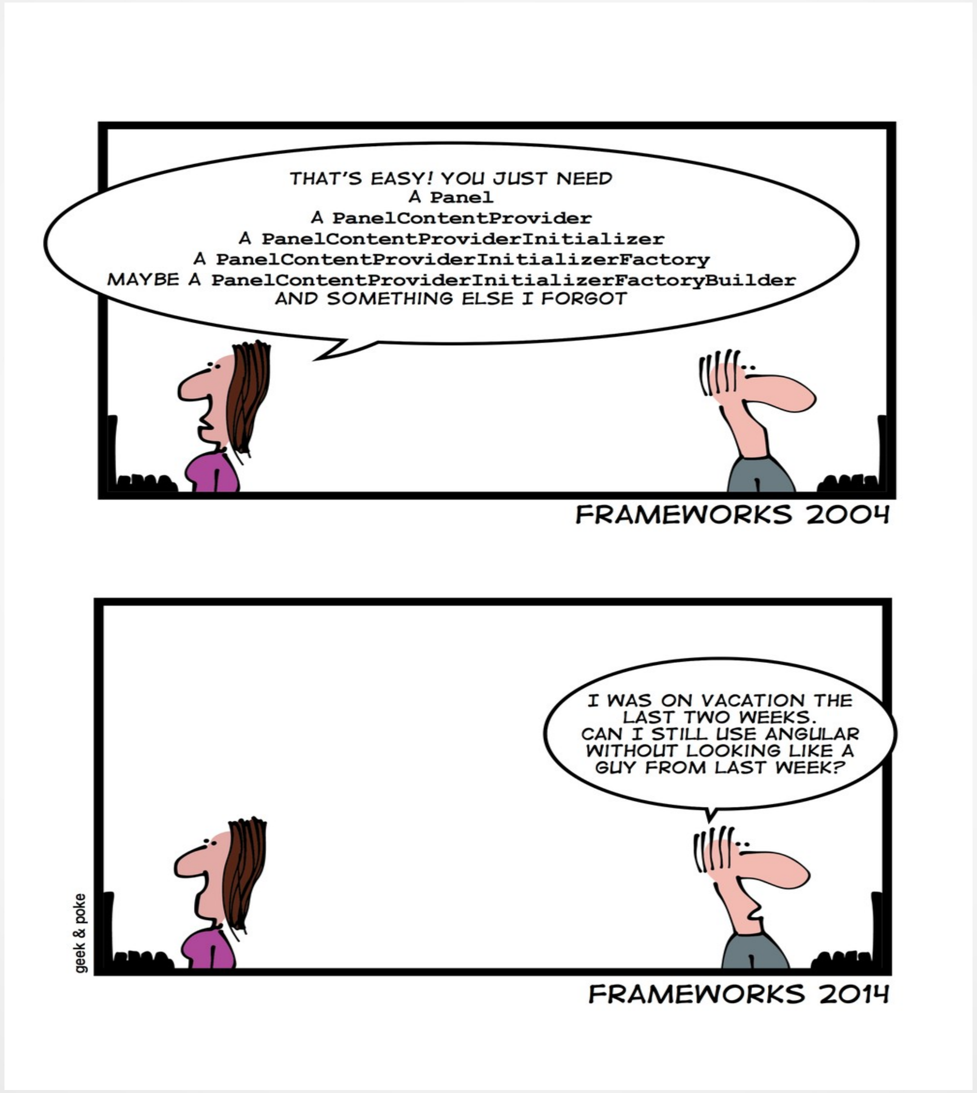
Universal references and Move semantics


THAT'S EASY! YOU JUST NEED
A Panel
A PanelContentProvider
A PanelContentProviderInitializer
A PanelContentProviderInitializerFactory
MAYBE A PanelContentProviderInitializerFactoryBuilder
AND SOMETHING ELSE I FORGOT
FRAMEWORKS 2004
I WAS ON VACATION THE LAST TWO WEEKS. CAN I STILL USE ANGULAR WITHOUT LOOKING LIKE A GUY FROM LAST WEEK?
geek & poke
FRAMEWORKS 2014

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 Move
- decltype(auto) move(T&& param) {
  - using ReturnType = remove_reference_t<T>&&;
  - return static_cast<ReturnType>(param);
- }

References