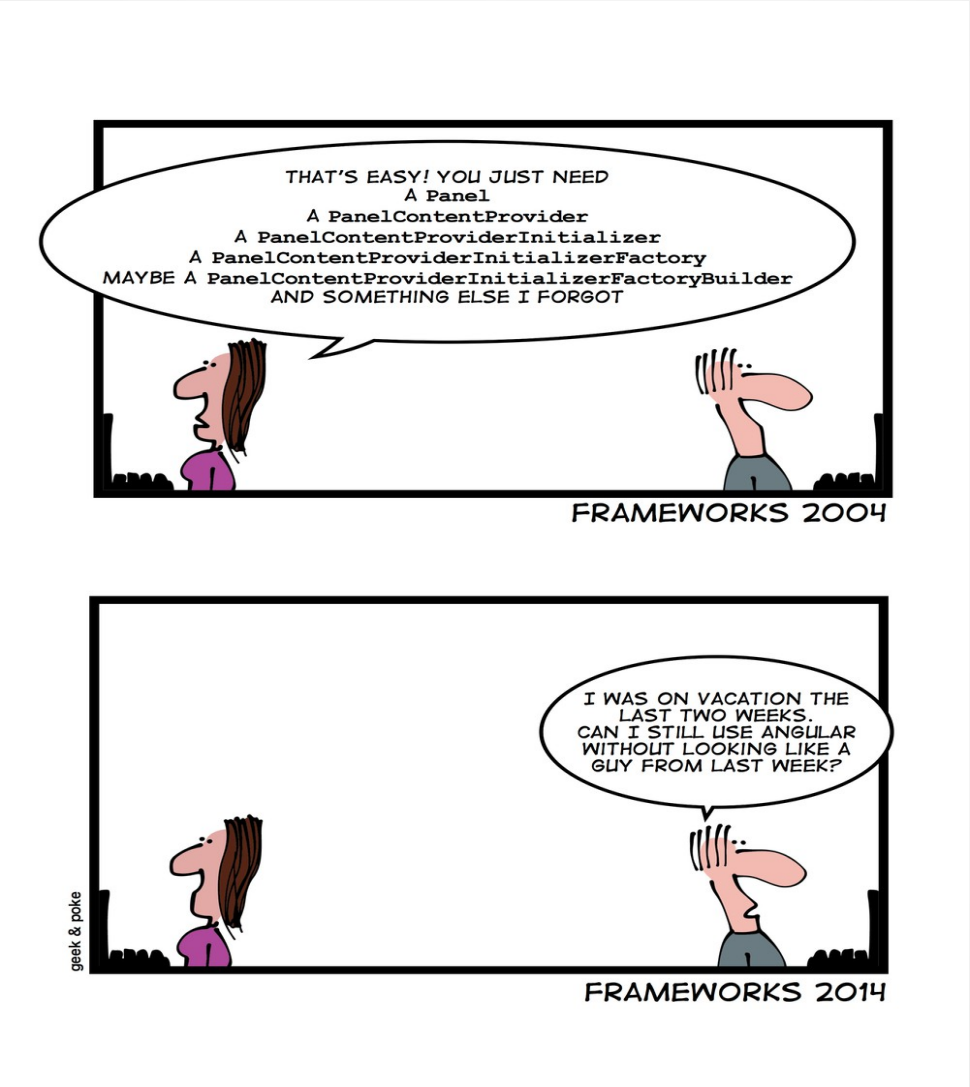
rvalue, move semantics & perfect forwarding



Krishna Kumar

Copy constructor & assignment operator

- A copy constructor is a special constructor for a class/struct that is used to make a copy of an existing instance.

  ```
  MyClass( const MyClass& other );
  MyClass( MyClass& other );
  MyClass( volatile const MyClass& other );
  MyClass( volatile MyClass& other );
  ```

  - // Not copy constructors
    ```
    MyClass( MyClass* other );
    MyClass( const MyClass* other );
    ```

- MyClass a;
  MyClass b(a); // or b{a}; // Call copy constructor
  MyClass b = a; // Call assignment operator

When do I need to write a copy constructor?

