Object Oriented Programming cont.

Temporary objects, Big four, Operators overloading

Object construction and destruction cont.

Living on my own ...

Constructors & Destructors - Order explanation

- The order of construction/destruction was demonstrated for **automatic-duration objects** relative to one another
- Order of construction/destruction for **composed classes** - Live demo
- As it was demonstrated, the simplest scenario of automatic-duration objects ensures **nested lifetimes** of objects
- The purpose of having nested lifetimes is to avoid **broken dependencies**
    - **Broken dependency: object A trying to use object B after B's destruction**
    - **In the simplest example objects cup1, cup2 did not keep pointer/reference to each other**
    - **However in the real world it is often the case that an object keeps pointer/reference to another object**

Constructors & Destructors - Temporary Objects

- In C++ there are many cases where a **temporary object** is created. A **temporary object** either *explicitly* (by the developer), or *implicitly* (by the compiler)
    - *Explicitly* using the form *Type()* (possibly with arguments to the constructor)
    - Returning by value from a function
    - Some casts
    - Intermediate values during expression evaluation
- When created, a temporary object has no name (or at least, initially...)
    - *“True temporary objects in C++ are invisible - they don't appear in your source code”* - Scott Meyers, "More Effective C++"

Yoda - 21 years old C++ developer

C++ Big Four

The rule of thumb

Dealing with dynamic memory

- When a class holds a specific *resource (e.g. pointer)*, dealing with dynamic memory is inevitable
- Basically that dynamic memory is controlled by the developer of a class
- Object construction is the point in time when an object acquires a resource (*a pointer*) - in particular dynamic memory allocation
  - Default constructor - *default?*
  - Parameter constructor - *may not be connected with dynamic memory allocation*
  - Copy constructor - *default?*
- Object destruction is the point in time when an object **should** release the acquired resource - in particular dynamic memory deallocation
- *RAII Idiom https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization*

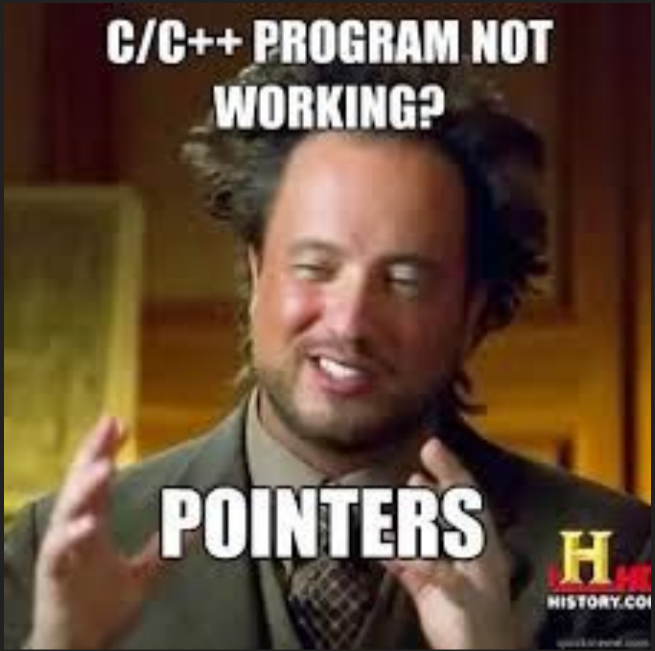
Object assignment

- Object assignment is like *primitive type variables* assignment
- After a variable is **defined**, it can **assign** the value of another **defined** variable
- Object assignment work on the same principle, but there is a special member function (*method*), to control the assignment
  - Copy assignment operator is what allows you to use = to assign one *instance* to another
  - Copy assignment operator - Live demo
  - Again, as previously met special methods, operator= is generated by default if missing
- But are we satisfied with the default generation of *copy assignment operator* in case of dynamic memory?
- *Shallow copy* vs. *Deep copy*

So, why Big Four?

1. Default constructor
2. Copy constructor
3. Copy assignment operator
4. Destructor

- The rule stands that "*If you need to **explicitly define** one of the special member functions above, then you need to **explicitly define** all of them*"
- Why? - *try to break that rule.*



Operators overloading

Operator overloading - Basics

- Most of the built-in operators available in C++ can be redefined or overloaded
- Overloaded operators are member functions with special names: the keyword "*operator*", followed by the symbol for the operator being defined.
- Like any other function, an overloaded operator has a return type and a parameter list
- *Complex number Live Demo*
- Most operators can be overloaded as non-member functions, but then the function should take one more parameter, because the current *live object (the object that "this" points to)* is not available.

Operators which can be overloaded

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Operators which cannot be overloaded

::	.*	.	?: