References

09 June 2021 07:07

1. References are usually implemented internally by the compiler using pointer

Pass non-pointer, non-fundamental data type variables (such as structs) by (const) reference, unless you know that passing it by value is faster.
Almost never return an r-value reference, for the same reason you should almost never return an l-value reference.

2. References rules and issues

Rules:

- We cannot get the address of a reference itself because a reference is not an object (a region of storage) according to the C++ standard.
- Declaring a reference to a reference, an array of references, and a pointer to a reference is forbidden in C++

Issues:-std::ref, std::reference_wrapper will solve this https://www.nextptr.com/tutorial/ta1441164581/stdref-and-stdreference_wrapper-common-use-cases

- Template function will NOT AUTOMATICALLY deduce type as T&. Either you should specify in caller or in parameterType of template function
- Reference cannot rebind to another object. Therefore, assigning a reference to another does not assign the reference itself; it assigns the object:

```
int x=10, y=20;
int &xref = x, &yref = y;
xref = yref;
//xref still refers x, which is 20 now.
```

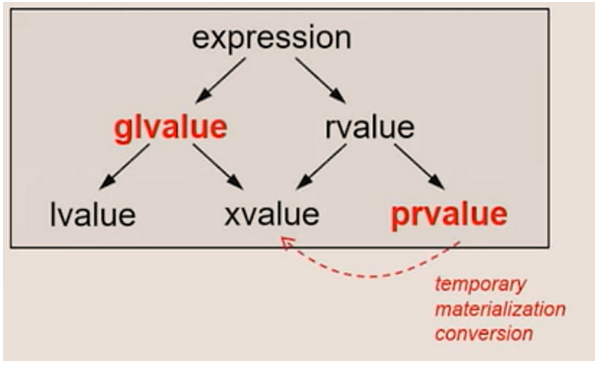
- As references cannot rebind, having a reference class member is a pain in the neck because a reference member makes **ENTIRE** class non-assignable
  - the default copy-assignment operator is deleted.
  - Move-semantics does not make sense with a reference member altogether.

- Reference types do not meet the Erasable requirement of STL container elements. Therefore, we cannot have a container (e.g., vector or list) of reference elements:

```
std::vector<int&> v; //Error
```

https://www.nextptr.com/tutorial/ta1441164581/stdref-and-stdreference_wrapper-common-use-cases

3. Lvalues and Rvalues



Each expression has some non-reference type, and each expression belongs to exactly one of the three primary value categories: prvalue, xvalue, and lvalue.

