

# Move semantics

Part 1: Value categories, move and  
(some types of) copy elision

# Index

- Prerequisites
  - Basics of move semantics
  - Basics of copy elision
  - What's an expression
- Value categories
  - Brief introduction
  - Identifying categories
  - Summarizing
- Guaranteed copy elision more in depth
- Tips for calls to `std::move`
- Questions
- References

# Basics of move semantics

- Moving an object A to an object B means that B will have the data of A and that A will be in a valid but unspecified state
  - Methods of A can be called (mainly its destructor) but you don't know which data it contains

```
std::string otherStr{ "bonjour les gens" };  
std::string str{ std::move(otherStr) };
```

- Now “str” contains the string “bonjour les gens” and “otherStr” contains something unspecified
  - But most of the time “otherStr” will contains an empty string because the content of “str” and “otherStr” was swaped

# Basics of copy elision

- Sometimes compilers can elide copies (and moves) to optimize the code
  - In C++17 some copy-elisions are mandatory

```
std::pair object = std::pair{ 1, '2' };
```

Without copy elision:

- The right object is constructed with “1” and “2”
- The left object is move-constructed with the right object

With copy elision:

- The left object is constructed with “1” and “2”
- That's all

# What's an expression

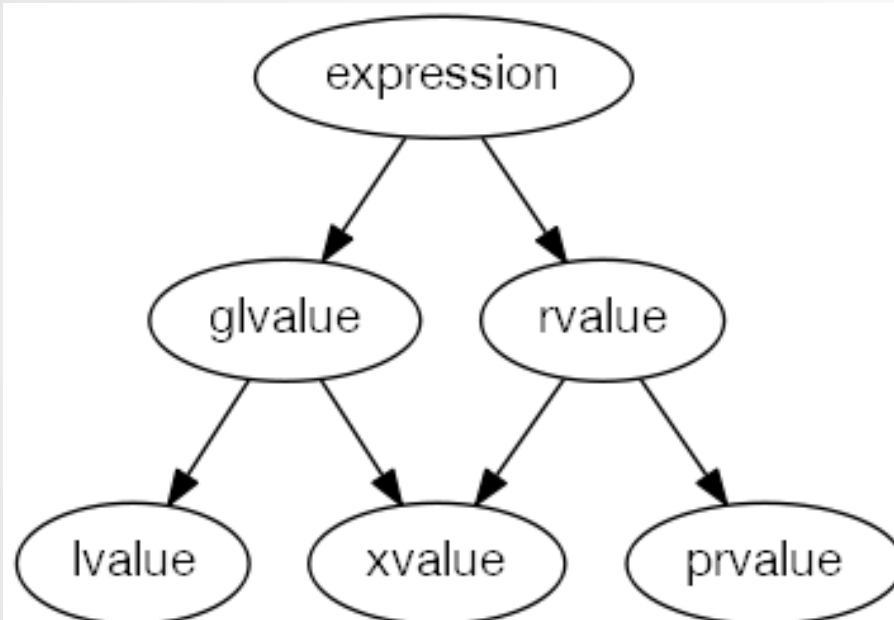
- An expression is a computation that may produce a result
  - If it doesn't return a result, it's called a void expression
- Expressions refer to entities, some of these entities are:
  - Values: data not stored in the memory, used to build objects
  - Objects / subobjects: data stored in the memory
  - References: refer an object

```
std::string str1, str2 /* not an expression */;
str1 /* expression refers to object str1 */;
str1 += str2 /* expression refers to object str1
              through an unnamed lvalue reference */;
str1 + str2 /* expression refers to an unnamed object */;
5 /* expression refers to a value */;
std::string{ "str" } /* expression refers to a value */;
```

# Value categories: disclaimers

- What is described here is what the standard say for C++17
- For C++14, if copy elision is not disabled everything will be mostly the same, some things may be described differently in the standard but at the end the binary generated will be the same
- If copy elision is disabled the things explained here won't be 100 % accurate, but because it should never be disabled, and because in C++17 it won't be possible to disable it, i chose to not talk about how it will work in this configuration

# Value categories



Types of data transfer of expressions :

lvalue: copy

xvalue: copy / move

prvalue: copy / move / copy elision

# Identifying categories

- Has an identity
  - You can compare two entities to know if they are the same
  - Basically you can know their address
- Can be moved from
  - Can be used by a move constructor / assignment
  - Basically can be bind to an rvalue reference



# Identifying lvalues

- Has an identity but can't be moved from
  - You can apply the unary & operator on it
  - You can't pass it to a move constructor / assignment
- It's an lvalue if it refers to (not exhaustive):
  - An lvalue reference or a named rvalue reference
  - An object that isn't a subobject of an rvalue
    - Static data members are complete objects
- The expression of a function that return an lvalue reference is an lvalue

# Identifying lvalues

```
std::string obj;  
obj;           // lvalue  
  
std::string& function();  
function();    // lvalue  
  
std::string&& rvalRef = std::move(obj);  
rvalRef;       // lvalue
```

