Universal / Forwarding References

Nicolai M. Josuttis
josuttis.com
@NicoJosuttis

11/22

C++
©2022 by josuttis.com
1
josuttis | eckstein
IT communication

Nicolai M. Josuttis

- **Independent consultant**
  - Continuously learning since 1962
- **C++:**
  - since 1990
  - ISO Standard Committee since 1997
- **Other Topics:**
  - Systems Architect
  - Technical Manager
  - SOA
  - X and OSF/Motif

C++
©2022 by josuttis.com
communication

Modern C++

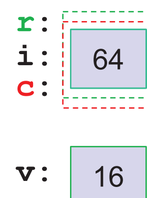
References

References

C++11

- **Reference**
  - ***Temporary additional name*** for an existing object/variable
  - Declared with **&**
    - Used as if declared without **&**
  - Always have to be initialized with the object it refers to
    - Cannot change the object to which it refers during lifetime

```
int i = 16;
int v = i;              // v is a new object
int& r = i;             // r refers to i
r *= 2;                 // modifies i

const int& c = i;       // c refers to i
c *= 2;                 // ERROR
i *= 2;                 // modifies r and c
```

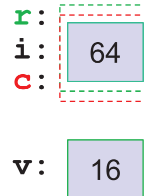


References with auto

C++11

- **Reference**
  - *Temporary additional name* for an existing object/variable
  - Declared with &
    - Used as if declared without &
  - Always have to be initialized with the object it refers to
    - Cannot change the object to which it refers during lifetime

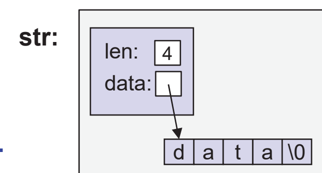
```
int i = 16;

auto v = i;             // v is a new int

auto& r = i;            // r has type int& and refers to i

r *= 2;                 // modifies i

const auto& c = i;      // c has type const int& and refers to i
c *= 2;                 // ERROR
i *= 2;                 // modifies r and c
```



Using Call by Reference

C++98

- **For better performance: Call by constant reference**
- **For out parameters: Call by non-constant reference**

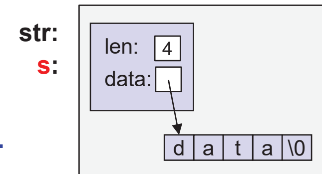



```
void print(const std::string& msg)  // msg refers to passed arg.
{                                   //    but can't modify it
  std::cout << msg << '\n';
}

void addQuotes(std::string& s)      // s refers to passed argument
{
  s = "'" + s + "'";      // calls + for std::string (passed literals are converted to std::string)
}

int main()
{
  std::string str = "data";
  addQuotes(str);         // str value: data => 'data'
  print(str);             // definition (implementation)
}
```

Using Call by Reference

C++98

- **For better performance:** Call by **constant reference**
- **For out parameters:** Call by **non-constant reference**




```
void print(const std::string& msg)  // msg refers to passed arg.
{                                   //    but can't modify it
  std::cout << msg << '\n';
}

void addQuotes(std::string& s)      // s refers to passed argument
{
  s = "'" + s + "'";       // calls + for std::string (passed literals are converted to std::string)
}

int main()
{
  std::string str = "data";
  addQuotes(str);          // str value: data => 'data'
  print(str);              // definition (implementation)
}
```

C++

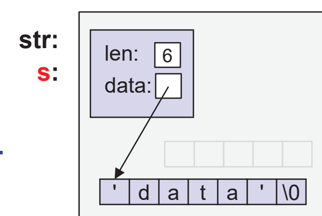
©2022 by josuttis.com

josuttis | eckstein
IT communication

Using Call by Reference

C++98

- **For better performance:** Call by **constant reference**
- **For out parameters:** Call by **non-constant reference**




```
void print(const std::string& msg)  // msg refers to passed arg.
{                                   //    but can't modify it
  std::cout << msg << '\n';
}

void addQuotes(std::string& s)      // s refers to passed argument
{
  s = "'" + s + "'";       // calls + for std::string (passed literals are converted to std::string)
}

int main()
{
  std::string str = "data";
  addQuotes(str);          // str value: data => 'data'
  print(str);              // definition (implementation)
}
```

C++

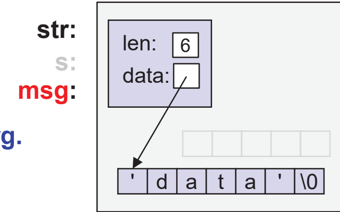
©2022 by josuttis.com

josuttis | eckstein
IT communication

Using Call by Reference

C++98

- **For better performance: Call by constant reference**
- **For out parameters: Call by non-constant reference**

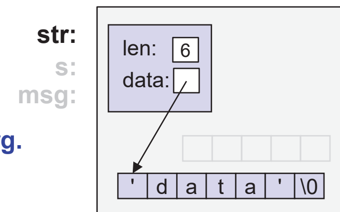



```
void print(const std::string& msg)  // msg refers to passed arg.
{                                    //    but can't modify it
  std::cout << msg << '\n';
}

void addQuotes(std::string& s)       // s refers to passed argument
{
  s = "'" + s + "'";      // calls + for std::string (passed literals are converted to std::string)
}

int main()
{
  std::string str = "data";
  addQuotes(str);         // str value: data => 'data'
  print(str);             // definition (implementation)
}
```