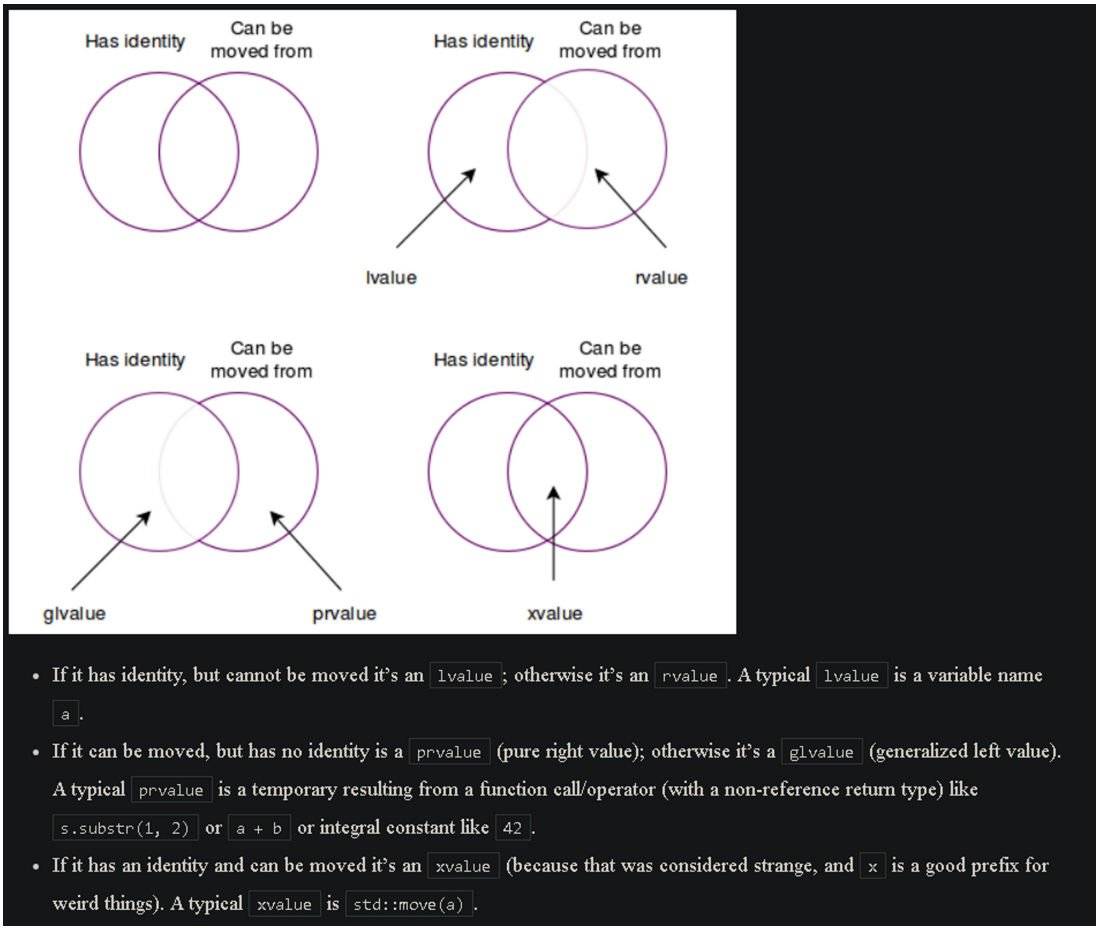
PRvalue = has no identity (Anonymous object with no address in memory) and can be moved.

X-value = has identity + can be moved. (expiring value).. Eg

- `a ? b : c`, the ternary conditional expression for certain b and c (see definition for detail);
- a cast expression to rvalue reference to object type, such as `static_cast<char&&>(x)`;

R-Value = xvalue + prvalue
L-Value = has identity & cannot be moved. Named-objects (has identity) are l-values. Has address in memory (const/non-const)
GLvalue = L-Value + X-Value. Expression whose evaluation determines identity of obj or function

With the advent of C++ 11, additional value categories have been identified and organized in a systematic way based on the observation that **there are two important properties of an expresion: has identity (i.e. 'can we get its address') and can be moved from.**



- If it has identity, but cannot be moved it's an `lvalue`; otherwise it's an `rvalue`. A typical `lvalue` is a variable name `a`.
- If it can be moved, but has no identity is a `prvalue` (pure right value); otherwise it's a `glvalue` (generalized left value). A typical `prvalue` is a temporary resulting from a function call/operator (with a non-reference return type) like `s.substr(1, 2)` or `a + b` or integral constant like `42`.
- If it has an identity and can be moved it's an `xvalue` (because that was considered strange, and `x` is a good prefix for weird things). A typical `xvalue` is `std::move(a)`.

