     // rvalue argument: T is int,   r is int &&
f(c = a+b); // lvalue argument: T is int &, r is int &
```

111

Rvalue Reference Argument Deduction

- If the argument to a `T &&` parameter is an *rvalue* of type `X`, then the template type parameter is deduced to be just `X`:

```
template <typename T>
void munge(T &&to_munge);
~~~
munge(String("temp"));  // munge<String>(String("temp"))
```

- Here, the compiler deduces `T` to be `String`.
- The function parameter type is `T &&`.

112

Rvalue Reference Argument Deduction

- If the argument to a `T &&` parameter is an *lvalue* of type `X`, then the template type parameter `T` is deduced to be `X &`:

```
template <typename T>
void munge(T &&to_munge);
~~~
String m1;      // m1 is an lvalue
munge(m1);      // same as munge<String &>(m1);
```

- Here, the compiler deduces `T` to be `String &`.
- The function parameter type `String & &&` collapses to `String &`.
- Thus, a function template declared with an rvalue reference parameter may be specialized as a function with an lvalue reference parameter.

113

Lvalue/Rvalue Overload

- You can overload function templates on rvalue reference and lvalue reference parameters:

```
template <typename T>       // #1
void munge(T &to_munge);
template <typename T>       // #2
void munge(T &&to_munge);
~~~
String s ("Masie");
String const t ("Constance");
~~~
munge(s);                   // #1: T is String
munge(t);                   // #1: T is String const
munge(String("temp"));      // #2: T is String
```

114

Lvalue/Rvalue Max Overload

- Including an overload for const:

```
template <typename T>       // #1
void munge(T &to_munge);
template <typename T>       // #2
void munge(T &&to_munge);
template <typename T>       // #3
void munge(T const &to_munge);
~~~
String s ("Masie");
String const t ("Constance");
munge(s);                   // #1: T is String
munge(t);                   // #3: T is String
munge(String("temp"));      // #2: T is String
```

115

Overload Confusion

- In both of the previous two sets of overloaded functions, it may appear that some calls should be ambiguous. For example:

```
template <typename T> void munge(T &);       // #1
template <typename T> void munge(T &&);      // #2
template <typename T> void munge(T const &); // #3
~~~
X var;
munge(var); // T is X, calls #1
```

- A reasonable programmer might object that T could also be deduced as `T &` for overload #2 which, after reference collapsing, would result in the same argument list as #1.

✓ *It's best not to disappoint reasonable programmers. Avoid ambiguous-appearing overloads.*

116

Lvalue/Rvalue No Overload

- Without overloading:

```
template <typename T>        // #2
void munge(T &&to_munge);
~~~
String s;
String const t;

munge(s);                    // #2, T is String &
munge(t);                    // #2, T is String const &
munge(String("temp"));       // #2, T is String
```

- Rvalue reference arguments in a type deduction context are *very* accommodating.

117

Too Accommodating

- Consider a different set of overloads:

```
template <typename T>        // #2
void munge(T &&to_munge);
void munge(size_t);          // #4
~~~
munge(123);                  // calls #2!, T is int
size_t n1 = 123;
munge(n1);                   // calls #4
auto n2 = 123;
munge(n2);                   // calls #2!, T is int &
```

✓ *Generally, we prefer not to overload on rvalue references in a type deduction context.*

118

Subtle Accommodation

- Here's a more subtle example of the problem:

```
template <typename T>
void munge(T &&);          // #2
void munge(X const &);     // #5
~~~
X var;
munge(var);                // calls #2!
```

- Here, T is deduced to `X &`, `X & &&` collapses to `X &`, and #2's `X &` is a better match than #5's `X const &`.
- This *may* appear *somewhat* less unreasonable than previous examples, but it can still be "surprising."
- Reasonable programmers don't like to be surprised…

119

Advice

- This is OK:

```
void munge(String const &);
void munge(String &&);
```

  There's no argument deduction going on.

