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).

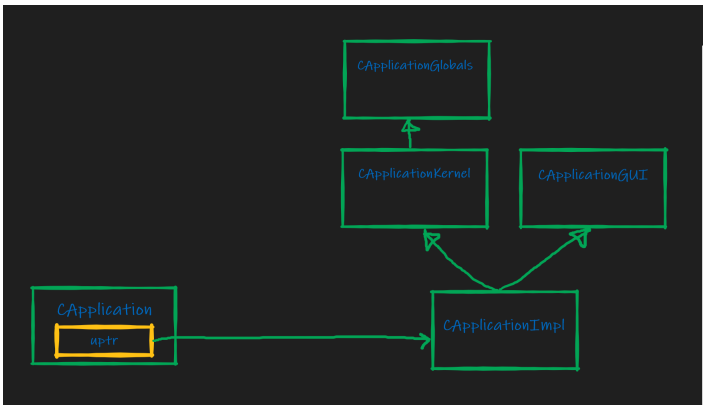
2 Application Structure

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:

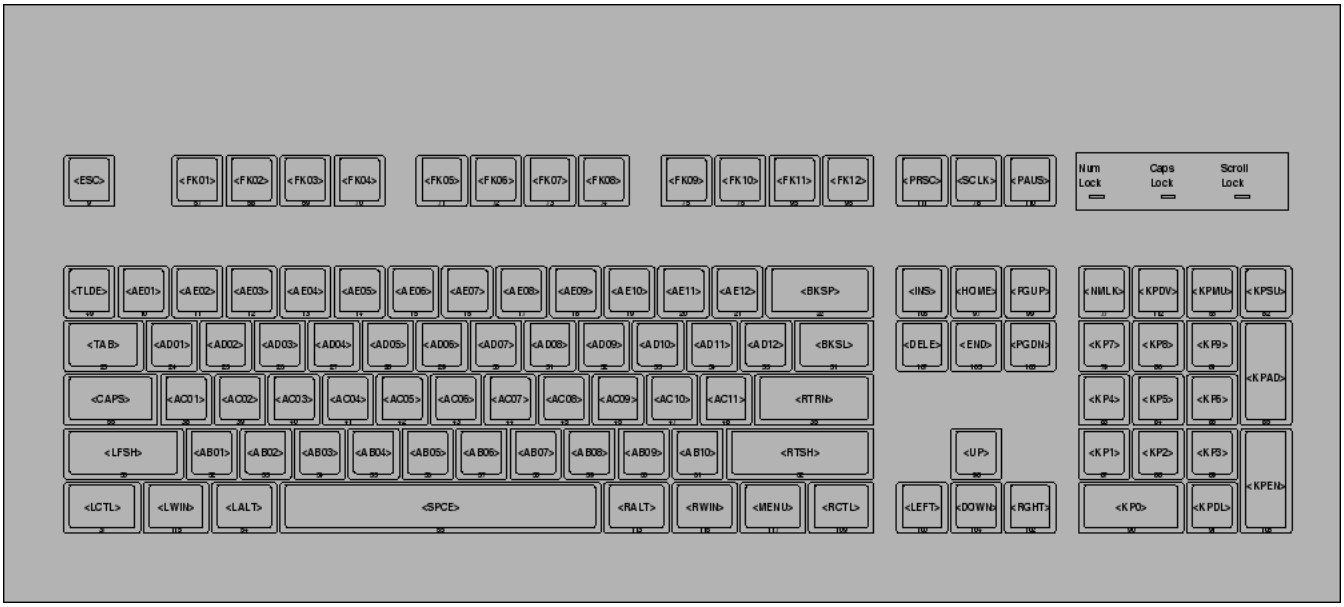
1. **CApplicationGlobals**. Its purpose is to initialize global resources, e.g., timers, loggers, thread pools, etc. Application initializes all global resources (basically singletons) at the start. This allows not to wast time on the first call. Also, it is inconvenient to initialize timers via the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

