EMBSYS 110 Module 2

Design and Optimization of Embedded and Real-time Systems

1 C++ for Embedded Systems

According to a survey by IEEE Spectrum (https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages), the 10 most popular programming languages in 2018 are:

1. Python
2. C++
3. Java
4. C
5. C#
6. PHP
7. R
8. Javascript
9. Go
10. Assembly

C and C++ have kept their positions in the top 5 after all these years (Note – C++ and C swapped their positions from 2017 survey). In fact C and C++ remain dominant in the embedded fields. The main reason is that while supporting high level constructs they are relatively efficient in interfacing with hardware (bit manipulation, register access, interrupt handling, DMA, etc.).

While some C++ features are costly and should be avoided in resource-constrained embedded systems, many object-oriented programming features offer great benefits with little extra overhead. In my projects I use a minimal subset of C++ to take advantages of its OOP support, namely encapsulation, inheritance and polymorphism.

Each of these key OOP concepts will be explained with examples in the following sections. Before that let's discuss a few general topics related to C++.

1.1 Memory Allocation

This is a debate between using static memory allocation and dynamic memory allocation. Conventional wisdom in embedded fields suggests that static memory allocation is preferred to dynamic allocation. Even in those cases where we do need to have dynamic memory allocation, we prefer using a custom block-based memory pool to the built-in heap.

1.1.1 Why Static?

Maybe the question should be "Why Dynamic?" Dynamic allocation is helpful when

1. You don't know how big your data structure (array, map, etc) is at compile-time, and you want to be able to build it at run-time based on the actual size required, and can grow or sink as needed.
2. You don't know which type (derived classes) of components to instantiate at compile-time, and you want to be able to create them at run-time based on the actual configuration.

Here are some counter arguments against these two points:

1. Embedded systems are often hardware-centric and by its nature such systems are much less dynamic in terms of the number and type of objects that are required. Once the hardware is designed, it is fairly stable and is unlikely to drastically change at run-time.

   Note it is a different story that some components can be scaled down or disabled due to configuration. This can be managed via component states.
2. For reliability embedded designers often need to consider the worst case scenario. It's true that on average dynamically memory location is more efficient since it does not reserve memory unneeded at the moment (and hence leaves it available for other purposes). However the system may run out of memory when the worst case scenario comes up.

   In order to ensure the system will always get the memory it needs, even in the worst case scenario, it will need to allocate the maximum required memory upfront, i.e. static allocation.
3. Again for reliability, a whole class of problems such as memory leak, dangling pointers will be gone without dynamic memory allocation. Software can be less complex (less to consider) and easier to debug (object addresses listed in map file).

   New features (e.g. shared_ptr) were invented just to overcome this class of problems. They come at a cost though (e.g. run-time overhead, readability, cyclic references, learning curve, etc.)

   Some development practices for medical devices outlaw dynamic memory allocation entirely!
4. DMA can be simpler with contiguous memory at predefined (linker-controlled) addresses.

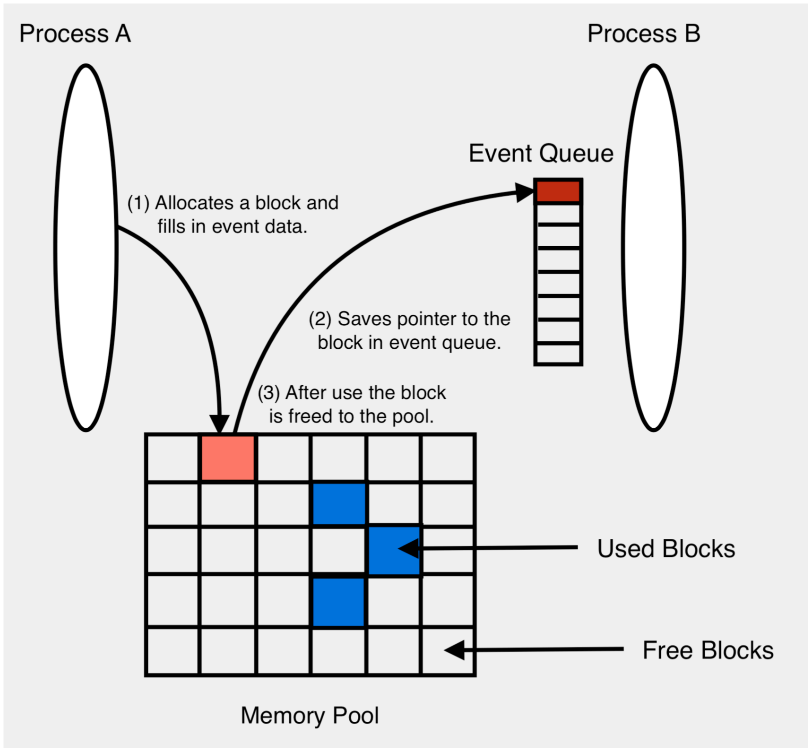
1.1.2 Memory Pool

There are cases when dynamic memory is desirable. Examples include passing events between two objects via an event queue, and handling messages sent or received over the network. The main reason is that events/messages are dynamic in nature as they may arrive in any number at any time.

If we don't use dynamic memory we would have to copy the entire event/message into and out of the event queue/message buffer. With dynamic memory we can allocate memory for an event/message as it comes up, pass its pointer around, and free up the memory after the event/message has been handled.\

In embedded systems it is very common to use custom block-based memory pools to manage dynamic memory explicitly, rather than using the built-in heap-based allocator in the standard library. QP provides an efficient implementation of memory pools for event passing and other general application uses. We will look at it closer when we introduce QP later.

(See 6.5.7 Memory Pools of PSiCC2 book.)


Process A
Process B
Event Queue
(1) Allocates a block and fills in event data.
(2) Saves pointer to the block in event queue.
(3) After use the block is freed to the pool.
Used Blocks
Free Blocks
Memory Pool

1.2 Standard Template Library (STL)

Another decision to make is whether to use STL or not. In the past when we worked with 4-bit or 8-bit processors, it was common that people avoided even the standard C library so they would write their own version of memcpy, strncpy, etc. Nowadays few people do that anymore.

C++ STL is a bit different since it uses dynamic memory internally and as we said we are trying to avoid using dynamic memory in embedded systems (especially with microcontroller).

That said, modern STL are highly optimized and it often allows you to provide your own allocator or preallocate memory for its container types. Indeed it seems to be a better choice than reinventing it yourselves especially when you are dealing with large data sets.

My suggestion is that STL can be used judicially. In my projects, I didn't use it much simply because:

1. I developed my own pair, map, string, array, pipe and fifo classes to illustrate basic OOP

concepts. I used them to illustrate how to manage memory statically.

2. The data sets I am dealing with are relatively small, and the simple algorithms in my own implementation are sufficient.
3. I tried to reduce external dependency to make it more portable (e.g. to make it usable on platforms where STL is not supported).

1.3 C++ Exception

Many believe C++ exception should be avoided in embedded systems and I reckon that. In earlier generations C++ exception imposed a rather hefty run-time cost even when exception did not occur. Over the years it may have been optimized and now could be a less concern.

Nevertheless the main reason why C++ exception is not a good choice for embedded systems is that by the time when an exception is caught, after an arbitrary number of propagation, its handler rarely knows the *context* (or states) in which the exception is thrown, not to mention what it should do to restore the system states to a good configuration to continue proper operation.

In other words, without the concept of states, it is very hard to provide a clear description of how a try...catch type of exception handling affects system behaviors. It is not surprising that most catch blocks simply dump some debug messages and quit the applications. It is not good enough for embedded systems in which reliability is a priority.

Later we will see alternatives to C++ exception:

1. Assert to catch program bugs or other non-recoverable faults.
2. State-based exception handling in which how it affects system behaviors are clearly and precisely described in statecharts.

   (See Section 3.7.2 State Machines and C++ Exception Handling in PSiCC2 book.)

1.4 Template

Template should also be used judicially. Template enables generic or meta-programming and automates a lot of programming tasks. However it tends to make the resulting code harder to read and debug. Compiler errors are also harder to digest. Code bloat can be an issue if it is not used carefully. In my projects template is used minimally.

Since STL depends on template a lot, we should at least be comfortable with it so we can use it when necessary.

1.5 C++98 vs C++1x

C++1x (11/14/17) offers many new features that makes it look like a different language from the previous 98 version. They include:

1. Smart pointers (shared_ptr, unique_ptr, weak_ptr) to support garbage collection.
2. Automatic type deduction, Rvalue-references and move semantics
3. Functional programming support such as lambda functions, promises and futures.

The list goes on. These new features help bring C++ up to the modern programming paradigms.

In my projects I pretty much stick with the 98 standard for the following considerations:

1. To keep the learning curve gentle, since many developers in the embedded fields are still using C.
2. New features solve old problems but also creates new ones. At least there are a few caveats to be aware of when using those features. Make sure you read and *understand* the "Effective Modern C++" book by Scott Meyers first. (Available in UW library online.)
3. New features add complexity to the syntax and rules of C++ if they were not complex to begin with.

   As it turns out, when we use QP in embedded applications many of the problems that those new features are trying to solve simply do not exist – problems that arise from wide-spread use of dynamic memory allocation and functional API with callbacks (i.e. passing objects and callbacks around). The event-driven paradigm in QP for asynchronous communications sees little use of *promises* and *futures* added in C++1x.

2 Encapsulation

2.1 Class

Before OOP, we use structure to group related data together. For example if we want to have a key-value pair, we would have

```
typedef struct {
    uint32_t key;
    uint32_t value;
} KeyValue;
```

Functions that operate on a structure are separated from its type type definition. An explicit pointer to the structure to be operated on is passed as one of the arguments to the function. For example, you may have

```
uint32_t GetValue(KeyValue const *kv) {
    return kv->value;
}
```

C++ makes it easier with classes. Related data and functions operating on those data are grouped together in a single entity called *class*. An instance of a class is called an *object* of that class. Classroom textbooks usually introduce the concept of objects with examples related to our real world, such as a table, a chair, a bank account. In practice objects often refer to some abstract concepts that are harder to grasp. For example an object can be a key-value pair, a FIFO, an event, a state-machine, an active object or the entire framework.

Using class, the key-value pair becomes

```
template <class Key, class Value>
class KeyValue {
public:
    KeyValue(Key k, Value v) : m_key(k), m_value(v) {}
    KeyValue() {}
    virtual ~KeyValue() {}

    Key const &GetKey() const { return m_key; }
    Value const &GetValue() const { return m_value; }
    void SetKey(Key const &k) { m_key = k; }
    void SetValue(Value const &v) { m_value = v; }

protected:
    Key m_key;
    Value m_value;
};
```

Here we explore the uses of:

1. Class definition with the keyword *class* and basic template.
2. Accessibility control with *public*, *protected* and *private*.
3. Data members (m_key and m_value).
4. Member functions (GetKey() and GetValue()).
5. Pass or return by reference.
6. Constructors and destructors.

To define an instance of the class KeyValue, you will write

```
typedef KeyValue<Hsmn, uint16_t> HsmnPin;

HsmnPin kv(USER_BTN, GPIO_PIN_13);

uint16_t pin = kv.GetValue();
```