Using Call by Reference

C++98

- **For better performance: Call by constant reference**
- **For out parameters: Call by non-constant reference**

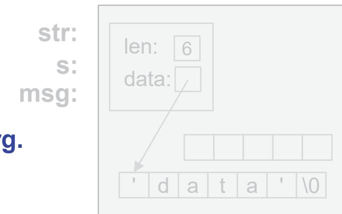



```
void print(const std::string& msg)  // msg refers to passed arg.
{                                    //    but can't modify it
  std::cout << msg << '\n';
}

void addQuotes(std::string& s)       // s refers to passed argument
{
  s = "'" + s + "'";      // calls + for std::string (passed literals are converted to std::string)
}

int main()
{
  std::string str = "data";
  addQuotes(str);         // str value: data => 'data'
  print(str);             // definition (implementation)
}
```

Using Call by Reference

C++98

- **For better performance:** **Call by constant reference**
- **For out parameters:** **Call by non-constant reference**




```
void print(const std::string& msg)  // msg refers to passed arg.
{                                   //    but can't modify it
  std::cout << msg << '\n';
}

void addQuotes(std::string& s)      // s refers to passed argument
{
  s = "'" + s + "'";     // calls + for std::string (passed literals are converted to std::string)
}

int main()
{
  std::string str = "data";
  addQuotes(str);        // str value: data => 'data'
  print(str);            // definition (implementation)
}
```

C++

©2022 by josuttis.com

josuttis | eckstein
IT communication

References Extend Lifetimes of Temporary Objects

C++98 / C++11

- **References can refer to temporary objects (rvalues)**
  - Extend the lifetime of what they directly refer to
  - They have to be `const` or rvalue references

```
std::string getVal();    // forward declaration
...
void foo()
{
  std::string s = getVal();             // OK: s is initialized with return value
  const std::string& r1 = getVal();     // OK: reference to temporary (as const)
  ...
  bar(r1);                              // OK: r1 extends lifetime of returned value
  std::string& r2 = getVal();           // ERROR: non-const reference to temporary
  std::string&& r3 = getVal();          // OK: reference to temporary (not const)
  ...
  bar(r3);                              // OK: r3 extends lifetime of returned value
} // return values, where references refers to, are destroyed here
```

C++

©2022 by josuttis.com

josuttis | eckstein
IT communication

References Extend Lifetimes of Temporary Objects

C++98 / C++11

- **auto References can refer to temporary objects (rvalues)**
  - Extend the lifetime of what they directly refer to
  - They have to be `const` or rvalue references

```
std::string getVal();    // forward declaration
…
void foo()
{
  auto s = getVal();                  // OK: s is initialized with return value
  const auto& r1 = getVal();          // OK: reference to temporary (as const)
  …
  bar(r1);                            // OK: r1 extends lifetime of returned value
  auto& r2 = getVal();                // ERROR: non-const reference to temporary
  auto&& r3 = getVal();               // OK: reference to temporary (not const)
  …
  bar(r3);                            // OK: r3 extends lifetime of returned value
} // return values, where references refers to, are destroyed here
```

This is special

C++

©2022 by josuttis.com

13

josuttis | eckstein
IT communication

Modern C++

RValue References

C++

©2022 by josuttis.com

14

josuttis | eckstein
IT communication

Vectors Without Move Semantics (C++03)

C++98

- **Containers have value semantics**
  - New elements are copied into the container
  - Passed arguments are not modified
- **This led to unnecessary copies with C++98/C++03**

```
template <typename T>
class vector {
  public:
    ...
    // copy elem into the vector:
    void push_back(const T& elem);
    ...
};
```

unnecessary copies in C++98 / C++03

```
std::vector<std::string> coll;
std::string s = getData();
...
coll.push_back(s);             // copy s into coll

coll.push_back(getData());     // copy temporary into coll

coll.push_back(s+s);           // copy temporary into coll

coll.push_back(s);             // copy s into coll again
                               // (no longer need s)
return coll;
```

C++
©2022 by josuttis.com

josuttis | eckstein
IT communication

Vectors With Move Semantics (C++11)

C++11

- **With rvalue references you can provide move semantics**
- **Rvalue references bind to rvalues**
  - Caller no longer needs the value
  - May *steal* but keep valid

```
template <typename T>
class vector {
  public:
    ...
    // copy elem into the vector:
    void push_back(const T& elem);
    ...
    // move elem into the vector:
    void push_back(T&& elem);
    ...
};
```

now named lvalue reference

declares rvalue reference

```
#include <utility>          // declares std::move()

std::vector<std::string> coll;
std::string s = getData();
...
coll.push_back(s);             // copy s into coll

coll.push_back(getData());     // move temporary into coll

coll.push_back(s+s);           // move temporary into coll

coll.push_back(std::move(s));  // move s into coll
                               // (no longer need s)
return coll;
```

C++
©2022 by josuttis.com

josuttis | eckstein
IT communication

Evolution of Value Categories

C / C++

K&R C:

```
int i;
i = 42;          // OK
42 = i;          // ERROR

int* p = &i;     // OK (for lvalue only)
int* q = &42;    // ERROR
```

`i` **is lvalue**:
OK on left-hand side of an assignment
`42` **is rvalue**:
only right-hand side of an assignment