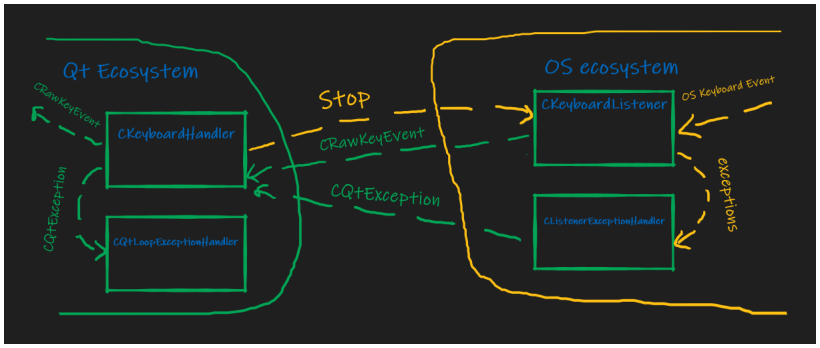
2.3 Keyboard Interception

The application must intercept keyboard events system-wide even if application is on the background. Qt ecosystem does not allow doing this. Hence, we need to implement our own mechanism. System wide interception usually means raw keyboard data is being intercepted. Hence, we need to implement translation of the key events to generated text. The main problem here is dead keys and ligatures. Current implementation supports dead keys but not ligatures (but the design allows to extend implementation to ligatures). Since system-wide interception must be done on the level of current OS, we implement these mechanism separately for each supported OS. Currently list of the supported OS

1. Windows 8 or higher.
2. Linux with X11 (there is a working prototype, not implemented yet, need to specify all the details).
3. macOS 10.14 (Mojave) or higher (there is a working prototype, not implemented yet, need to specify all the details).

Current design `CKeyboardHandler` serves as an object receiving all key events in a system independent form. This is a singleton described in section 2.6.2.

Let us consider the following figure:



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

`CKeyboardHandler` acts on the main GUI thread. It lives in Qt ecosystem and spawns an additional thread while construction. This additional thread lives in the ecosystem of OS. The main object acting on the additional thread is `CKeyboardListener`. Its roles are:

1. Intercept key events system-wise
2. Extract all the information from OS key events and translate it to OS independent form.
3. Resend OS independent key events to `CKeyboardHandler` via Qt event system.
4. It receives only one message from `CKeyboardHandler` via OS ecosystem message system. This is a "stop" message. Receiving the message it stops the additional thread.

The second object acting on the additional thread is `CListenerExceptionHandler`. Its roles are:

1. Extract exception information.
2. Resend special exception messages to `CKeyboardHandler`.

If an exception happens the event loop on the additional thread stops. `CKeyboardHandler` resend the message to `CQtEventLoopExceptionHandler`. The latter one shows the error message and terminates the program.

2.4 System independent Key event

The `CKeyEvent` contains the following information (TO DO not implemented, subject to change):

1. Pressing time.
2. Releasing time.
3. `CKeyPosition` is an identifier of the Key position on the keyboard (independent on the layout this is a physical instance of a key).
4. `CKeyId` is an identifier of a Key depending on layout. It is based on Windows Virtual Key table.
5. Key label. This is a symbol giving a human readable name for a key. For example, a letter or a number written on a keyboard, or a special symbol for system keys like shifts, backspace, enter, etc.
6. Key Text. This is a sequence of symbols appearing after pressing this key. It can be 0, 1, or 2 symbols. The exact rules are explained below.
7. Flags (subject to change). These flags are reserved and can be needed to pass some additional information.

`CKeyboardHandler` receives the following structures from `CKeyboardListener`:

1. `CKeyPressing`.
2. `CKeyReleasing`.

The structure `CKeyPressing` contains the following fields:

1. `CTime` The pressing time of the key.
2. `CKeyPosition` The OS independent position of the key.
3. `CKeyID` The OS independent ID of the key.
4. `QChar` The key label. This symbol is shown on the keyboard.
5. `QString` The text generated by the key.

The structure `CKeyReleasing` contains the following fields:

1. `CTime` The releasing time of the key.
2. `CKeyPosition` The OS independent position of the key.
3. `CKeyID` The OS independent ID of the key. (subject to change, probably need to exclude this)

When we need to pair the pressing and releasing events into one key, we identify the key by its position. It should be noted that the ID of the key can change. For example, if we press a number on the numpad (keep pressing), then press NumLock, and only then release the numpad number key, the numpad key will have different IDs while pressing and releasing. This happens because NumLock changes the mapping for physical keys to the corresponding IDs.

2.5 Key mapping

This section is mostly influenced by a series of blogs by Michael S. Kaplan. His posts explain all subtleties of the keyboard handling on Windows. However, the principals he explains are applicable to any keyboard handling on any OS.

We separate keys to the following categories:

1. Producing symbols keys (not necessarily printable symbols).
2. Shifters. These are: Shift (left, right), Ctrl (left, right), Alt (left, right), Capslock.
3. Ignorable. All other keys.

The key Capslock has a toggled state. Also NumLock and ScrollLock have toggled state but we ignore them.

