C++ London University Session 24

Tristan Brindle

Feedback and Communication

- Your feedback is vital
- Otherwise, we don’t know what you don’t know!
- Please join the #ug_uk_cpplondonuni channel on the cpplang Slack — Go to https://cpplang.now.sh/ for an “invitation”

Today's Lesson Plan

- Introduction to move semantics and perfect forwarding
  - Motivation
  - Value categories
  - r-value references and `std::move`
  - Move constructors and assignment operators
  - Forwarding ("universal") references and `std::forward`

Why Move Semantics?

- R-value references were introduced in C++11 to solve two different but related problems:
  - Efficiently “stealing” the contents of an object whose lifetime is about to end
  - Writing generic code which needs to pass a value to another function with the same semantics (“perfect forwarding”)
- We’re mostly going to talk about the former today

Copy and Move

- C++11 introduced a new fundamental operation to the language: *moving an object*
- Moving is related to copying, but potentially changes the source object
- Moving is sometimes described as an optimisation of copying, but it is better to think of it as a separate operation
- All copyable objects can be moved, but not all moveable objects can be copied!

Copy vs Move

- *Copying* an object gives the destination the same value as the source, and *leaves the source object unchanged*

- For example:

```
std::vector<int> a{1, 2, 3};
std::vector<int> b;

b = a; // a and b both contain {1, 2, 3}
```

Copy vs Move

- *Moving* an object gives the destination the same value as the source, but potentially modifies the source object

- For example:

```
std::vector<int> a{1, 2, 3};
std::vector<int> b;

b = std::move(a); // b contains {1, 2, 3}
                  // a is in a "moved-from" state
```

The Mechanics of Moving

- This section is all about what move actually does behind the scenes
- It’s not essential to know this in order to use move operations
- But it will give you a better understanding of why move semantics are used

Statements and Expressions

- A C++ program is made up of *statements* and *expressions*
- Statements are things like function and class declarations, `if` statements, `for` loops etc
- Expressions are things like

```
x // identity expression
42 // literal expression
x = 3 // assignment expression
x + y // additive expression
foo(x, y) // function call expression
x < 10 ? x += 24 : y = foo(y, x * x)  // ternary expression
```

Value Categories in C++98

- The most important property of an expression is its *type*: every expression has a type

- Historically, the other important property of an expression was whether it referred to a named object in memory or not. This was called its *value category*

- For example,

```
int x = 0, y = 0;

x; // refers to object
42; // does not refer to object
x + y; // does not refer to object
```

- Expressions which referred to an object were called *lvalues*; expressions which referred to a temporary were called *rvalues*

Value Categories in C++98

Refers to object	**Temporary**
lvalue	rvalue

Value Categories in C++11

- C++11 added a second important property of an expression: whether it can be *moved from*

- *rvalues* can always be moved from (otherwise we cannot do anything with them!)

- “lvalues” can *sometimes* be moved from

Value Categories

	Refers to object	Temporary
Cannot be moved from	???	-
May be moved from	???	???

Value Categories

	Refers to object	Temporary
Cannot be moved from	*lvalue*	-
May be moved from	*xvalue*	*prvalue*