ANSI C:

```
const int c = 42;
c = 77;              // ERROR (like rvalue)
const int* r = &c;   // OK    (like lvalue)
```

`c` **is lvalue**:
locator value
rvalue:
readable value

C++11:

```
std::string s;
std::move(s) = "hello";   // OK     (like lvalue)
auto sp = &std::move(s);  // ERROR  (like rvalue)
```

But for fundamental data types:
`std::move(i) = 42; // ERROR`

`std::move(...)` **is xvalue**:
- a bit like lvalue
- a bit like rvalue

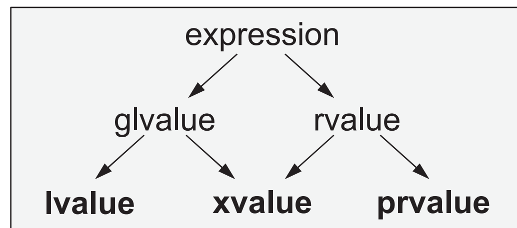
+ Most for rvalue applies to xvalue:
- Let's rename rvalue to prvalue
- and let rvalue represent both

C++
©2022 by josuttis.com
17
josuttis | eckstein
IT communication

Value Categories since C++11

C++11

C++11:



Every expression is one of:

- **LValue: Localizable value**
  - Variables, string literals, returned lvalue references, functions, all references to functions, data members of lvalues

  Everything that has a **name** and **string literals**

- **PRValue: Pure RValue** (former RValue)
  - All literals except string literals (**`42`**, **`true`**, **`nullptr`**,...), returned values (not references), results of constructor calls, lambdas, **`this`**

  Temporaries without name and **non-string literals**

- **XValue: eXpiring value**
  - Returned rvalue references (e.g., by **std::move()**), casts of objects (not of functions) to an rvalue reference, value members of rvalues

  Objects marked with std::move()

General categories: GLValue: Generalized **LV**alue, **RValue: R**eadable **V**alue

C++
©2022 by josuttis.com
18
josuttis | eckstein
IT communication

Overload Resolution on References

C++98

expression → glvalue, rvalue; glvalue → lvalue, xvalue; rvalue → xvalue, prvalue

Test program:

```
class Type;

Type v;
const Type c;

f(v);
f(c);
f(Type());
```

Declarations

Calls:	f(Type&)	f(const Type&)
f(v)	1	2
f(c)	no	1
f(Type())	no	2

modifiable lvalue

non-modifiable lvalue

rvalue => prvalue

f(Type&) has priority over f(const Type&)

C++

©2022 by josuttis.com

josuttis | eckstein IT communication

Overload Resolution on References

C++11

expression → glvalue, rvalue; glvalue → lvalue, xvalue; rvalue → xvalue, prvalue

Test program:

```
class Type;

Type v;
const Type c;

f(v);
f(c);
f(Type{});
f(std::move(v));
```

Declarations

Calls:	f(Type&)	f(const Type&)	f(Type&&)
f(v)	1	2	no
f(c)	no	1	no
f(Type{})	no	2	1
f(move(v))	no	2	1

modifiable lvalue

non-modifiable lvalue

prvalue

xvalue

copy is fallback for move

- *Rvalue reference*
- Can only bind to rvalues

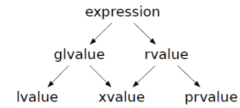
C++

©2022 by josuttis.com

josuttis | eckstein IT communication

Type versus Value Category

C++11




```
void fooRV (std::string&&);          // declaration
```

can only pass rvalues

```
std::string str;                     // str is used as lvalue
fooRV(str);                          // ERROR: cannot bind rvalue reference to lvalue
fooRV("hello");                      // OK (string literal (lvalue) converted to std::string (prvalue))
fooRV(std::string{"hello"});         // OK (pass prvalue)
fooRV(std::move(str));               // OK (pass xvalue)
```

```
void fooRV (std::string&& s)
{
  ...
  fooRV(s);                          // ERROR: cannot bind rvalue reference to lvalue
  fooRV(std::move(s));               // OK (pass xvalue)
}
```

s:
- has type `std::string&&`
- has name => is used as lvalue

`std::move()` casts `s` back to its type:
`static_cast<std::string&&>(s)`
which changes the value category to xvalue

- **Move semantics is not automatically passed through**

C++

©2022 by josuttis.com

josuttis | eckstein
IT communication

Overloading on References

C++11

- **`void foo(const Type&)`**
  - Pass value without creating a copy
  - Can bind to everything

read-only access
- in parameter to read

- **`void foo(Type&)`**
  - Pass named entity to return a value
  - Only non-`const` named object (lvalues)

write access
- (in)out parameter

- **`void foo(Type&&)`**
  - Pass value that is no longer needed
  - Only non-`const` objects without name or with `move()` (rvalues)

move access
- in parameter to adopt value

- `void foo(const Type&&)`
  - Possible, but semantic contradiction
  - Usually covered by `const Type&`

```
std::vector<std::string> coll;
...
const std::string s = getData();
...
coll.push_back(std::move(s));   // copies
```