- This is OK:

```
template <typename T>
void munge(T &&);
```

  There's no overloading going on.

120

Advice

- This is worrisome:

```
template <typename T> void munge(T const &);
template <typename T> void munge(T &&);
```

- This is deadly:

```
void munge(X const &);
template <typename T> void munge(T &&);
```

- This is deadly:

```
void munge(double);
template <typename T> void munge(T &&);
```

121

Lvalue/Rvalue/Const Member Overload

- In the context of a (non-template) member function of a template, argument deduction doesn't come into play if the argument type is already known due to class template specialization.
- For example:

```
template <typename T>
class X { ~~~ };

X<Widget> obj;
obj.func();        // no deduction for func; T is Widget
```

122

Lvalue/Rvalue/Const Member Overload

```
template <typename T>
struct Munge {
    void munge(T &&to_munge);       // #1
    void munge(T const &to_munge);  // #2
    void munge(T &to_munge);        // #3
};
~~~
Munge<String> m;
String mungeable;
String const unmungeable;

m.munge(mungeable);                 // #3: lvalue
m.munge(String("pro tempore"));     // #1: rvalue
m.munge(unmungeable);               // #2: const lvalue
```

123

Standard Example

- It's unusual to overload on all three combinations of lvalue, non-modifiable lvalue, and rvalue.
- However, `std::array` has all three of these overloads for its nonmember get.

```
template <size_t I, typename T, size_t N>
constexpr T &get(array<T, N> &a) noexcept;

template <size_t I, typename T, size_t N>
constexpr T &&get(array<T, N> &&a) noexcept;

template <size_t I, typename T, size_t N>
constexpr const T &get(const array<T, N> &a) noexcept;
```

124

More Conventionally…

- Usually the question of interest is whether the argument is an lvalue or an rvalue.
- That is, can the argument be moved, or must it be copied?

```
template <typename T>
struct Munge {
    void munge(T &&to_munge);        // #1, can move
    void munge(T const &to_munge);   // #2, must copy
    void munge(T &to_munge);         // #3
};
~~~
m.munge(mungeable);                  // calls #3 #2
m.munge(String("pro tempore"));      // calls #1
m.munge(unmungeable);                // calls #2
```

125

Standard Example

- This lvalue/rvalue overloading is common in STL containers:

```
template <typename T, typename Allocator = allocator<T>>
class forward_list {
    ~~~
    forward_list(forward_list const &); // lvalue, copy
    forward_list(forward_list &&);      // rvalue, move

    forward_list &operator =(forward_list const &);
    forward_list &operator =(forward_list &&);

    void push_front(T const &);         // lvalue, copy
    void push_front(T &&);              // rvalue, move
    ~~~
```

126

Container Insertion

- For example, in modern C++, standard containers overload the push_back operation to allow optimization:

```
void push_back(T const &);  // #1: for lvalues
void push_back(T &&);       // #2: for rvalues
```

- This ameliorates the situation in which a temporary is used for insertion, then discarded.

```
vector<string> names1;
list<string> names2;
~~~
names1.push_back(names2.front());                // #1
names1.push_back(String("Waste not, want not!"));  // #2
```

127

Container Insertion

- Use of rvalue reference parameters allows moves rather than copies.
- Argument passing by moving is not ideal because it creates and destroys a temporary.
- But it's typically more efficient than copying.

128

Emplacement

- It's often more efficient to avoid creating the temporary entirely, skip the copy or move, and just initialize a new container element directly.
- Modern C++ containers add an "emplace" operation to do just that.
- The emplacement operation is a member template.
  - A call to an emplacement operation is a template deduction context.
  - It employs the special deduction semantics for rvalue reference arguments.
- The interface is careful not to overload insertion and emplacement operations!

129

Emplacement

