Factory C

<https://directory.fsf.org/wiki/Factory_C> - or - [https://factoryc.org](https://factoryc.org/)

@author Christopher "willy" Posten:

Summary:

Factory C is... What I am going to do is drive the point as hard as I can for everyone who may be a critic of a new form of Object-Oriented Programming being introduced here, a point for what Factory C implementations are capable of supporting when you implement a "struct class" properly using all the principles, patterns, and practices of OOP in C (my own personal definition of OOC) (Factory C) and its "keyword" definitions (methods and macro definitions). What Factory C can deliver in terms of STRATEGIC programming is so incredibly developed and superior to anything I was ever procedurally capable of in C++ (Im not the first one to implement an OOC after dis-enchantment in C++) that a "struct class" when properly implemented, is a fully self-contained unit of organized information that not only is multiple inheritance possible, but partial inheritance (Bridge Pattern) and changes of inheritance at runtime is possible (easy as using a constructor).

Sending the message method to the object pointer to by the pointer obj (as in calling a function or invoking a method) would require the following code in C++:

obj->method(argument);

In Objective-C, this is written as follows:

[obj method:argument];

In Factory C:

((void(\*)(Object\*,...))\*struct(class)

(obj)("method"))(argument);

And to override at runtime (C++ i think was disabled):

\*struct(class)(obj)("method") = & func ;

Or if Factory C was used to implement a compiler (like C is for Constructor):

obj("method")(argument);

In Factory C the method or data member name can become a variable when accessed via Factory Method (map). This is, for one, a place to include strategy enough to put a variable there, It is, (also) a potential overload case or bridge between classes when obj points to a different instance from another class, therefore dynamic also or still polymorphic without a variable, and using a virtual heap to access the datafield position of the virtual table interface. This and the object datamembers become overloaded also. There is therefore no limitations to inheritance or coupling, anything (data member or interface method primarily) can be at any data field position or at any level of extension of any object or its interface and be used as an inherited data member or interface method in any other class and its object that declares the same (STRATEGICALLY THE SAME)(as in same name + datatype) datamember in a nearby class. In Factory C there are definitions for implementing a class inside of a set of functions using a series of constructors before being able to use the vto (virtual table object) constructor for an object of a bytesize amount determined by the class(ComplexHeap) registration declared inside the class interface heap for that class (and dynamically by typeid). All that is offered elsewhere I know is in C++ where it offers an extra parameter with the new operator (operator new) one that allows for an undefined amount of bytes to be allocated as extra bytes for an object of a type specified by class name (as dynamic as compile-time so non-dynamic to a programmer) not factory method. But the easiest way to see the difference or any further OO Development in that area is that classes get registered by keyword and their objects are instantiated also (so not only by the new operator) by factory method that returns the constructor/initializer that returns the object, as compared to the constructor only being returned from a class name using new.

Actually I might argue a bit myself over the cast being removed from the procedure used to access an interface method/data member (if a compiler was implemented using Factory C) since it looks like what gets determined at runtime would be at a different point from what is determined at compile-time. But you never know what magic a new compiler can produce until you get there. C++ conjoins both object and interface using its member access operator, which has no place in level 2 and ecspecially level 3 OOC using Factory C (the member access operator(s) . or ->), being replaced by a factory method. A Factory Method, is essentially a map (with or without being for a virtual table object in my definitions). A map called without seeing the data structure it uses passed into it, by default a singleton method for a class (implemented struct class)(the class factory method).

So to put things in perspective, when a function is called on a class object it may "build" the class of the object by adding a datamember or interface method using a constructor, if that data member or interface method is already added, then its a constructor that acts like an extra prototype (ecspecially since you CAN put prototypes inside of a function) but that is what i would define as a "builder function". also there is a "complex function" that uses the "class(ComplexHeap)" of the class object the function is called on, so it does something like add the datamember bytesize amounts together to compute the object size or initialize every datamember (you don't need to implement a constructor/destructor for a level 2-3 object). Also to consider inheritance (one of the last things to get considered by many) is that, like I said Inheritance is dynamic-at-runtime and easy like using a constructor inside a function (Factory C does what every programming language extended from C "does" the only difference is that Factory C: IMPLEMENTS COMPLEX OBJECT LEVEL PROGRAMMING) (THIS MEANS THAT A CLASS CAN BE AN OPERATING SYSTEM OR A SECURITY COMPLEX). Dynamic and non-dynamic classes are polymorphic together and so they are accessed using the same multimap or iterated using the same iterator. (DONE).

Factory C is Strategically Inclined to use a twin-Virtual Table configuration (after C++ was known to use a single virtual table). which allows destroy() to be called, deallocating every object inside the primary virtual table (also calling each destructor), leaving objects to accumulate inside the secondary virtual table only. The primary Virtual Table is fully under the programmers control, no access to it unless the programmer themselves instantiates a basic vto (virtual table object) using a keyword like new. The secondary virtual table accumulates certain important variables or "objects" including the jump buffer used by each nested try/throw/catch and the stacks of returns from the printf family or the scanf family (of functions) when using cin cend/cout cend.