Don't use `const` if you later move

C++

©2022 by josuttis.com

josuttis | eckstein
IT communication

Overloading on References

C++11

- **`void foo(const Type&)`**
  - Pass value without creating a copy
  - Can bind to everything
- **`void foo(Type&)`**
  - Pass named entity to return a value
  - Only non-**`const`** named object (lvalues)
- **`void foo(Type&&)`**
  - Pass value that is no longer needed
  - Only non-**`const`** objects without name or with **`move()`** (rvalues)
- `void foo(`**`const`** `Type&&)`
  - Possible, but semantic contradiction
  - Usually covered by **`const`** `Type&`

read-only access
- in parameter to read

write access
- (in)out parameter

move access
- in parameter to adopt value

```
std::vector<std::string> coll;
…
void insert(const std::string& s) {
  coll.push_back(std::move(s));   // copies
}
```


©2022 by josuttis.com
josuttis | eckstein
IT communication

Overloading on References

C++11

- **`void foo(const Type&)`**
  - Pass value without creating a copy
  - Can bind to everything
- **`void foo(Type&)`**
  - Pass named entity to return a value
  - Only non-**`const`** named object (lvalues)
- **`void foo(Type&&)`**
  - Pass value that is no longer needed
  - Only non-**`const`** objects without name or with **`move()`** (rvalues)
- `void foo(`**`const`** `Type&&)`
  - Possible, but semantic contradiction
  - Usually covered by **`const`** `Type&`

read-only access
- in parameter to read

write access
- (in)out parameter

move access
- in parameter to adopt value

```
const std::string getValue(); // forward decl.
…
std::vector<std::string> coll;
…
coll.push_back(getValue());   // copies
```

Don't use `const` when returning by value

©2022 by josuttis.com
josuttis | eckstein
IT communication

Modern C++

Universal References

Motivation for Perfect Forwarding 1/2

C++11

- **Overloading for `const`/non-`const` lvalues and rvalues:**

```
class C {
  …
};
void foo(const C&);      // read-only access (binds to all values)
void foo(C&);            // write access (binds to non-const lvalues)
void foo(C&&);           // move access (binds to non-const rvalues)
```

```
C v;
const C c;
foo(v);                  // calls foo(C&)
foo(c);                  // calls foo(const C&)
foo(C{});                // calls foo(C&&)
foo(std::move(v));       // calls foo(C&&)
```

Motivation for Perfect Forwarding 2/2

C++11

- **Forward move semantics in helper functions:**

```
class C {
  …
};

void foo(const C&);        // read-only access (binds to all values)
void foo(C&);              // write access (binds to non-const lvalues)
void foo(C&&);             // move access (binds to non-const rvalues)

void callFoo(const C& x) {
  foo(x);                  // x is const lvalue      => calls foo(const C&)
}
void callFoo(C& x) {
  foo(x);                  // x is non-const lvalue => calls foo(C&)
}
void callFoo(C&& x) {
  foo(std::move(x));       // x is non-const lvalue => needs std::move() to call foo(C&&)
}

C v;
const C c;
callFoo(v);                // calls foo(C&)
callFoo(c);                // calls foo(const C&)
callFoo(C{});              // calls foo(C&&)
callFoo(std::move(v));     // calls foo(C&&)
```

use `std::move()` to forward move semantics

C++

©2022 by josuttis.com

27

josuttis | eckstein
IT communication

Perfect Forwarding

C++11

- **Perfect forwarding of parameters:**
  1. Template parameter `T`
  2. Declare the parameter as `T&&` of the template parameter
  3. Pass with `std::forward<T>()`

```
void foo(const C&);        // read-only access (binds to all values)
void foo(C&);              // write access (binds to non-const lvalues)
void foo(C&&);             // move access (binds to non-const rvalues)

template<typename T>
void callFoo(T&& x)        // x is a universal (or forwarding) reference
{
  foo(std::forward<T>(x));    // perfectly forwards move semantics
}

C v;
const C c;
callFoo(v);                // foo(std::forward<T>(x)) => foo(x)
callFoo(c);                // foo(std::forward<T>(x)) => foo(x)
callFoo(C{});              // foo(std::forward<T>(x)) => foo(std::move(x))
callFoo(std::move(v));     // foo(std::forward<T>(x)) => foo(std::move(x))
```

Universal / forwarding reference
(community / C++ standard term)
- Can refer to `const` and non-`const`
- Can refer to rvalue and lvalue

`std::forward<>()` is `std::move()` only for rvalues

C++

©2022 by josuttis.com

28

josuttis | eckstein
IT communication

The Two Meanings of `&&` Declarations

C++11

- **&& declares**
  - For types: raw rvalue references
  - For template params: universal/forwarding references

```
class Type {
  …
};

void foo(Type&& x)      // rvalue reference
{
  std::is_const<Type>::value   // always false
  …
  // perfectly forward x:
  bar(std::move(x));
  // x has valid but unspecified state
}

Type v;
const Type c;

foo(v);                 // ERROR
foo(c);                 // ERROR
foo(Type{});            // OK
foo(std::move(v));      // OK
```

```
class Type {
  …
};

template<typename T>
void foo(T&& x)      // universal reference
{
  std::is_const<T>::value   // maybe true or false
  …
  // perfectly forward x:
  bar(std::forward<T>(x));
  // x has valid but unspecified state
}

Type v;
const Type c;

foo(v);               // OK, x is non-const
foo(c);               // OK, x is const
foo(Type{});          // OK, x is non-const
foo(std::move(v));    // OK, x is non-const
```