```
void push_back(T const &);           // #1: for lvalues
void push_back(T &&);                // #2: for rvalues
template <typename... Args>
void emplace_back(Args &&... args); // #3: direct
```

- An emplacement operation takes a set of constructor arguments, and initializes the new element directly.
- The standard library defines the emplacement operations as member templates and employs variadic templates in their implementation:

```
names1.push_back(names2.front());                   // #1
names1.push_back(string("Waste, not, want not!"));  // #2
names1.emplace_back(12, '*');                       // #3
```

130

Forwarding References

- An rvalue reference used in a type deduction context is often called a "forwarding reference."
- The most common use for a forwarding reference is, not surprisingly, to forward a call:

```
void doit(string const &arg);   // for lvalues
void doit(string &&arg);        // for rvalues

template <typename T>
void dispatch(T &&arg) {        // a forwarding reference
    // do something...
    // then call the appropriate doit
}
```

131

Universal References

- Scott Meyers has coined "universal reference" as a useful alternative name for forwarding reference.

```
template <typename T>
void dispatch(T &&arg);         // a universal reference
```

- Remember: a universal/forwarding reference must have the form `T &&` in a context where `T` is deduced.
- These are not universal references:

```
void doit(string &&arg);        // for rvalue strings
void doit(string const &arg);   // for lvalue strings
template <typename T>
void doit(vector<T> &&);        // for rvalue vectors
```

132

Perfect Forwarding

- The trick is to recover information about the original argument to dispatch, and forward it perfectly to its destination:

```
void doit(string &&arg);          // for rvalues
void doit(string const &arg);     // for lvalues

template <typename T>
void dispatch(T &&arg) { ~~~ }    // a perfect forwarder
~~~
string str1 = "Hello, ";
dispatch(str1);
dispatch(string("World!"));
```

133

Implementing Dispatch

- A successful call to dispatch results in a deduced parameter type of either T & or T &&.
- It seems like the implementation of dispatch should be straightforward:

```
template <typename T>
void dispatch(T &&arg) {
    ~~~
    doit(arg);      // call appropriate doit... not!
}
```

- Even if the name arg has type T &&, it is a named variable.
- Named variables are lvalues and always result in a call to the lvalue version of doit.

134

Implementing Dispatch

- We can change this behavior with a well-disguised cast:

```
template <typename T>
void dispatch(T &&arg) {
    ~~~
    doit(std::move(arg));    // nope.
}
```

- But now we will always call the version of `doit` for rvalues.
- `std::move` is, basically, a cast to an rvalue reference.

135

A La Recherche Du Type Perdu

- We need a mechanism for recovering the original type of the argument that was passed to `dispatch`.
- That is the purpose of `std::forward`:

```
template <typename T>
void dispatch(T &&arg) {
    ~~~
    doit(std::forward<T>(arg)); // "perfect" forwarding
}
```

136

move vs. forward

- `std::move` is an *unambiguous* cast to an rvalue reference.
- Its template argument is deduced from the function argument.

```
doit(std::move(arg));
```

- `std::forward` is a *potential* cast to an rvalue reference:

```
doit(std::forward<T>(arg));
```

- Calling `forward` uses the types of both template argument `T` and function argument `arg` to recover the original argument type.

137

Idiomatic Forwarding Patterns

- Here's how you forward a single argument:

```
template <typename T>
void dispatch(T &&arg) {
    doit(std::forward<T>(arg));
}
```

- Here's how you forward an argument list generated from a pack expansion:

```
template <typename... Ts>
void dispatch(Ts &&... args) {
    doit(std::forward<Ts>(args)...);
}
```

138

Example: make_unique

- The C++14 make_unique helper function allocates an object of arbitrary type, initializes it with an arbitrary set of constructor arguments, and wraps an appropriate unique_ptr around the result:

```
template<typename T, typename... Args>
unique_ptr<T> make_unique(Args &&... args) {
    return unique_ptr<T>
        (new T(std::forward<Args>(args)...));
}
```

- make_unique is a factory function.
- It uses forward and variadic templates in combination to avoid a combinatorial explosion of overloaded make_unique functions.

