References

- 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/copy_elision
- https://en.cppreference.com/w/cpp/language/return#Notes
- https://stackoverflow.com/questions/3106110/what-is-move-semantics

Disclaimer

- 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 stuff talked 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

Basics of the move semantic

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

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

With copy elision:

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

Without copy elision:

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

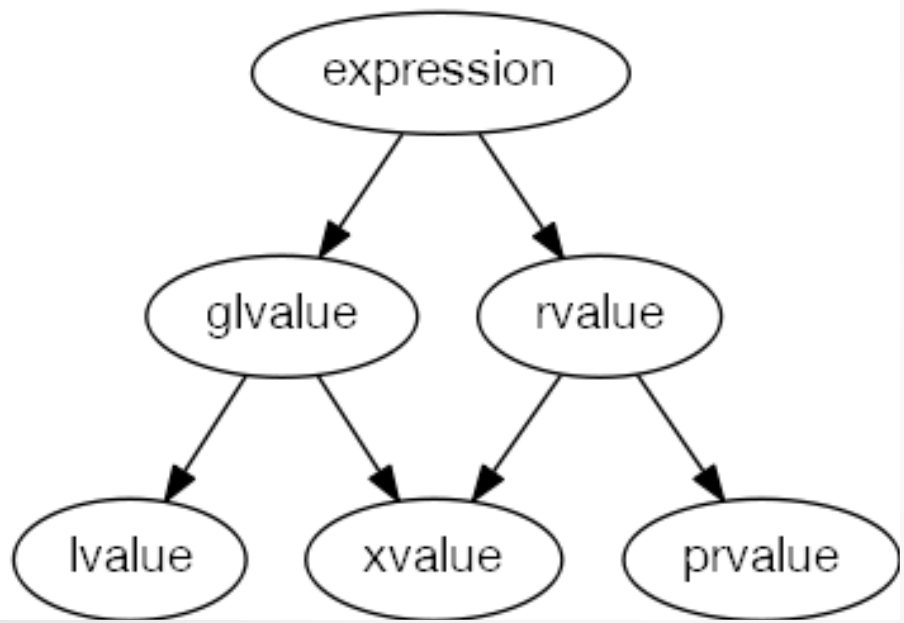
Warning : Type of types

- Type of an object is not the same thing as type of an expression

```
std::string otherStr;
std::string& str = otherStr;
```

- The object “str” is of type “lvalue reference to string”
- The expression “(str)” is an lvalue of type “string”

Value categories



Types of resources transfert of expressions :

lvalue: copy
xvalue: copy / move
prvalue: copy / move / copy elision

Identifying categories

- Has an identity
  - You can compare two objects to know if they are the same
  - Basically you can know his address
- Can be moved from
  - Can be used by a move constructor / assignment
  - Basically can be bind to a 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
- Most of the time if it has a name it is an lvalue
  - That mean named rvalue references are lvalues
  - Except for (not exhaustive):
    - Non-static data members of rvalue objects
    - Non-static members functions
- 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 original address of the object that it’s constructed from
  - You can move construct / assign from it
- Xvalue means eXpiring value, an object that you can move but that can still be referenced elsewhere
  - A non-static data member of an rvalue is an xvalue
- An unnamed rvalue reference is an xvalue
  - 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 a prvalue or deduce it from another object
  - You can move construct / assign from it
- Most of the time if it doesn’t have a name it’s a prvalue, except for (not exhaustive):
  - Unnamed rvalue references that are xvalue
  - Non-static member functions that are prvalues
- 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 represent named objects, they exist “physically” in the memory and must kept their “value” until they are destroyed
- Xvalues represent objects that exists “physically” in the memory but that won’t be used anymore so their “value” can be moved to another object
- Prvalues represent objects that don’t exists “physically” in the memory, they don’t have an address, they are accessible only in the expression that created them, so their “value” can safely be moved somewhere else
  - If a prvalue is bound to a reference, it is materialized (it now exists “physically”) and its lifetime is extended to the lifetime of the reference it was first bound

Guaranteed copy elision (C++17)

- Copy elision is linked to prvalues
- Prvalues don’t represent “physical” object, but object that are “waiting” for being constructed
- When the object represented by a prvalue is needed for real work, the prvalue is materialized
  - That mean a temporary object is constructed with the prvalue, resulting in a xvalue
- All of this means that passing prvalues is free because there is no object to move, you just pass informations for constructing an object
  - But in fact the physical object is still constructed when the prvalue is constructed, it’s just not constructed where the prvalue was first build

Guaranteed copy elision (C++17)

```
VerboseClass iBuildStuff()
{
    return VerboseClass{};
}

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

--std=c++14

```
VerboseClass default constructor.
```

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

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

https://godbolt.org/z/jwWFoj

Guaranteed copy elision (C++17)

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

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

--std=c++14

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

-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)
```

https://godbolt.org/z/jwWFoj

When to move

- Lvalues:
  - Move them whenever you can, it will prevent some copy to be made, making the software faster
  - Moving const object 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

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);
}
```