C++

©2022 by josuttis.com

josuttis | eckstein
IT communication

Universal/Forwarding References with Template Parameters

C++11

```
template<typename T>
void fooCR(const T& arg) { … }

template<typename T>
void fooR(T& arg) { … }

template<typename T>
void fooUR(T&& arg) { … }
```

```
std::string s{"some lvalue"};
std::string returnTmpString();
const std::string& returnConstStringRef();

fooCR(s);                          // OK, arg is const
fooCR(returnTmpString());          // OK, arg is const
fooCR(returnConstStringRef());     // OK, arg is const

fooR(s);                           // OK, arg is not const
fooR(returnTmpString());           // ERROR: cannot bind non-const lvalue reference to rvalue
fooR(returnConstStringRef());      // OK, arg is const

fooUR(s);                          // OK, arg is not const
fooUR(returnTmpString());          // OK, arg is not const
fooUR(returnConstStringRef());     // OK, arg is const
```

`const T&` can refer to everything **but makes `const`**

`T&` **cannot** refer to temporaries

`T&&` **can** refer to everything and **keeps non-constness**

C++

©2022 by josuttis.com

josuttis | eckstein
IT communication

References Extend Lifetimes of Temporary Objects

C++98 / C++11

- **`auto` References can refer to temporary objects (rvalues)**
  - Extend the lifetime of what they directly refer to
  - They have to be `const` or rvalue references

```
std::string getVal();    // forward declaration
…
void foo()
{
  auto s = getVal();                // OK: s is initialized with return value
  const auto& r1 = getVal();        // OK: reference to temporary (as const)
  …
  bar(r1);                          // OK: r1 extends lifetime of returned value
  auto& r2 = getVal();              // ERROR: non-const reference to temporary
  auto&& r3 = getVal();             // OK: reference to temporary (not const)
  …
  bar(r3);                          // OK: r3 extends lifetime of returned value
} // return values, where references refers to, are destroyed here
```

This is special

C++

©2022 by josuttis.com

josuttis | eckstein
IT communication

Universal/Forwarding References with `auto&&`

C++11

- **References that**
  - can *universally* refer to all expressions
    - Rvalues ((non-const) temporary objects and objects marked with `std::move()`)
    - Lvalues (named objects and string literals)
  - keep non-constness
  - may be used by move semantics for *perfect forwarding*

```
std::string s{"some lvalue"};
std::string returnTmpString();
const std::string& returnConstStringRef();

const auto& s1 = s;                        // OK, s1 is const
const auto& s2 = returnTmpString();        // OK, s2 is const
const auto& s3 = returnConstStringRef();   // OK, s3 is const

auto& s4 = s;                              // OK, s4 is not const
auto& s5 = returnTmpString();              // ERROR: cannot bind non-const lvalue reference to rvalue
auto& s6 = returnConstStringRef();         // OK, s6 is const

auto&& s7 = s;                             // OK, s7 is not const
auto&& s8 = returnTmpString();             // OK, s8 is not const
auto&& s9 = returnConstStringRef();        // OK, s9 is const
```

`const auto&` can refer to everything **but makes `const`**

`auto&` **cannot** refer to temporaries

`auto&&` **can** refer to everything and **keeps non-constness**

C++

©2022 by josuttis.com

josuttis | eckstein
IT communication

Modern C++

Universal References by Rule

Perfect Forwarding in Detail

C++11

- **Perfect forwarding of parameters:**
  1. Template parameter `T`
  2. Declare the parameter as `T&&` of the template parameter
  3. Pass with `std::forward<T>()`

```
void foo(const C&);    // read-only access (binds to all values)
void foo(C&);          // write access (binds to non-const lvalue)
void foo(C&&);         // move access (binds to non-const rvalue)

template <typename T>
void f(T&& x)          // x is universal/forwarding reference
{
  g(std::forward<T>(x));   // forwards move semantics
}                          // calls std::move() only for passed rvalues

C v;
const C c;

f(v);
f(c);
f(C{});
f(std::move(v));
```

	arg is:	T is:	x is:	forward<T>(x):
f(v);	lvalue			
f(c);	lvalue			
f(C{});	prvalue	C	C&&	
f(std::move(v));	xvalue	C	C&&	

Perfect Forwarding in Detail

C++11

- **Perfect forwarding of parameters:**
  1. Template parameter `T`
  2. Declare the parameter as `T&&` of the t
  3. Pass with `std::forward<T>()`

Rule in §14.8.2.1 [temp.deduct.call]:
If the parameter type is an rvalue reference to a cv-unqualified template parameter and the argument is an lvalue, the type "lvalue reference to T" is used in place of T for type deduction.

Collapsing rule in C++ §8.3.2 [dcl.ref]:

`Type& &`	becomes	`Type&`
`Type& &&`	becomes	`Type&`
`Type&& &`	becomes	`Type&`
`Type&& &&`	becomes	`Type&&`

```
void foo(const C&);        // read-only access (binds to all va
void foo(C&);              // write access (binds to non-const
void foo(C&&);             // move access (binds to non-con

template <typename T>
void f(T&& x)              // x is universal/forwarding reference
{
  g(std::forward<T>(x));   // forwards move semantics
}                          // calls std::move() only for passed rvalues

C v;
const C c;

f(v);
f(c);
f(C{});
f(std::move(v));
```

	arg is:	T is:	x is:	forward<T>(x):
`f(v);`	lvalue	`C&`	`C& && => C&`	`x`
`f(c);`	lvalue	`const C&`	`... => const C&`	`x`
`f(C{});`	prvalue	`C`	`C&&`	`std::move(x)`
`f(std::move(v));`	xvalue	`C`	`C&&`	`std::move(x)`