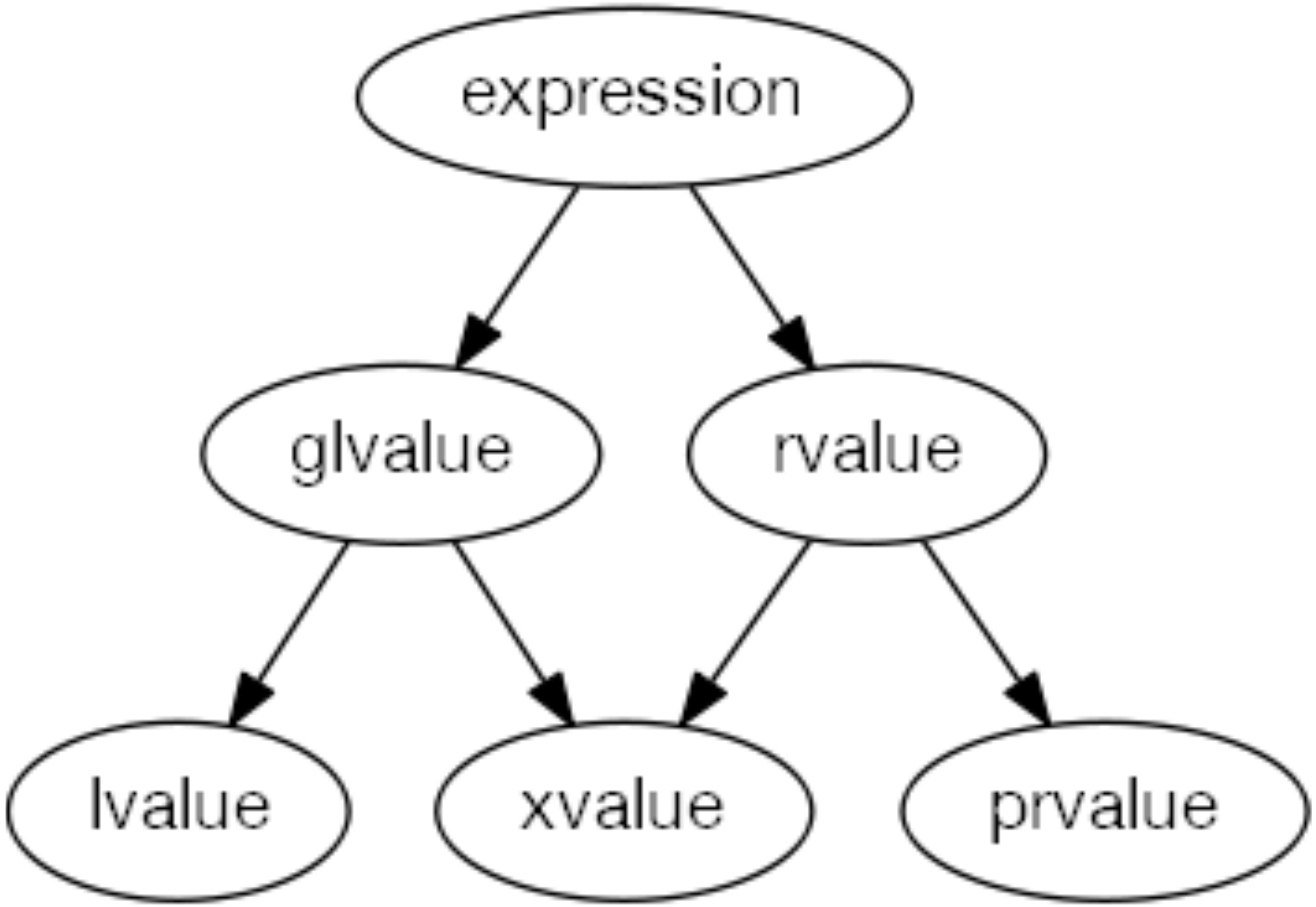
Value Categories

	glvalue	
	lvalue	-
rvalue	*xvalue*	*prvalue*

Value Categories

- C++11 has three value categories: *lvalue*, *xvalue* and *prvalue*
- **Every expression has precisely one of these three categories**
- There are also two “umbrella categories”: *glvalue* and *rvalue*

Value Categories



Know your value categories

- If it has a name, it's an *lvalue*
- If you can take its address, it's an *lvalue*
- If it's an object returned from a function, it's a *prvalue*
- `std::move()` is used to turn *lvalue*s into *xvalue*s

Value categories and references

- We can bind an ordinary reference to an lvalue, for example:

```
int x = 0;

int& r = x;
const int& cr = x;
```

- We can also bind a const reference to an rvalue, for example:

```
int get_int();

const int& cr = get_int();
```

- We **cannot** bind a non-const reference to an rvalue:

```
int get_int();

int& cr = get_int(); // ERROR!
```

rvalue references

- C++11 added *rvalue references*, written `Type&&`
- Ordinary references we renamed *lvalue references*
- We can bind an rvalue reference to an rvalue, for example:

```
int get_int();
int x = 0;

int&& rr = get_int();
int&& rm = std::move(x);
int&& rl = 42;
```

- We cannot bind an rvalue reference to an lvalue, for example:

```
int x = 0;

int&& ri = x; // ERROR
```

rvalue references and overloading

- If a function takes an *rvalue reference* parameter, we can only pass an rvalue
- If a function takes a *non-const lvalue reference* parameter, we can only pass an lvalue
- If a function takes a ***const*** *lvalue reference* parameter, we can pass an lvalue or an rvalue
- However, an rvalue reference version (if present) will **always be preferred** when called with an rvalue argument

rvalue reference overloading: examples

```
void insert_into_vector(vector<string>& vec, const string& str)
{
    vec.push_back(str); // copies str
}


int main()
{
    vector<string> vec;
    string s = "Hello world";

    insert_into_vector(vec, s); // Ok
    insert_into_vector(vec, std::string("goodbye world")); // Ok
}
```

rvalue reference overloading: examples

```
void insert_into_vector(vector<string>& vec, const string& str)
{
    vec.push_back(str); // copies str
}

void insert_into_vector(vector<string>& vec, string&& str)
{
    vec.push_back(std::move(str)); // moves str
}

int main()
{
    vector<string> vec;
    string s = "Hello world";

    insert_into_vector(vec, s);
    insert_into_vector(vec, std::move(s));
    insert_into_vector(vec, std::string("goodbye world"));
}
```

Move construction

- The rules for passing rvalue references to functions apply to constructors and assignment operators as well
- This allows us to define *move constructors* and *move-assignment* operators for our classes
- These complement the copy constructor and copy-assignment operators

Move Construction

- Live demo!

