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).

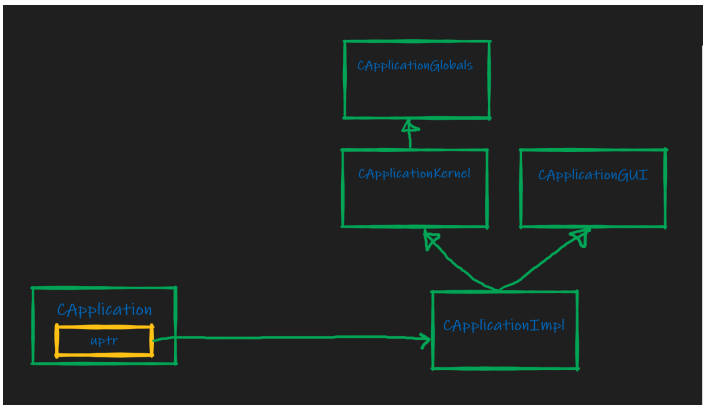
2 Application Structure

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:

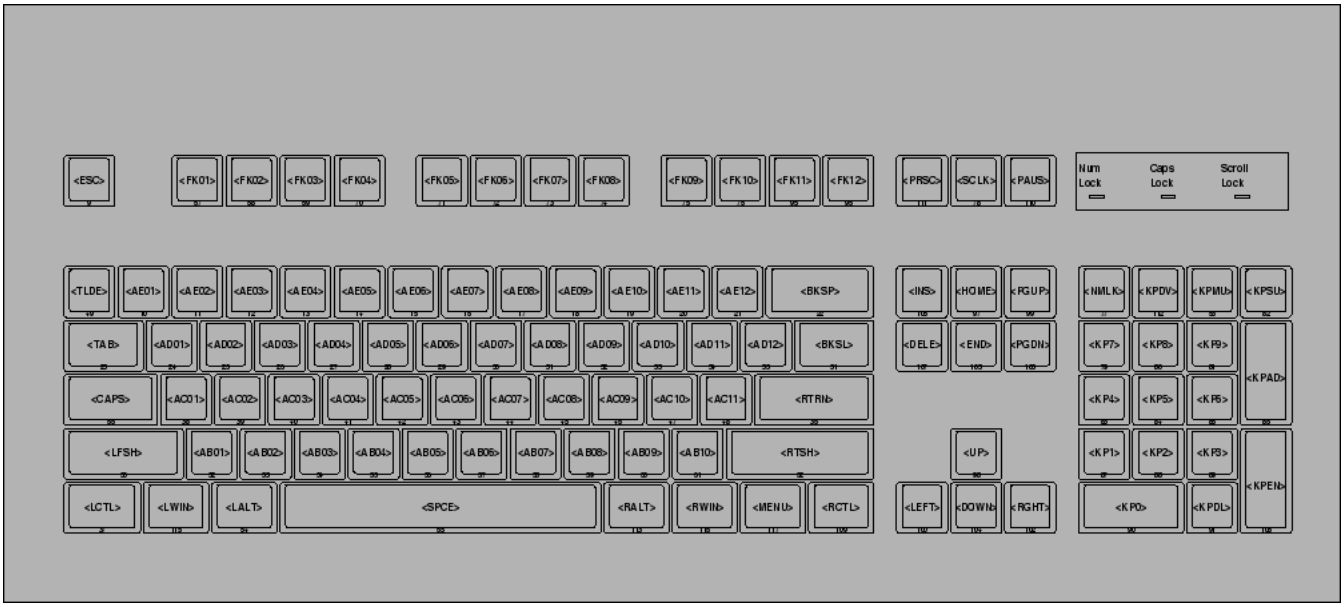
1. **CApplicationGlobals**. Its purpose is to initialize global resources, e.g., timers, loggers, thread pools, etc. Application initializes all global resources (basically singletons) at the start. This allows not to wast time on the first call. Also, it is inconvenient to initialize timers via the 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 Keyboard

Each key on a keyboard has its position identifier `CKeyPosition` and a key identifier `CKeyId` (TO DO not implemented). CKeyPosition points out a physical location on a keyboard (e.g. a row and a column where the key is located). CKeyId identify the key depending on the keyboard layout (qwerty, Dvorak, etc.) CKeyPosition is an enum with following identifiers (xkb identifiers):



The picture is taken from here

It should be noted that the numeric values of the identifiers are slightly different from the one in xkb.

2.3 Global Objects

2.3.1 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.

2.3.2 KeyboardHandler

Application uses CKeyboardHandler object to intercept keyboard system-wise, that is, it intercepts the keyboard even if application is not in the focus. Qt does not support such functionality. Thus the corresponding mechanism is implemented.

CKeyboardHandler is wrapper into a singleton according to Section 5.1. It is initialized in CApplicationGlobal. From the user point of view CKeyboardHandler able to send the following information to the application objects:

1. **CRawKeyEvent**. This is a system independent representation of a keyboard event (TO DO currently not implemented). This event is sent via Observer pattern (TO DO currently not implemented).

2. **CQtException**. This is a message with information about an exception encountered. CKeyboardHandler sends a Qt signal to CQtLoopExceptionHandler with the corresponding CQtException object. The application terminates on any exception.

Interception of the keyboard is implemented as follows. CKeyboardHandler spans a worker thread with OS dependent message loop. There are two objects operate on the worker thread:

1. **CKeyboardListener**. This object starts an OS dependent event loop with exec() function. It listen to two type of messages:

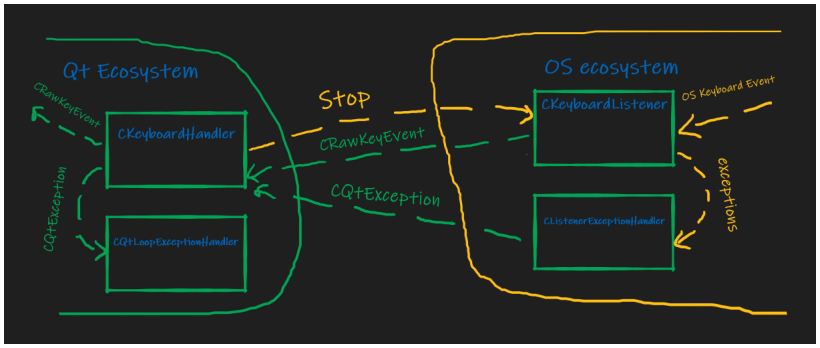
   (a) OS key events.

   (b) CKeyboardHandler stop event.

   CKeyboardListener intercepts any key events in OS, transform system dependent key events into CRawKeyEvents (TO DO currently not implemented), and sends the events to CKeyboardHandler. If stop event is encountered the event loop on the worker thread is terminated and the thread stops.

2. **CListenerExceptionHandler**. This object intercepts any exceptions on the working thread, transforms the excetions to CQtException-s, and sends them to CKeyboardHandler. If an exception is encountered the event loop is terminated and the worker thread stops.

CKeyboardListener is OS dependent. Currently support for Windows, MacOS, and Linux is provided (TO DO only Windows listener is implemented). In order to send the Stop signal in and OS independent fashion the application uses CAnyKeyboardKiller object. It is based on CAnyMoveable object as described in 5.2.1.



Green arrows represent queued Qt signals and yellow arrows represent OS dependent signals.

2.4 Kernel

2.5 Qt Resources

2.6 ApplicationImpl

3 Code

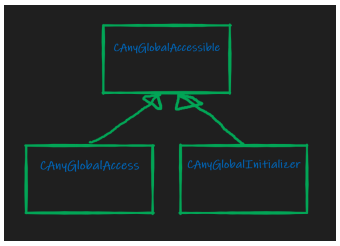
4 Implementation

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;
...
```

5.2 AnyObject

5.2.1 AnyMovable

Description `CAnyMovable` allows you to store any movable only class without any restrictions on the class. It also allows you to provide an interface to a class and implementations for different classes of the method.

The `operator->()` goes without any checks and may fail if the object is empty, that is, does not store anything. It is your responsibility to call `isDefined()` method before accessing the interface.

The class `CAnyMovable` does not use Small Object Optimization. In particular, move operations are always cheap. `CAnyMovable` has value semantics and extends the ideas of Sean Parent's talk on cppcon about Run-time Polymorphism.

How to use In order to use the template you need:

1. Create an interface class:

```
template<class TBase>
class IAny : public TBase {
public:
  virtual void print() const = 0;
};
```

   The class describes the interface of an abstract object you want to support. Do not use names with prefix underscore here, e.g.,

BAD: `virtual void _print() const = 0;`
GOOD: `virtual void print() const = 0;`

   This interface will be accessible when using `operator->()` on `CAnyObject`. The example is below.

2. Create an implementation class:

```
template<class TBase, class TObject>
class CAnyImpl : public TBase {
  using CBase = TBase;
public:
  using CBase::CBase;
  void print() const override {
    std::cout << "data = " << CBase::Object() << std::endl;
  }
};
```

   Here you implement all the functions from the interface. The parameter `TObject` is used to reimplement the behaviour for different types of objects if needed. In the example above, it may happen that `TObject` does not support stream `operator<<` and you want to define a specialization of the implementation class. Access to the stored object is provided by the method `CBase::Object()`.

3. Create your Any class:

```
class CAny : public CAnyMovable<IAny, CAnyImpl> {
  using CBase = CAnyMovable<IAny, CAnyImpl>;
public:
  using CBase::CBase;
  friend bool operator==(const CAny&, const CAny&) {
    ...
  }
};
```

   You can add any additional functionality to your `CAny` class.

Now you can use it like this:

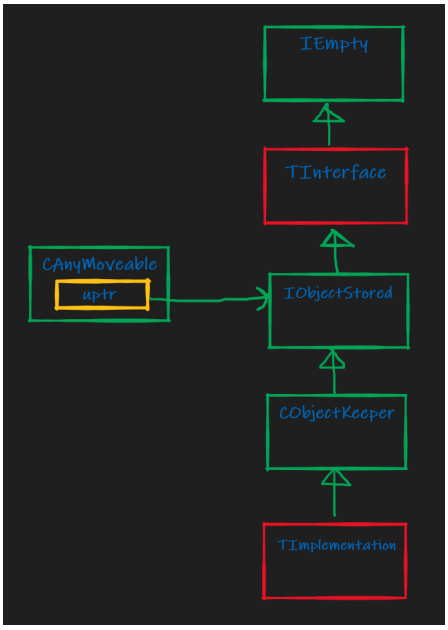
```
CAny x = 'c';
x->print();
x = std::move("123");
x->print();
x = 1.45;
x->print();
```

If you want to store an object of Type `R` and construct it on the fly from the data: `x, y, z`. Then use emplace function or emplace constructor like this:

```
CAny s;
s.emplace<R>(x, y, z);
CAny y(std::in_place_type_t<R>(), x, y, z);
```

Then the object of type `R` will be created without creation of intermediate objects.

Implementation Internally `CAnyMovable` stores an `std::unique_ptr` to interface `IObjectStored`. Let us look at the diagram below:



The red rectangles are templates provided by the user. `IEmpty` is an empty interface with virtual distructor. Its only purpose is to avoid user having to provide a virtual distructor in the `TInterface` template. Then `IObjectStored` adds some virtual methods to provide required functionality. `CObjectKeeper` is a template storing an instance of an object you want to move into `CAnyMovable`. `TImplementation` is a user defined template implementing all virtual methods from `TInterface`.