Type Deduction of Universal/Forwarding References

C++11

- **Universal** / forwarding **references**
  - deduce lvalue references for lvalues
  - Only rvalue become rvalue references

passed	T	x
`v`	`Type&`	`Type&`
`c`	`const Type&`	`const Type&`
`Type{}`	`Type`	`Type&&`
`move(v)`	`Type`	`Type&&`

```
class Type {
  ...
};

void foo(Type&& x)      // rvalue reference
{
  std::is_const<Type>::value    // always false
  ...
  // perfectly forward x:
  bar(std::move(x));
  // x has valid but unspecified state
}

Type v;
const Type c;

foo(v);                 // ERROR
foo(c);                 // ERROR
foo(Type{});            // OK
foo(std::move(v));      // OK
```

```
class Type {
  ...
};

template<typename T>
void foo(T&& x)         // universal reference
{
  std::is_const<T>::value    // maybe true or false
  ...
  // perfectly forward x:
  bar(std::forward<T>(x));
  // x has valid but unspecified state
}

Type v;
const Type c;

foo(v);                 // OK, x is non-const
foo(c);                 // OK, x is const
foo(Type{});            // OK, x is non-const
foo(std::move(v));      // OK, x is non-const
```

Universal/Forwarding Reference

C++11

- **Refer universally to any expression**
- **Declared as**
  - Template parameter with `&&`
  - `auto&&`
- **Preserves**
  - Type
  - Constness
    - `const` remains to be `const`
    - non-`const` remains to be non-`const`
  - Basic value category
    - LValues becomes lvalue reference (`&`)
    - RValues become rvalue reference (`&&`)

```
template<typename T>
foo(T&& x);
```

```
auto&& x = ...
```

	Passed/Initial Value	Type of `x`:
lvalue	`int i`	`int&`
lvalue	`const int ci`	`const int&`
rvalue	`42`	`int&&`
rvalue	`std::move(i)`	`int&&`

C++

©2022 by josuttis.com

josuttis | eckstein
IT communication

`std::move()` vs. `std::forward<>()`

C++11

- **std::move(arg)**
  - Converts to rvalue reference
  - aka:

```
static_cast<remove_reference_t<decltype(arg)>&&>(arg)
```

- **std::forward<T>(arg)**
  - converts to rvalue reference if rvalue (otherwise lvalue reference)
  - aka:

```
static_cast<decltype(arg)&&>(arg)
```

```
MyType val;
static_cast<MyType&&>      => MyType&&   (xvalue/rvalue)

MyType& val;
static_cast<MyType& &&>    => MyType&    (lvalue)

MyType&& val;
static_cast<MyType&& &&>   => MyType&&   (xvalue/rvalue)
```

C++

©2022 by josuttis.com

josuttis | eckstein
IT communication

Non-generic Universal References

C++20

- **How to specify universal references to a specific type?**

// When the type must match (no implicit conversions allowed):
```
template<typename T>
requires std::same_as<std::remove_cvref_t<T>, std::string>
void processString(T&&) {
  …
}
```

// When implicit conversions should be allowed:
```
template<typename T>
requires std::convertible_to<T, std::string>
void processString(T&&) {
  …
}
```

// When even explicit conversions should be allowed:
```
template<typename T>
requires std::constructible_from<std::string, T>
void processString(T&&) {
  …
}
```

C++
©2022 by josuttis.com

josuttis | eckstein
IT communication

Generic Pure RValue References

C++20

- **How to specify generic rvalue references for rvalues only?**

```
template<typename T>
requires (!std::is_lvalue_reference_v<T>)      // bind to rvalues only
void callFoo(T&& arg) {
  foo(std::forward<T>(arg));
}
```

// before C++20:
```
template<typename T,
         typename = typename std::enable_if<!std::is_lvalue_reference<T>::value
                                           >::type>
void callFoo(T&& arg) {
  foo(std::forward<T>(arg));
}
```

C++
©2022 by josuttis.com

josuttis | eckstein
IT communication

Modern C++

Perfect Passing and Returning

Perfect Returning with `decltype(auto)` (since C++14)

C++14

```
template<typename T>
auto callFoo(T&& arg)
{
  return foo(std::forward<T>(arg));
}
```

auto
- can't return a reference
- because it decays
(which usually is good)

```
template<typename T>
auto&& callFoo(T&& arg)
{
  return foo(std::forward<T>(arg));
}
```

Fatal runtime error
- might return reference to local object
(compilers may warn)

```
template<typename T>
decltype(auto) callFoo(T&& arg)      // since C++14
{
  return foo(std::forward<T>(arg));
}
```

returns perfectly
- **temporary by value**
- **reference by reference**

auto vs. auto&& vs. decltype(auto)

C++11 / C++14

	Type v; ... = v;	const Type c; ... = c;	Type& r{...}; ... = r;	... = Type();	Type v; ... = std::move(v);
auto a = ...	Type	Type	Type	Type	Type
auto& a = ...	Type&	const Type&	Type&	*invalid*	*invalid*
const auto& a = ...	const Type&	const Type&	const Type&	const Type&	const Type&
auto&& a = ...	Type&	const Type&	Type&	Type&&	Type&&
C++14: **decltype(auto)** a = ...	Type	const Type	Type&	Type	Type&&