- When a class is a resource handle, that is, it is responsible for an object accessed through a pointer, the default memberwise copy is typically a disaster.
- If a class defines one of the following it should probably explicitly define all three:
  - ***Destructor*** – Call the destructors of all the object's class-type members
  - ***Copy constructor*** – Construct all the object's members from the corresponding members of the copy constructor's argument, calling the copy constructors of the object's class-type members, and doing a plain assignment of all non-class type (e.g., int or pointer) data members
  - ***Copy assignment operator*** – Assign all the object's members from the corresponding members of the assignment operator's argument, calling the copy assignment operators of the object's class-type members, and doing a plain assignment of all non-class type (e.g. int or pointer) data members.
- The Rule of Three claims that if one of these had to be defined by the programmer, it means that the compiler-generated version does not fit the needs of the class in one case and it will probably not fit in the other cases either.

Moving containers

- We can control copying by defining a copy constructor and a copy assignment, but copying can be costly for large containers. Consider:
- Vector operator+(const Vector& a, const Vector& b) {
  - if (a.size()!=b.size()) throw Vector_size_mismatch{};
  - Vector res(a.size());
  - for (int i=0; i!=a.size(); ++i) res[i]=a[i]+b[i];
  - return res;}
- void f(const Vector& x, const Vector& y, const Vector& z) { Vector r = x+y+z;}
- That would be copying a Vector at least twice (one for each use of the + operator). If a Vector is large, say, 10,000 doubles, that could be embarrassing.
- 'res' in operator+() is never used again after the copy. We just wanted to get the result out of a function: we wanted to move a Vector rather than to copy it.

Lvalue and rvalue

- An lvalue is an expression that refers to a memory location and allows us to take the address of that memory location via the & operator.
  - int i = 42; // i is an lvalue
  - int* p = &i; // p is an lvalue
  - int& foo(); // foo() is an lvalue
  - arr[0]; // arr[0] is an lvalue

- An rvalue is an expression that is not an lvalue.
  - int y = x+z; // x+z is an rvalue – valid only till the semi-colon
  - int j = 42; // 42 is an rvalue
  - j = foobar(); // foobar() is an rvalue

Lvalue and rvalue references

- int x, foo();
- int& lrx = x; // Reference to an l-value
- int& lrx = foo(); // fail – can't take reference to an lvalue
- const int& clrx = x; // Constant reference – to an lvalue
- cont int& crrx = foo(); // Constant reference – to an rvalue
  // crrx cannot be modified.
- int&& rrx = foo(); // Reference to an rvalue – modified
- const int&& crrx = foo(); // Constant rvalue reference –
  // not common

Why rvalue references

- std::string s1{“Wall-E”};
- std::string s2{s1}; // copy-construction is invoked
- We hope s2 is a independent copy of s1, a new memory has to be allocated, member-wise copy of elements, and memory deleted when no longer used. This is a lot of work, but we do have an independent copy s2 and we require this.
- However. Image we have a function:
- std::string newmovie() {return “Inside out”;}
- std::string s3{newmovie()};
- Compiler should do named-value optimisation, which is less likely but the compiler calls the copy-constructor to be safe in c++98
- Wouldn't it be nice, if we just used the memory allocated by newmovie()? In C++11, that's what happens, even if the return-value optimisation fails, it calls the move constructor. Standard ensures this happens!

Move semantics

- Move semantics makes it possible for compilers to replace expensive copying operations with less expensive moves. In the same way that copy constructors and copy assignment operators give you control over what it means to copy objects, move constructors and move assignment operators offer control over the semantics of moving.
- Move semantics also enables the creation of move-only types, such as std::unique_ptr, std::future, and std::thread.
- std::move is merely a function templates that performs an unconditionally casts its argument to an rvalue. In and of itself, it doesn't move anything.
- Doesn't do anything at runtime.
- template<typename T> //C++14 std::move
- decltype(auto) move(T&& param) {
  - using ReturnType = remove_reference_t<T>&&;
  - return static_cast<ReturnType>(param);
- }

std::move

- // Classic swap function

```
template<class T>
void swap(T& a, T& b) {
  T tmp(a);
  a = b;
  b = tmp;
}

X a, b;
swap(a, b);
```

- // Swap with move

```
template<class T>
void swap(T& a, T& b) {
  T tmp(std::move(a));
  a = std::move(b);
  b = std::move(tmp);
}

X a, b;
swap(a, b);
```

Perfect Forwarding: The Problem

- // simple factory function:

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg arg) {
    return shared_ptr<T>(new T(arg));
}
```

- Obviously, the intent here is to forward the argument arg from the factory function to T's constructor. Ideally, as far as arg is concerned, everything should behave just as if the factory function weren't there and the constructor were called directly in the client code: perfect forwarding. The code above fails miserably at that: it introduces an extra call by value, which is particularly bad if the constructor takes its argument by reference.

Perfect forwarding: Quick fix

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg& arg) {
    return shared_ptr<T>(new T(arg));
}
```