The symbol appearing on the screen depends on the sequence of keys pressed and not just one key. There are several reasons for that: dead keys or chained dead keys. I am currently focused on the dead key handling but keep in mind chained dead keys and other possible features for the future implementation. In order to handle a key pressing we need to know:

1. Key.
2. Shifters.
3. Layout.

These triples may be classified as follows:

1. Undefined. This combinations do not produce any symbols and are ignored by symbol producing system.
2. Control. This combinations do not produce any symbols and are ignored by symbol producing system. However, this combinations may copy and past some text. We cannot handle these events and just ignore them. They are usually combinations of Ctrl or Ctrl+Shift with some other keys.
3. Printable Key. This keys usually produce a single symbol. There are cases when they produce a ligature, that is, a sequence of two UTF-16 characters combined into one symbol on print (ligatures currently are not supported but the support is possible).
4. Printable Dead Key. This are special keys. They do not produce symbol on its own but can be combined with other keys if compatible. A symbol producing system has a special buffer to store a previous dead key. This dead key is flushed from the buffer if the symbol producing system accepts one of the next keys. Theoretically, it is possible to use chained dead keys, that is, to store a sequence of dead keys and then compose them with other characters. However, this feature is not considered and is not implemented.
5. Non-printable Key. (On Windows they are: cancel, backspace, tab, enter, esc). These keys are a pain in the neck. They behave differently in different Text Editors.

These triples behave as follows.

1. Undefined. This combinations do not produce any symbols and are ignored by symbol producing system.
2. Control. This combinations do not produce any symbols and are ignored by symbol producing system.
3. Printable Key. This key behaves as follows:
   (a) If there is no active dead key (the dead key was not pressed before this key or was flashed from the buffer of the symbol producing system). In this case Printable Key result in a single printable symbol. The dead key is flushed from the buffer.
   (b) If there is an active dead key (the dead key was pressed and is in the buffer of the symbol producing system). If Printable Key is compatible with the dead key they result in a single composed character. In other case, they produce a pair of symbols: a symbol for the dead key and a symbol for the Printable Key. The dead key is flushed from the buffer.
4. Printable Dead Key. This key behaves as follows:

(a) If there is no active dead key (the dead key was not pressed before this key or was flashed from the buffer of the symbol producing system). This dead key is stored in a buffer of the symbol producing system. No symbol is generated.

(b) If there is an active dead key (the dead key was pressed and is in the buffer of the symbol producing system). The dead keys are never compatible. They produce two symbols: a symbol for the first dead key and a symbol for the second dead key. The dead key is flushed from the buffer.

5. Non-printable Key. (On Windows they are: cancel, backspace, tab, enter, esc). This key behaves as follows:

   (a) If the key is Enter (Tab). In case of an active dead key, it produces two symbols: a symbol for the dead key and a new line symbol (Tab symbol), then it flushes the dead key from the buffer. In case of no active dead key, it produces a new line symbol (Tab symbol).

   (b) Any other key. It flushes dead key from the buffer if any and produces no symbols.

2.5.1 Keyboard Interception on Windows

`CKeyboardListenerWin` substitutes `CKeyboardListener` on Windows. This object is addressable, hence it stores a unique pointer to its implementation. When created it does the following operations:

1. It register and create a background message Window.
2. It provides a hook to RAW INPUT system with global keyboard interception.
3. It sends a special Killer object to `CKeyboardHandler` via `std::promise` to receive "stop" messages.
4. It connects its Qt signals to the corresponding Qt slots of `CKeyboardHandler` to send messages about key events.

This object starts a Windows message loop on its thread and react to two messages:

1. `WM_INPUT`. It corresponds to RAW INPUT mechanism.
2. `WM_STOP_LISTENING`. This is an application defined event for the "stop" message.

In the event loop

1. On `WM_STOP_LISTENING` message. It posts the Quit message to terminate the Windows thread loop. Since it receives such a message the main thread is about to terminate the application, hence we do not need to execute any additional steps.
2. On `WM_INPUT` message. It performs the following steps:

   (a) Get time of the message. There are two options: get current time from the global Timer or get OS provided time. I stick to the first option to have one consistent time line in the application. (TO DO currently not implemented).

   (b) Extract `RAWKEYBOARD` data from the OS key event.

   (c) Compute `CKeyPosition` and if it is unknown, then return.

   (d) Compute `CKeyID` and if it is unknown or ignorable, then return.

   (e) Check if the event is pressing or releasing:

      i. If pressing. It computes key text and key symbol (the symbol denote the key on the keyboard), then it generates `CKeyPressing` structure and sends the corresponding Qt signal to `CKeyboardHandler`.

      ii. If releasing. It generates `CKeyReleasing` structure and sends the corresponding Qt signal to `CKeyboardHandler`.