`auto` **never is a reference** (decayed copy/move of whatsoever)
`auto&` **always is a reference** (valid only for lvalues)
`const auto&` **always is a reference** (refers to everything, but `const`)
`auto&&` **always is a reference** (universal/forwarding reference)
`decltype(auto)` **sometimes is a reference** (copy/move for temporaries)

C++
©2022 by josuttis.com

josuttis | eckstein
IT communication

Perfect Forwarding in Generic Code

C++14

- **Perfect forward parameters as `T&&`**
- **Perfect pass returned values as `auto&&`**
- **Perfect return returned values as `decltype(auto)`**
  - Take temporary value (prvalue) by value
  - Take reference (lvalue/xvalue) by reference

```
template<typename T>
decltype(auto) callFoo(T&& arg)
{
  auto&& val = foo1(std::forward<T>(arg));                 // perfect forward arg
  ...
  decltype(auto) ret{foo2(std::forward<decltype(val)>(val))}; // perfect pass val
  ...
  if constexpr (std::is_rvalue_reference_v<decltype(ret)>) {  // perfect return ret
    return std::move(ret);   // return returned value with move semantics
  }
  else {
    return ret;              // return plain value or lvalue reference
  }
}
```

`std::move(ret)` would disable named return value optimization
`(ret)` would cause `decltype(auto)` to return a reference

C++
©2022 by josuttis.com

josuttis | eckstein
IT communication

Perfect Forwarding in Generic Code

C++20

- **Perfect forward parameters as `T&&`**
- **Perfect pass returned values as `auto&&`**
- **Perfect return returned values as `decltype(auto)`**
  - Take temporary value (prvalue) by value
  - Take reference (lvalue/xvalue) by reference

```
decltype(auto) callFoo(auto&& arg)     // abbreviated function template syntax (since C++20)
{
  auto&& val = foo1(std::forward<decltype(arg)>(arg));          // perfect forward arg
  ...
  decltype(auto) ret{foo2(std::forward<decltype(val)>(val))}; // perfect pass val
  ...
  if constexpr (std::is_rvalue_reference_v<decltype(ret)>) {  // perfect return ret
    return std::move(ret);  // return returned value with move semantics
  }
  else {
    return ret;             // return plain value or lvalue reference
  }
}
```

`std::move(ret)` would disable named return value optimization
`(ret)` would cause `decltype(auto)` to return a reference

©2022 by josuttis.com

Modern C++

Universal References in the Range-based `for` Loop

©2022 by josuttis.com

Range-Based for Loop

C++11

- **The expression:**

```
for ( decl : coll ) {
  statement
}
```

```
// print all elements of coll:
for (const auto& elem : getColl()) {
  std::cout << elem << '\n';
}
```

is in fact equivalent to:

```
{
  reference rg = coll;    // init a reference with coll to be able to use it twice
  for (auto pos = rg.begin(), pend = rg.end(); pos != pend; ++pos) {
    decl = *pos;
    statement
  }
}
```

```
{
  reference rg = getColl();
  for (auto pos = rg.begin(), pend = rg.end(); pos != pend; ++pos) {
    const auto& elem = *pos;
    std::cout << elem << '\n';
  }
}
```

A reference extends the lifetime of to where it refers (return value of `getColl()`)

C++

©2022 by josuttis.com

josuttis | eckstein IT communication

Range-Based for Loop Intern

C++11

- **The expression:**

```
for ( decl : coll ) {
  statement
}
```

```
// print all elements of coll:
for (const auto& elem : getColl()) {
  std::cout << elem << '\n';
}
```

is in fact equivalent to:

```
{
  auto&& rg = coll;    // init a reference with coll to be able to use it twice
  for (auto pos = rg.begin(), pend = rg.end(); pos != pend; ++pos) {
    decl = *pos;
    statement
  }
}
```

```
{
  auto&& rg = getColl();
  for (auto pos = rg.begin(), pend = rg.end(); pos != pend; ++pos) {
    const auto& elem = *pos;
    std::cout << elem << '\n';
  }
}
```

Can iterate over everything without making it `const`

`auto`	would create a copy
`const auto&`	would make it const
`auto&`	would not compile

C++

©2022 by josuttis.com

josuttis | eckstein IT communication

Using the Range-based for Loop

C++11

```
template<typename Coll, typename T>
void setAllElemsTo(Coll& coll, const T& value)
{
  for (auto& elem : coll) {
    elem = value;
  }
}

std::vector<int> coll1{0, 8, 15};
…
setAllElemsTo(coll1, 42);      // OK

std::vector<bool> collB{false, true, false};
…
setAllElemsTo(collB, true);    // ERROR
```

```
namespace std {
 template<…>
 class vector<bool, …> {
  public:
   …
   class reference {
     …
   };
   …
 };
}
```

The reference in `vector<bool>` is an **object**

```
{
  auto&& rg = coll;
  for (auto pos = rg.begin(), end = rg.end(); pos != end; ++pos) {
    auto& elem = *pos;    // *pos yields a reference to the element
    elem = value;
  }
}
```