# Identifying xvalues

- Has an identity and can be moved from
  - You can't apply the unary & operator on it but can have the address from the object it refers
  - You can move construct / assign from it
- Xvalue means eXpiring value, they refer to objects that you can move but that can still be accessed from elsewhere, like:
  - Unnamed rvalue references
  - Subobjects of rvalues
    - Static data members are complete objects
- The expression of a function that return an rvalue reference is an xvalue

# Identifying xvalues

```
std::string obj;  
std::move(obj);           // xvalue  
  
std::pair<int, int>{}.first; // xvalue  
  
std::pair<int, int> pair;  
std::move(pair).first;     // xvalue
```

# Identifying prvalues

- Don't have an identity but can be moved from
  - You can't get the address of the entity referred by a prvalue or deduce it from another entity
  - You can move construct / assign from it
- If it's neither an lvalue or an xvalue, it's a prvalue
  - That means prvalues don't refer to objects or references, they refer to values (and some others entities)
- The expression of a function that don't return a reference is a prvalue

# Identifying prvalues

```
std::string{};           // prvalue  
  
std::string otherFunction();  
otherFunction();         // prvalue
```

# Summarizing value categories

- Lvalues refer to objects that can still be used after the expression, they exist “physically” in the memory and must keep their “data” until they are destroyed
- Xvalues refer to objects that are expiring, they exist “physically” in the memory but they won’t be used anymore so their “data” can be moved to another object
- Prvalues don’t refer to objects, they don’t exist “physically” in the memory, they don’t have an address, they are accessible only in the expression that created them, so their “data” can safely be moved somewhere else

# Guaranteed copy elision (C++17)

- Copy elision is linked to prvalues
- Prvalues don't refer to “physical” object, but to values that are used to construct objects
- When an object is needed for computation, a temporary can be materialized from a prvalue
  - That means an xvalue is produced from the prvalue
- All of this means that passing prvalues is free because there is no object to move, the compiler just wait until a temporary is materialized to see where to build it
  - But in fact the object is still constructed when the prvalue is created, it's just not constructed where the prvalue was created



# Guaranteed copy elision (RVO)

```
VerboseClass iBuildStuff()  
{  
    return VerboseClass{};  
}  
  
void basicExemple()  
{  
    VerboseClass imStuff = iBuildStuff();  
}
```

-fno-elide-constructors --std=c++14

```
VerboseClass default constructor.  
VerboseClass move constructor.  
VerboseClass move constructor.
```

--std=c++14

```
VerboseClass default constructor.
```

# Guaranteed copy elision (RVO)

```
VerboseClass iBuildMoreStuff()
{
    VerboseClass notStuff{ 21 };
    return VerboseClass{ 25 };
}

void advancedExemple()
{
    VerboseClass alsoNotStuff{ 11 };
    VerboseClass imStuff = iBuildMoreStuff();
    VerboseClass stillNotStuff{ 12 };
}
```

-fno-elide-constructors --std=c++14

```
VerboseClass arg constructor. (11: 0x7fff1b0f1168)
VerboseClass arg constructor. (21: 0x7fff1b0f1130)
VerboseClass arg constructor. (25: 0x7fff1b0f1128)
VerboseClass move constructor. (25: 0x7fff1b0f1158)
VerboseClass move constructor. (25: 0x7fff1b0f1160)
VerboseClass arg constructor. (12: 0x7fff1b0f1150)
```

--std=c++14

```
VerboseClass arg constructor. (11: 0x7ffc39e3d878)
VerboseClass arg constructor. (21: 0x7ffc39e3d840)
VerboseClass arg constructor. (25: 0x7ffc39e3d870)
VerboseClass arg constructor. (12: 0x7ffc39e3d868)
```

# When to use `std::move`

- Lvalues:
  - Move them whenever you can, it will prevent some copies to be made, making the software faster
  - Moving const objects is useless because for the move constructor / assignment to work the object need to be non-const
- Xvalues:
  - It's useless to move them because it will just convert them into xvalues, what they already are
- Prvalues:
  - Don't move them because it will prevent copy elision, forcing objects to be moved so making the software slower

# Questions

```
training.getAttendees()  
| std::views::filter([](const auto& attendee) -> bool { return attendee.haveQuestion(); })  
| std::ranges::for_each([&training](const auto& attendee) { training.getTrainer().answer(attendee.getQuestions()); });
```

# What's next

- In the second part we will see:
  - Forwarding references
  - `std::forward`
  - Why this code is bad:

```
std::vector<int> badFunction()
{
    std::vector<int> bigVector(816);
    return std::move(bigVector);
}
```

# References

- [https://en.cppreference.com/w/cpp/language/value\\_category](https://en.cppreference.com/w/cpp/language/value_category)
- <https://stackoverflow.com/questions/3601602/what-are-rvalues-lvalues-xvalues-glvalues-and-prvalues>
- [https://en.cppreference.com/w/cpp/language/implicit\\_conversion#Temporary\\_materialization](https://en.cppreference.com/w/cpp/language/implicit_conversion#Temporary_materialization)
- [https://en.cppreference.com/w/cpp/language/copy\\_elision](https://en.cppreference.com/w/cpp/language/copy_elision)
- <https://en.cppreference.com/w/cpp/language/return#Notes>
- <https://stackoverflow.com/questions/3106110/what-is-move-semantics>