Structure of CKeyboardListenerWin This section is subject to change. `CKeyboardListenerWin` contains an `std::unique_ptr` to `CKeyboardListenerWinImpl`. The latter object contains:

1. static method `WndProc`.
2. `CWinRawInputHook KeyboardHook_`.
3. `CRawInputReader RawInputReader_`.
4. `CKeyPositionWin KeyPosition_` .
5. `CKeyTextMaker KeyTextMaker_`.

`WndProc` is a windows procedure. It is used by message window to react to messages. This function calls an appropriate method of `CKeyboardListenerWinImpl`.

`CWinRawInputHook` provides integration of `CKeyboardListenerWinImpl` into Windows ecosystem. It registers and creates a message non-gui window and then register this window in RAW INPUT system to receive all keyboard messages system-wide.

`CRawInputReader` provides a convenient way of extraction `RAWKEYBOARD` data from the key event.

`CKeyPositionWin` computes `CKeyPosition` for a key.

`CKeyTextMaker` gets a text produced by the current key event and also produces labels for keys.

Known issues In order to get a symbol from a key event `KeyTextMaker_` needs to know the current keyboard layout. It gets the layout of the current foreground window. However, certain windows do not receive broadcast messages, hence are unaware of the layout changes. For example, all console windows, calc.exe, etc. If you type while a console window has focus and switch the layout, the application will not notice this. This is a well known issue. This happens because a console application is not a window from the OS point of view. It has a parent window that does receive the broadcast messages, however it is a complicated task to find out the handle to the window. I am leaving this issue as it is for now.

2.5.2 Keyboard Interception on Linux

TO DO

2.5.3 Keyboard Interception on macOS

TO DO

2.6 Global Objects

2.6.1 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.

2.6.2 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:

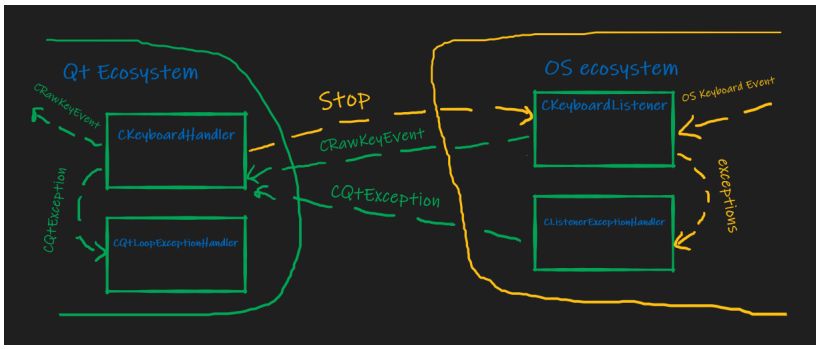
   (a) OS key events.

   (b) CKeyboardHandler stop event.

   CKeyboardListener intercepts any key events in OS, transform system dependent key events into CRawKeyEvent-s (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.

2.7 Kernel

2.8 Qt Resources

2.9 ApplicationImpl

3 Code

4 Implementation

5 Library

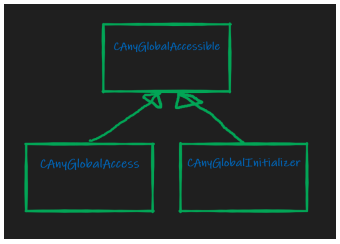
5.1 Singleton

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:

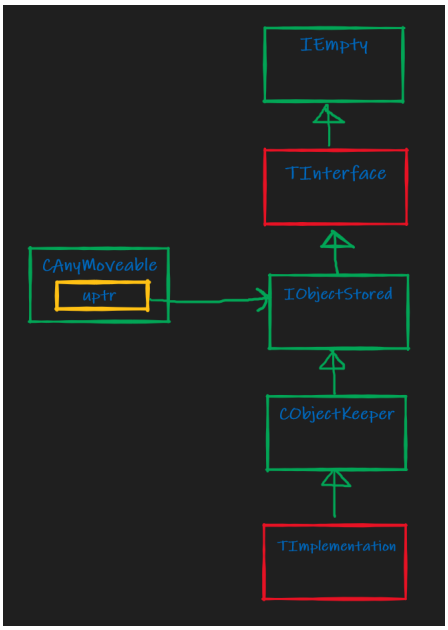
```
CAny x = 'c';
x->print();
x = std::move("123");
x->print();
x = 1.45;
x->print();
```

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`.