C++ josuttis | eckstein IT communication

©2022 by josuttis.com

49

Using the Range-based for Loop

C++11

```
template<typename Coll, typename T>
void setAllElemsTo(Coll& coll, const T& value)
{
  for (auto&& elem : coll) {
    elem = value;
  }
}

std::vector<int> coll1{0, 8, 15};
…
setAllElemsTo(coll1, 42);      // OK

std::vector<bool> collB{false, true, false};
…
setAllElemsTo(collB, true);    // OK
```

```
namespace std {
 template<…>
 class vector<bool, …> {
  public:
   …
   class reference {
     …
   };
   …
 };
}
```

The reference in `vector<bool>` is an **object**

```
{
  auto&& rg = coll;
  for (auto pos = rg.begin(), end = rg.end(); pos != end; ++pos) {
    auto&& elem = *pos;    // *pos yields a reference to the element
    elem = value;
  }
}
```

C++ josuttis | eckstein IT communication

©2022 by josuttis.com

50

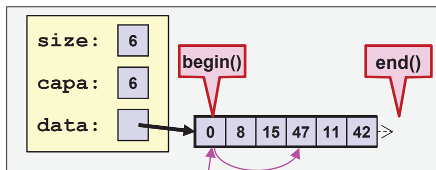
Modern C++

Universal References for Views

Views that Cache `begin()`

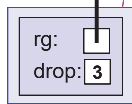
C++20

coll1:



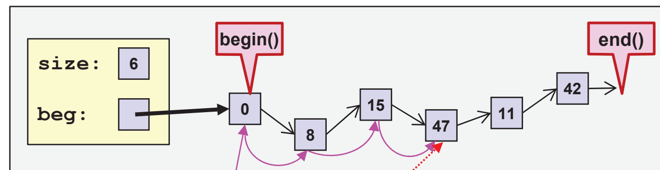
begin()
+= 3

v1:



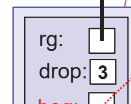
begin()

coll2:



begin()
++
++
++

v2:



begin()

Views that cache **`begin()`**:

Always: `filter`, `drop_while`, `split`

Sometimes: `drop`, `reverse`

```
std::vector<int> coll1{0, 8, 15, 47, 11, 42};
auto v1 = coll1 | std::views::drop(3);
auto pos = v1.begin();

std::list<int> coll2{0, 8, 15, 47, 11, 42};
auto v2 = coll2 | std::views::drop(3);
auto pos = v2.begin();
```

Processing Containers and Views

C++20

```
void print(const auto& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

std::vector<int> coll1{0, 8, 15, 47, 11, 42};
std::list<int> coll2{0, 8, 15, 47, 11, 42};

print(coll1);
print(coll2);
print(coll1 | std::views::take(3));         // print first three elements
print(coll2 | std::views::take(3));         // print first three elements
print(coll1 | std::views::drop(3));         // print fourth to last element
print(coll2 | std::views::drop(3));         // Compile-time ERROR
for (int v : coll2 | std::views::drop(3)) { // OK: print fourth to last element
   std::cout << v << ' ';
}
```

Output:

```
0 8 15 47 11 42
0 8 15 47 11 42
0 8 15
0 8 15
47 11 42
ERROR
47 11 42
```

Some views (may) cache on iteration to make a second `begin()` cheap.
You can't iterate when such a view is `const`.

Passing Containers and Views by Universal Reference

C++20

```
void print(auto&& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

std::vector<int> coll1{0, 8, 15, 47, 11, 42};
std::list<int> coll2{0, 8, 15, 47, 11, 42};

print(coll1);
print(coll2);
print(coll1 | std::views::take(3));         // print first three elements
print(coll2 | std::views::take(3));         // print first three elements
print(coll1 | std::views::drop(3));         // print fourth to last element
print(coll2 | std::views::drop(3));         // OK: print fourth to last element
for (int v : coll2 | std::views::drop(3)) { // OK: print fourth to last element
   std::cout << v << ' ';
}
```

Universal (or *forwarding*) ***reference***

- Can universally refer to every expression (even temporaries/rvalues) without making it `const`

Output:

```
0 8 15 47 11 42
0 8 15 47 11 42
0 8 15
0 8 15
47 11 42
47 11 42
47 11 42
```

Summary

- **Universal/forwarding References are used for**
  - Perfect forwarding
  - Universally referring to expressions without making them const
    - Necessary for **generic code taking containers *and* C++20 views**
      - There is **no way** to declare a **generic function** that takes
        - both **const** and **non-const**
        - both **containers** and **views**

        and **guarantee** by the signature **not to modify the elements**
- **Open issues:**
  - Terminology:
    - ***Universal*** or ***forwarding***
      (a mess introduced by the C++ standard committee)
  - Teaching:
    - When to introduce universal/forwarding reference to beginners?
  - Application:
    - When to use universal/forwarding references?

Thank You!

Nicolai M. Josuttis

www.josuttis.com
nico@josuttis.com
@NicoJosuttis

- Why refs
- Why universal/forwarding refs (from move)
- Categories and binding rules
- name chaos
- different purpose
- universal or not?
- why now for views?

Abstract

- Universal/forwarding references were introduced in C++11 mainly for perfect forwarding of move semantics.
- However, over time they became more and more a key feature in various situations of C++. One example is the implementation for the range-based for loop.
- With C++20 they even get an important for ordinary application programmers. You need them to be able to implement generic code that works for both containers and views.
- Time to look carefully at T&&, auto&& (and maybe their cousin decltype(auto)) in detail.