Move Construction

- Remember the “rule of five”: if you write a class with any of
  - Custom destructor
  - Custom copy constructor
  - Custom copy-assignment operator
  - Custom move constructor
  - Custom move-assignment operator
- Then you probably need to write **all five** of these functions

Move Construction

- If a class has no *user-declared* copy constructor, copy-assignment operator or destructor, then the compiler will automatically generate the move operations
- The compiler-generated version will move each member variable in declaration order
- Declaring a move constructor or move-assignment operator will stop the compiler from generating copy operations
- This can be used to create move-only types, for example std::unique_ptr
- The “rule of zero” is best: if possible, avoid defining any of the special member functions yourself!

Forwarding References

- *Template argument deduction* is the process by which the compiler will try to determine the template arguments of a call to a function template. For example:

```
template <typename T>
void foo(T& t);

int main()
{
    int i = 0;
    foo(i); // T is deduced as int

    const int ci = 0;
    foo(ci); // T is deduced as const int
}
```

Forwarding References

- If an argument to a function template has the form T&&, where T is a template parameter, then some special deduction rules are used:
  - If the passed argument is an *lvalue* of type A, then T is deduced as A&, and the function argument type is A&
  - If the passed argument is an *rvalue* of type A, then T is deduced as A, and the function argument type is A&&
- An argument of the type T&& is called a *forwarding reference*, sometimes called a *universal reference*

Forwarding References

```
template <typename T>
void foo(T&& t);

int main()
{
    int i = 0;
    foo(i); // T is deduced as int&, t is int&

    const int ci = 0;
    foo(ci); // T is deduced as const int&, t is const int&

    foo(std::move(i)); // T is deduced as int, t is int&&
}
```

std::forward

- Remember, within a function body, the function arguments are *lvalues* — they have names
- To *perfectly forward* a function argument, we need to restore the value category it originally had
- The standard library function std::forward<T>(), used with forwarding references, can do this for us

std::forward example

```
void foo(int&){} // lvalue overload

void foo(int&&) {} // rvalue overload

template <typename T>
void non_forwarding(T&& t)
{
    foo(t);
}

template <typename T>
void forwarding(T&& t)
{
    foo(std::forward<T>(t));
}

int main()
{
    int i = 0;

    non_forwarding(i); // calls foo(int&) -> bar(int&), good
    non_forwarding(std::move(i)); // calls foo(int&&) -> bar(int&), bad!

    forwarding(i); // calls foo(int&) -> bar(int&), good
    forwarding(std::move(i)); // calls foo(int&&) -> bar(int&&), good
}
```

Move Sematics Tips

- Always declare move operations if you define copy
- Use `std::move()` to move an object unconditionally
- Use `std::forward()` to move an object that is a forwarding reference
- A moved-from object is in a “valid but undefined” state: in general, it must be re-initialised (assigned to) or destroyed
- Don’t `std::move()` an object in a `return` statement — this inhibits copy elision
- Use `auto&&` (sparingly) if you want forwarding reference rules applied to a variable declaration

Online Resources

- https://isocpp.org/get-started
- cppreference.com — The bible, but aimed at experts
- cplusplus.com — Another reference site, also has a tutorial section
- learncpp.com — Free online tutorial, very up-to-date
- https://www.pluralsight.com/authors/kate-gregory - Comprehensive set of courses from an experienced C++ trainer (free trial)
- reddit.com/r/cpp_questions
- Cpplang Slack channel — https://cpplang.now.sh/ for an “invite”
- StackOverflow (but…)

Thanks for coming!

C++ London University:

- Website: cpplondonuni.com
- Twitter: @cpplondonuni
- Github: github.com/CPPLondonUni
- Reddit: reddit.com/r/CppLondonUni

Where to find Tom Breza:

- On Slack: cpplang.slack.com #learn #ug_uk_cpplondonuni
- E-mail: tom@PCServiceGroup.co.uk
- Mobile: 07947451167

My stuff:

- Website: tristanbrindle.com
- Twitter: @tristanbrindle
- Github: github.com/tcbrindle

See you next time! 🙂