- That's better, but not perfect. The problem is that now, the factory function cannot be called on rvalues:
- factory<X>(hoo()); // error if hoo returns by value
- factory<X>(41); // error
- This can be fixed by providing an overload which takes its argument by const reference:

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg const & arg) {
    return shared_ptr<T>(new T(arg));
}
```

Perfect forwarding – Quick fix (issues)

- There are two problems with this approach. Firstly, if factory had not one, but several arguments, you would have to provide overloads for all combinations of non-const and const reference for the various arguments. Thus, the solution scales extremely poorly to functions with several arguments.

- Secondly, this kind of forwarding is less than perfect because it blocks out move semantics: the argument of the constructor of T in the body of factory is an lvalue. Therefore, move semantics can never happen even if it would without the wrapping function.

- It turns out that rvalue references can be used to solve both these problems

Universal reference

- void f(Widget&& param); // rvalue reference
- Widget&& var1 = Widget(); // rvalue reference
- auto&& var2 = var1; // not rvalue reference
- template<typename T>

  void f(std::vector<T>&& param); // rvalue reference
- template<typename T>

  void f(T&& param); // not rvalue reference
- “T&&” is either rvalue reference or lvalue reference. Such references look like rvalue references in the source code (i.e., “T&&”), but they can behave as if they were lvalue references (i.e., “T&”).
- Their dual nature permits them to bind to rvalues (like rvalue references) as well as lvalues (like lvalue references).

Universal reference – type deduction

- *Type deduction* determines if an universal reference is a rvalue reference or a lvalue reference.

```
template<typename T>
void f(T&& param); // param is a universal reference
Widget w;
f(w); // lvalue passed to f; param's type is
        // Widget& (i.e., an lvalue reference)
f(std::move(w)); // rvalue passed to f; param's type is
                    // Widget&& (i.e., an rvalue reference)
```

Pefect forwarding: Solution

- Given these rules, we can now use rvalue references to solve the perfect forwarding problem

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg&& arg) {
  return shared_ptr<T>(new T(std::forward<Arg>(arg)));
}
```

- std::forward casts its argument to an rvalue only if that argument is bound to an rvalue.
- Use std::forward on universal references
- Use std::forward do noting at runtime.

Reference collapsing rules

- A& & becomes A&
- A& && becomes A&
- A&& & becomes A&
- A&& && becomes A&&
- std::forward is defined as follows:

```
template<class S>
S&& forward(typename remove_reference<S>::type& a)
noexcept {
    return static_cast<S&&>(a);
}
```

Perfect forwarding: Solution (lvalues)

- X x;
- factory<A>(x);
- Then, by the special template deduction rule stated above, factory's template argument Arg resolves to X&.
- shared_ptr<A> factory(X& && arg) {
- return shared_ptr<A>(new A(std::forward<X&>(arg)));
- }
- X& && forward(remove_reference<X&>::type& a) noexcept {
- return static_cast<X& &&>(a); }
- Applying the reference collapsing rules, this becomes:
- **shared_ptr<A> factory(X& arg) {**
  - **return shared_ptr<A>(new A(std::forward<X&>(arg))); }**
- **X& std::forward(X& a) { return static_cast<X&>(a); }**

Perfect forwarding: Solution (rvalues)

- factory<A> is called on an rvalue of type X:
- X foo();
- factory<A>(foo());
- The compiler will now create the following function template instantiations:
- shared_ptr<A> factory(X&& arg) {
- return shared_ptr<A>(new A(std::forward<X>(arg)));
- }
- X&& forward(X& a) noexcept {
- return static_cast<X&&>(a);
- }

Move only types

- unique_ptr<int> up{new int{1}}; // Create unique ptr
- auto up = make_unique<int> (1); //C++14
- unique_ptr<int> makeMagicNumber() {

  return unique_ptr<int>(new int(12)); //return moves!

  } // Okay
- unique_ptr<int> up2{up1} // ERROR
- unique_ptr<int> foo = std::move(up); // Okay

References

- Bjarne Stroustrup's “C++ Programming Language 4ed”
- Scott Meyer's “Effective Modern C++”
- https://www.youtube.com/watch?v=cO1lb2MiDr8
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html
- http://thbecker.net/articles/rvalue_references/section_01.html