139

Aside: T const &&

- You can declare and use an “rvalue reference to const”.
- However, the special rules that apply to argument deduction apply only to “rvalue reference to non-const”:

```
template <typename T>
void dispatch(T const &&arg) {
    ~~~
}
~~~
string str;
dispatch(str);  // error! T is string, not string &
                // arg is string const &&,
                // not string const &
```

140

Reprise: Member Functions

- Here's a typical use of overloaded members in a class template:

```
template <typename T>
class X {
public:
    void operation(T const &);  // for lvalues
    void operation(T &&);       // for rvalues
    ~~~
};
```

- The `T &&` argument is not a universal reference.
- `T` is fixed when `X` is specialized, so there's no argument deduction going on.

141

Universal Fail

- In this implementation, `T &&` is not a universal reference because there's no template argument deduction going on:

```
template <typename T>
class X {
public:
    void operation(T const &);   // no lvalues
    void operation(T &&);        // only rvalues
    ~~~
};
string str ("lval");
X<string> a;                    // T is string
a.operation(string("rval"));    // no deduction, OK
a.operation(str);               // no deduction, error
```

142

Universal Member Functions

- However, a member template can be used to establish a type deduction context:

```
template <typename T>
class X {
public:
    template <typename S>
    void operation(S &&);        // now universal
    ~~~
};
string str ("lval");
X<string> a;                     // T is string
a.operation(string("rval"));     // deduction, string &&
a.operation(str);                // deduction, string &
a.operation(12);                 // deduction, int &&
```

143

Greedy Universal Members

- Recall that universal references are accommodating:

```
template <typename T>
class X {
public:
    void operation(T const &);  // #1: lvalue version
    void operation(T &&);        // #2: rvalue version
    template <typename S>
    void operation(S &&);        // #3: universal version
    ~~~
};
```

- They often provide somewhat surprising better matches than functions without universal reference arguments…

144

Universal Greediness

- A call with an rvalue argument will match both the rvalue and universal versions.
- The non-template match is preferred.

```
X<T> a;
a.operation(T());   // call #2, rvalue version
```

- A call with a const lvalue argument will match both the lvalue and universal versions.
- The non-template match is preferred.

```
T const aT;
a.operation(aT);   // call #1, lvalue version
```

145

Universal Greediness

- A call with a non-const lvalue will match both the lvalue and universal versions.
- However, the universal version is a better match.
- It doesn't require a non-const to const conversion.

```
T anT;
a.operation(anT);   // call #3, universal version!
```

- Surprise!

✓ *Prefer not to overload with universal references.*

146

Decay

- There are many examples of decay in C++:

```
template <typename T> void death(T arg);
Widget wa[4];

death(wa);                        // arg is Widget *
death(death<int>);                // arg is void (*)(int)
```

- Other conversions are similar to decay:

```
const volatile float &cf = 1.2f;
death(cf);                        // arg is float
```

147

std::decay

- The `std::decay` type trait models the conversions and decay that occur when passing by value:

```
typename decay<Widget[4]>::type // Widget *, C++11
decay_t<Widget[4]>              // Widget *, C++14
decay_t<void(int)>              // void (*)(int)
decay_t<const volatile float &> // float
```

- We can use mutual decay to decide whether two types are "pretty much" the same:

```
template <typename S, typename T>
using similar = is_same<decay_t<S>, decay_t<T>>;
```

148

Limiting Greediness

- We can use SFINAE to limit the use of the universal version of `operation` to types that are "not similar to" the type used to specialize `X`:

```
template <typename T>
class X {
public:
  void operation(T const &);  // #1: lvalue version
  void operation(T &&);       // #2: rvalue version
  template <typename S,
           typename = enable_if_t<!similar<S, T>::value>>
  void operation(S &&);      // #3: universal version
};
```

149

Limited Greediness

- The resultant behavior is less surprising:

