Typing Analysis

Dima Trushin

Contents

1 General information

The application uses Qt environment as an ecosystem. It does not use RTTI but uses exceptions. All components of the program must be inside NSApplication namespace. Nested namespaces should be placed in a subfolder of the project.

1.1 Addressable objects

I use term addressable object. It means an object that can be referenced by other objects. There are several types of addressable objects:

1. QObject
2. CApplicationImpl
3. Observer/Observable (TO DO)

Addressable objects must be created on heap via `std::make_unique` or similar mechanism. An exception to this rule: you may create an addressable object inside another addressable object.

1.2 Exceptions

The strategy is to catch exception, show a message, and die. Since Qt environment is not totally exception safe, it is not allowed to throw exceptions in an event loop. If a QObject requires to throw an exception and terminate the program it catches the exception by itself and then sends a signal to CQtLoopException object. CQtLoopException shows a message and stops the event loop. CQtLoopException is a singleton, it uses CAnyGlobalAccess template (see 5.1).

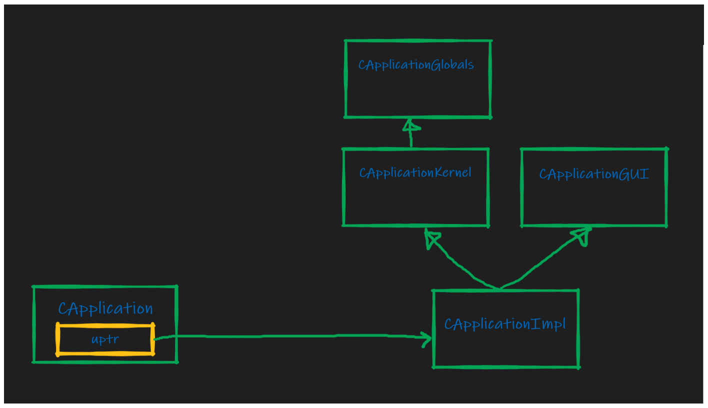
2 Application Structure

2.1 Overview

CApplication object initialize all required resources for the application including the ones to interact with Qt ecosystem. It may throw an exception while constructing. In order to minimize stack usage CApplication contains `std::unique_ptr` to CApplicationImpl (it is addressable object). CApplicationImpl consists of four parts:

1. **CApplicationGlobals**. Its purpose is to initialize global resources, e.g., timers, loggers, thread pools, etc. Application initializes all global resources (basically singletons) at the start. This allows not to wast time on the first call. Also, it is inconvenient to initialize timers via the first call.
2. **CApplicationKernel**. Its purpose is to initialize the kernel of the application. The kernel does not depend on the GUI and uses MVC via observer pattern to interact with GUI.
3. **CApplicationGUI**. Its purpose is to provide View wrappers over Qt resources compatible with MVC pattern.
4. **CApplicationImpl**. Its purpose is to connect the kernel and the GUI via MVC.

The order of construction is ensured by the inheritance mechanism.



2.2 Global Objects

Timer Application uses one main timer. This is a singleton with respect to the template here 5.1. It starts in the constructor of CApplicationGlobal. Timer return CTime object. It does not depend on particular units but you may convert it to any units you want. Internally `std::chrono` is used.

2.3 Kernel

2.4 Qt Resources

2.5 ApplicationImpl

3 Code

4 Implementation

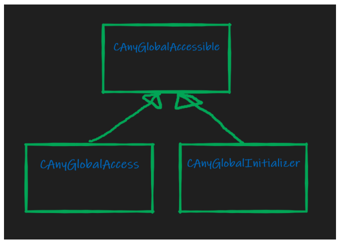
5 Library

5.1 Singleton

CAnyGlobalAccess template consists of three parts:

1. **CAnyGlobalAccessible**. It provides a static storage for a global object. You do not access it directly.
2. **CAnyGlobalAccess**. This object is used to access the global object. The global object must be initialized before access object is created.

3. **CAnyGlobalInitializer**. This object is used to initialize the global object. You need to create one instance of this object in order to initialize the global object.



Description The pattern is used to make a global object with non-trivial constructor without explicitly defining the object globally. **WARNING** this thing is NOT thread safe! In order to use the pattern we must provide:

1. `TAccessible` – the class for the global object
2. `TID` – identification class. If we want to have several global objects of a class Type, we must distinguish them by a dummy class `TID`. For example, `CAnyGlobalAccessible<Type, A>` and `CAnyGlobalAccessible<Type, B>` store different instances of objects of type Type in static storage. Since static objects defined by the class they belong to, classes `A` and `B` are required to distinguish the instances.

   Example of a dummy class declaration:

   ```
   class CGlobalAccessibleID;
   ```

3. A class for initialization of the global object. It must be inherited from `CAnyGlobalInitializer`. The class must inherit all the constructors of the base class.

   ```
   class CMyInitializer : CAnyGlobalInitializer<TAccessible, TID> {
      using CBase = CAnyGlobalInitializer<TAccessible, TID>;
    public:
      using CBase::CBase;
    };
   ```

4. A class for getting access to the global object must be publicly inherited from `CAnyGlobalAccess`. It has only a default constructor. It asserts if the global object has not yet been initialized.

   ```
   class CMyAccessor : public CAnyGlobalAccess<TAccessible, TID> {};
   ```

Example Suppose we want a global `int` variable for logging:

```
class CLoggerCounterID;
class CLoggerCounterInitializer :
  CAnyGlobalInitializer<int, CLoggerCounterID> {
  using CBase = CAnyGlobalInitializer<int, CLoggerCounterID>;
public:
  using CBase::CBase;
};
class CLoggerCounterAccess : public CAnyGlobalAccess<int, CLoggerCounterID> {};
```

Then in code we write something like that:

```
...
CLoggerCounterInitializer Init(0);
...
CLoggerCounterAccess LogCounter;
++(*LogCounter);
std::cout << *LogCounter << std::endl;
...
```