4. C++ provides non-const references, const-references, r-value references

Lvalue reference:

- Non-const ref : References to non-const values can only be initialized with non-const l-values. They can not be initialized with const l-values or r-values.
- Const ref: can be initialized with const or non-const l-values or r-values .. Cannot modify value of referred var

Rvalue ref: References to r-values extend the lifetime of the referenced value. Initialized only using r-values. Int &&rref{5}

- Non-const
  - Can modify r-value since rvalue ref points to temporary variable
- const

5. Using function parameters as references for non-fundamental types instead of copy will lead to better performance

Downside of using non-const references as function parameters is that the argument must be a non-const l-value

```
int func(int& a)
int func(const int&a)
int func(int (&arr)[4]) ---> reference of array
int func(int& arr[4]) -----> array of references

int main()
{
  int a; int arr[4];
  func(a); //func(int&a) will be called
  func(2); //func(const int&a) will be called
  func(arr);
}
```

6. Conversion from l-value to r-value - static_cast using std::move

Can covert from a lvalue to a rvalue (to an xvalue more precisely) by using static_cast<X &&> without creating temporaries.

7. Function overloads

```cpp
struct X {};

// overloads
void fn(X &) { std::cout<< "X &\n"; }
void fn(const X &) { std::cout<< "const X &\n"; }
void fn(X &&) { std::cout<< "X &&\n"; }

int main()
{
  X a;
  fn(a);
  // lvalue selects fn(X &)
  // fallbacks on fn(const X &)

  const X b;
  fn(b);
  // const lvalue requires fn(const X &)

  fn(X());
  // rvalue selects fn(X &&)
  // and then on fn(const X &)
}
```

8. Function with r-value ref arg

Here you pass an r-value (accepts only rvalue), get a r-value reference which is an l-value
Use std::move on above to convert into x-value

But when used inside the function body, a parameter, whether `lvalue reference` or `rvalue reference`, is an `lvalue` itself: it has a name like any other variable.

```cpp
// parameter is rvalue reference
void fn(X && x)
{
  // but here expression x has an lvalue value category
  // can use std::move to convert it to an xvalue
}
```

9. Forwarding references

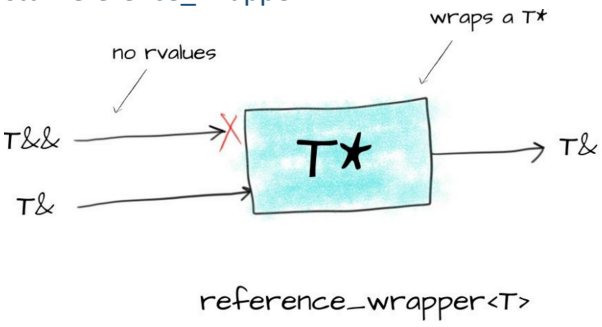
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4164.pdf
T&& is forwarding reference

```cpp
template<typename T>
void fn(T &&) { std::cout<< "template\n"; }

int main()
{
  X a;
  fn(a);
  // argument expression is lvalue of type X
  // resolves to T being X &
  // X & && collapses to X &

  fn(X());
  // argument expression is rvalue of type X
  // resolves to T being X
  // X && stays X &&
}
```

10. std::reference_wrapper



std::reference_wrapper is a copyable and assignable object that emulates a reference.
It works by encapsulating a pointer (T*) and by implicitly converting to a reference (T&).

```
std::reference_wrapper<int> nr;           //Error! must initialize.
auto r2 = std::ref(std::string("Hello")); //Error! no temporary (rvalue) allowed.

std::string str1{"Hello"};
std::string str2{"World"};
auto r1 = std::ref(str1);                 //OK - reference wrapper
auto r2 = std::ref(str2);                 //OK
//Assignment rebinds the reference_wrapper
r2 = r1;                                  //r2 also refers to str1 now
//Implicit conversion to std::string&
std::string cstr = r2;                    //cstr is "Hello"
//Possible to create an array of reference_wrapper
std::reference_wrapper<std::string> arr[] = {str1, str2};
```

To access and assign the members of an object (T), we have to use the 'std::reference_wrapper<T>::get' method

11. Passing std::reference_wrapper to template function

Template func will deduce the type after specifying in template function that it's a reference.. Manual specifying at caller doesn't work unlike normal references.

On contrary normal reference works either ways i.e. specify manually at caller (and not specifying in func def) (or) specifying only at func def.