```
X<T> a;
a.operation(T());    // same: call #2, rvalue version
T const aT;
a.operation(aT);     // same: call #1, lvalue version
T anT;
a.operation(anT);    // change: call #1, was #3 universal
T *pT;
a.operation(pT);     // call #3, universal
```

150

Syntax Simplification

- Not everyone…appreciates the syntactic appearance of SFINAE, so a template typedef can help:

```
template <typename T>
class X {
public:
  ~~~
  template <typename R>
  using unlikeT = enable_if_t<!similar<R, T>::value>;
  template <typename S,
           typename = unlikeT<S>>
  void operation(S &&);      // #3: universal version
};
```

151

The Better Part of Valor Is Discretion?

- A better solution is often to avoid the situation entirely by avoiding the universal overload:

```
template <typename T>
class X {
public:
  void operation(T const &);  // #1: lvalue version
  void operation(T &&);       // #2: rvalue version
  template <typename S>
  void operation_prime(S &&); // #3: universal version
};
```

- This is the approach typically taken by the standard library.

152

Variadic Universals

- It's not uncommon to use variadic universal member functions.

```
template <typename T>
class X {
public:
    template <typename... Ts>
    void operation(Ts &&... args) {
        doit(forward<Ts>(args)...);
    }
    ~~~
};
```

- A typical implementation will perfect-forward the arguments to an appropriate implementation function.

153

A Typical Member Scenario

- A typical scenario is to overload one operation to enable move operations, and another as a variadic universal operation:

```
template <typename T>
class Cont {
public:
    void insert(T const &);          // for copying
    void insert(T &&);               // for moving
    template <typename... Ts>        // for in-place init
    void emplace(Ts &&... args);
    ~~~
};
```

✓ *Again: it's inadvisable to overload with a universal reference.*

154

You May Forward But Once

- When you forward an object, it may be moved.
- A forward is a conditional move.
- The same warnings apply as with an explicit move:

```
bool is_palindrome(string a);
bool is_perfect_pangram(string a);
~~~
template <typename T>
void func(T &&s) {
    if (is_perfect_pangram(forward<T>(s)) // was s moved?
    && is_palindrome(forward<T>(s)))    // what is s now?
        cout << "impossible!!!" << endl;
}
```

155

You May Forward But Once

- If you plan to use the value of an object multiple times, you should forward (or move) it only on its last use:

```
template <typename T>
void func(T &&s) {
    if (is_perfect_pangram(s)
        && is_palindrome(forward<T>(s)))
        cout << "impossible!!!" << endl;
}
~~~
func("Ma is as selfless as I am."); // OK, however...
func("Pa's a sap.");
```

156

Collected Advice

- *It's best not to disappoint reasonable programmers. Avoid ambiguous-appearing overloads.*
- *Prefer not to overload with universal references.*

157

Addendum: forward Implementation

- forward is implemented as an overloaded function template:

```
template <typename T>
constexpr T &&
forward(typename remove_reference<T>::type &t) noexcept
    { return static_cast<T &&>(t); }

template <typename T>
constexpr T &&
forward(typename remove_reference<T>::type &&t) noexcept
    { return static_cast<T &&>(t); }
```

- Any questions?

158

Dereferencing…

- Let's look first at that exciting argument declaration:

```
typename remove_reference<T>::type &t
```

- This use of `remove_reference` in the first overload ensures that the argument is always declared to be an lvalue reference.

```
typename remove_reference<T>::type &&t
```

- This use of `remove_reference` in the second overload ensures that the argument is always declared to be an rvalue reference.