Factory C offers strategic control over what catch block an exception jumps (or is thrown) to by popping a controlled amount of jump buffers off of the jump stack. Also extended from the Exception is an Assertion object/class that uses something similar to the old assert macro inside of its constructor, along with an Override exception that is for a program to have tables of registered overrides (as runtime overrides) (something C++ was disabled with again like not being able to work with the constructors address and having a non-dynamic class).

Features:

Datastructures - (this includes templates available by typename)

-Hash Table (Pointer)

-Hash Table (CString)

-Stack (Single Linked-List)

-Vector

-Binary Search Tree

-Binary Search (for array)

Design Patterns - (after the primary patterns: Template, Polymorphism, Virtual)

-Factory (Factory Method)

-Factory (Factory Function)

-Control Factory (Constructor Sequence) (plus Command)

-Adapter (plus Decorator and Observer)

-Strategy vs. Overload

Keywords - (every C++ keyword given a similar definition or not)

- this includes but not limited to (everything but operator):

new delete typeid class

override virtual friend using

template nullptr typename try

throw catch this explicit

...

- this includes C++ standard library "keywords" or user defined keywords:

multimap map complex function

is\_polymorphic type\_info accumulate

is\_class is\_object nothrow iterator

ref cin cout endl

set equal greater cend

cbegin ...

- extended use of C keywords (includes but not limited to):

register default case struct

union ...

- Factory C (mostly original) Keyword Candidates:

factory adapter destroy build

demolish deaccumulate is\_adapter is\_builder

typemax define init int\_id

bytes instanceof allocate deallocate

reallocate drop primary jump

...

- Multi-Keywords:

class(builder)(multimap) class(builder)(function)

class(builder)(register) class(builder)(complex)

class(builder)(auto) struct(bunker)(typeid)

struct(bunker)(is\_object) struct(bunker)(instanceof)

stack(control) stack(arg)

object(cout) object(cin)

string(equal) string(greater)

...

- Overlaps:

struct(bunker)(new) defined as: accumulate

struct(bunker)(delete) defined as: deaccumulate

typename(Macro) defined as: define(Macro)

multimap(-1) + map() defined as: is\_class

instance defined as: is\_object

...

- To be anticipated (extended use of):

goto do break ...

- Overloads (to be used by any class object):

copy insert remove search

at size replace begin

end front back ...

- And of course, every C keyword (that I won't name).

Library - ...

LIBC or CLIB (C Standard) + GNU GCC Compiler required (but not limited to)

Copyright:

copyright 2023 General Public Licence v.3

References:

OOC (textbook) (in C)

Head First: Design Patterns (textbook) (in Java)

structclass (stackoverflow.com) (in C)

Conclusion:

Now... since i don't have to push a point anymore (as to why its worth considering), or we have a conclusion, my conclusion is that Factory C is still just C (like is should be) but there is no new compiler, no new OOPC (Object-Oriented Pre-Compiler) although that may just be the next thing on my list. Factory C uses 3 definitions or level of Object: Virtual, Factory, Complex. The points made before are for only if you follow all the principles, patterns and practices of this well developed form of OOC into using both Factory level Object definitions with Complex level Object definitions both with implemented struct classes and with "builder" classes that are given their implementation(s) inside of function(s) using a series of constructors/factory methods. The Virtual level Object definition may only use a portion of the OOC/OOC.h include but offer basic Object-Oriented Programming or no Object-Oriented Programming. That's when we cut-corners, or saw-off an implementation procedurally creating something made only for a very specific purpose like the "interface registration" meant for a "class interface heap" of a "factory table class" at the "factory table", using a very strategic method a factory method that is known to Object-Oriented Design Patterns as the vto factory or virtual table object factory, possibly in the form of a heap of heaps allthough the virtual table object factory in Factory C is an overload across the full range of registered factory table classes at the Factory Table.

Factory C boasts 3 primary hash tables and two secondary hash tables with 10 additional stacks for everything from a factory method, to a vto (virtual table object), to exceptions, to jump buffers for the exception throws (try), to interfaces, etc... Factory C uses several multi-methods that are themselves "flags" so you put true or false in a function parenthesis, that would determine how a set of function parenthesis(s) get returned after and what they return (if anything). These flag functions return things like factory methods and constructors for things, potentially returning something in the end. Or, like the Control Factory, put function returns onto a stack, a stack placed strategically in an array beside the function address to be called from the Control Factory, after its initializer is called, initializing the iterator known as the Controller, This Controller, with the Control Factory, is also used as the Controller for the Command (that we should all know and love). By default, you would use an array as a virtual table object from a typename (that its type would be from a typename statement) to initialize the Control Factory (Constructor Sequence). Factory C is my personal extravaganza of datatypes/constructors. Factory Constructor. Cheers.

<https://www.linkedin.com/in/christopher-posten-20a025265>