Move semantics

Part 2: _TODO_

Index

- Forwarding references
  - How to declare them, why to use them
- std::forward
- Named return value optimization (NRVO)
  - When it doesn’t work but fallback to move
  - When you have to call std::move manually
- Pass by value vs by reference
- Lifetime extension of rvalues ?
- Other ?

Named return value optimization

- A local object return from a function may, in some case, be converted into an rvalue (xvalue or prvalue)
  - It's up to the compiler to chose (depending on other limitations)
  - If it's converted to a prvalue, then a copy elision can happen (NRVO)
- For this to happen, the return expression needs:
  - To be an id-expression (directly referring a name)
  - The return value of the function must be constructible from an xvalue of the type of the object returned

NRVO standard limitation

- Only id-expression are converted to rvalue, this implies some limitations:
  - If you std::move a variable then the compiler won’t be able to cast it to a prvalue, but it will still be moved
  - If you return an expression that result in a reference to a local variable, then a copy be made
- The later means that sometime it can be better to write more lines to have the return statement separated from some computation
  - “return (c ? a : b);”
  - “return a += b”;

NRVO technical limitation

- If the object returned is a parameter of the function, then NRVO can’t happen and the object will only be moved
  - This is because a function doesn’t control the lifetime of its parameters, it’s the caller that is responsible of creating and deleting the object
  - That means if the object was NRVO’d, it will be immediately destructed after the end of the function call, making the returned object unusable from the caller
- Sometimes the compiler will not be able to do NRVO because it will not be able to deduce which variable will be returned, but the conversion to an rvalue (xvalue) will still happen

Pass by value vs by reference

- For small and simple type it’s better to pass by copy, because it’s cheap and there will be no aliasing possible
- Otherwise if you need a copy in your function for computation purpose only, pass by copy
- If you need a copy because it will be assigned to something else, then a pass by const ref is usually better to preserve some cache (like std::string / std::vector)
- And lastly, if unsure pass by const ref