159

`forward` Implementation

- Recall how `T` is deduced for an lvalue reference argument:

```
string msg ("Hello, World!");
dispatch(msg);                  // lvalue: T is string &
```

- In the call to `dispatch` with the lvalue, after argument deduction we have essentially:

```
void dispatch(string &arg) {   // string & && => string &
    ~~~
    doit(std::forward<string &>(arg));
}
```

160

forward Implementation

- The overloaded `forward` declarations, after substituting `string &` for `T` (and ignoring a few details!) look like:

```
string & &&
forward(string &t)
    { return static_cast<string & &&>(t); }

string & &&
forward(string &&t)
    { return static_cast<string & &&>(t); }
```

161

forward Implementation

- After reference collapsing, we have:

```
string &
forward(string &t)
    { return static_cast<string &>(t); }

string &
forward(string &&t)
    { return static_cast<string &>(t); }
```

- The `string &` argument to `forward` will match the first overload, the cast will have no effect, and the lvalue will remain an lvalue.
- As a result, the call is forwarded to the lvalue version of `doit`.

162

forward Implementation

- Recall how T is deduced for an rvalue reference argument:

```
dispatch(string("Hello, World!")); // rvalue: T is string
```

- In the call to dispatch with the rvalue, after argument deduction we have essentially:

```
void dispatch(string &&arg) {
    ~~~
    doit(std::forward<string>(arg));
}
```

163

forward Implementation

- The overloaded forward declarations, after substituting string for T (and ignoring a few details!) look like:

```
string &&
forward(string &t)
    { return static_cast<string &&>(t); }

string &&
forward(string &&t)
    { return static_cast<string &&>(t); }
```

- The string && argument to forward will match the second overload, and the lvalue will be cast to an rvalue.
- As a result, the call is forwarded to the rvalue version of doit.

164

Implementation of move

- After seeing the implementation of `std::forward`, the implementation of `std::move` is trivial:

```
template <typename T> constexpr
typename remove_reference<T>::type &&
move(T &&t) noexcept {
    using CT = typename remove_reference<T>::type &&;
    return static_cast<CT>(arg);
}
```

- In short:

✓ *`std::move` is an unconditional cast to an rvalue reference.*

✓ *`std::forward` is a conditional cast to an rvalue reference.*

165

166

Bibliography

- Abrahams [2010]. David Abrahams, Rani Sharoni, Doug Gregor, N3050=10-0040
- ISO [1998]. *ISO/IEC Standard 14882:1998, Programming languages—C++*.
- ISO [2003]. *ISO/IEC 14882:2003: Programming languages — C++*.
- ISO [2005]. *ISO/IEC TR 19768, C++ Library Extensions*.
- ISO [2011a]. *ISO/IEC 14882:2011: Programming languages — C++*.
- ISO [2011b]. *ISO/IEC 9899:2011: Programming languages — C*.
- ISO [2014]. *ISO/IEC Standard 14882:2014, Programming languages—C++*.

167

Bibliography

- Hinnant [2014]. Howard Hinnant, http://stackoverflow.com/questions/9322174/move-assignment-operator-and-if-this-rhs
- Karlsson [2004]. Bjorn Karlsson, "The Safe Bool Idiom". *The C++ Source*. www.artima.com/cppsource/safebool.html
- Meyers [2015]. Scott Meyers, Effective Modern C++. O'Reilly.
- Stroustrup [2013]. Bjarne Stroustrup, *The C++ Programming Language, 4th ed*. Addison-Wesley.
- Sutter [2013]. Herb Sutter, "GotW #94 Solution: AAA Style (Almost Always Auto)", *Sutter's Mill*. herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/

168

Bibliography

- Sutter [2013a]. Herb Sutter, "GotW #91: herbsutter.com/2013/06/05/gotw-91-solution-smart-pointer-parameters/
- Sutter [2014]. Herb Sutter, "Back to the Basics! Essentials of Modern C++ Style", *CppCon*. www.youtube.com/watch?v=xnqTKD8uD64
- Vandevoorde [2003]. David Vandevoorde and Nicolai Josuttis, *C++ Templates*. Addison-Wesley.

169

170