2.2 Composition

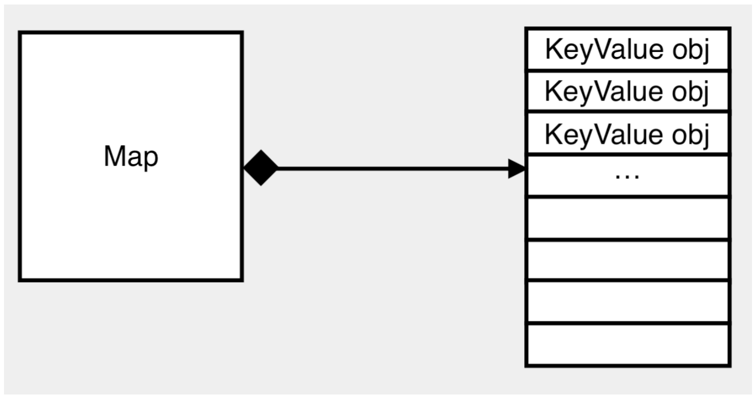
An object can contain (be composed of) other objects. In other words, we can use another class as the type of a data member. For example having defined the KeyValue class, we could build a basic key-value *map* by composing an array of KeyValue objects in the map class.

In the following example, we add a little trick to use a pointer in the map class to refer to the array of KeyValue objects rather that storing the array by value internally. The purpose of this is to place the responsibility of allocating the memory required for the array on the user of the map class. Ultimately we want to avoid using dynamic memory allocation inside the map class. This is a common technique in embedded systems and you will find in QP or uCOS-II source.

```
template <class Key, class Value>
class Map {
public:
    Map(KeyValue<Key, Value> kv[], uint32_t count,
        KeyValue<Key, Value> const &unusedKv) :
        m_kv(kv), m_count(count), m_unusedKv(unusedKv) {
        FW_MAP_ASSERT(m_kv && m_count);
        Reset();
    }
    virtual ~Map() {}

    void Reset() {
        FW_MAP_ASSERT(m_kv && m_count);
        for (uint32_t i = 0; i < m_count; i++) {
            m_kv[i] = m_unusedKv;
        }
    }

    KeyValue<Key, Value> *GetByIndex(uint32_t index) {
        FW_MAP_ASSERT(m_kv && (index < m_count));
        return &m_kv[index];
    }
    void ClearByIndex(uint32_t index) {
        FW_MAP_ASSERT(m_kv && (index < m_count));
        m_kv[index] = m_unusedKv;
    }
    // OK to pass unused key to find first empty slot.
    KeyValue<Key, Value> *GetByKey(Key const &k);
    bool ClearByKey(Key const &k);
    KeyValue<Key, Value> *GetFirstByValue(Value const &v);
    void Save(KeyValue<Key, Value> const &kv);
    void Put(uint32_t index, KeyValue<Key, Value> const &kv);

    KeyValue<Key, Value> const &GetUnusedKv() const { return m_unusedKv; }
    uint32_t GetUnusedCount() const;
    uint32_t GetUsedCount() const { return m_count - GetUnusedCount(); }
    uint32_t GetTotalCount() const { return m_count; }
    bool IsFull() const { return GetUnusedCount() == 0; }
    bool IsEmpty() const { return GetUsedCount() == 0; }

protected:
    KeyValue<Key, Value> *m_kv;
    uint32_t const m_count;
    KeyValue<Key, Value> const m_unusedKv;
};
```

Here we explore the uses of:

1. Composition by having a data member of another class type.
2. Inline functions.
3. Const functions.
4. External memory allocation.
5. More examples of member functions and pass or return by reference.



3 Inheritance

We have seen how to build more complex objects from simpler ones using composition (e.g. building Map from KeyValue). We are now going to look at another way of building objects – inheritance.

Inheritance allows us to derive a subclass from a base class, through which the subclass inherits members of the base class. Inheritance is transitive, i.e. there can be multiple levels of inheritance and the final derived class inherits from all the base classes including intermediate ones. It promotes code reuse since we don't need to rewrite anything that has already been defined in the base classes. We only need to code the *differences* by adding new members, or by overriding members already defined in the base classes.

3.1 Example of Inheritance

One very important concept is that composition is a “*has-a*” relationship while inheritance is an “*is-a*” relationship. In the previous example, we say a Map *has an* array of KeyValue entries. In the following example we shall see immediately, we say an Evt *is a* special kind of QEvt with additional members.

QEvt is the event class defined in QP. It represents an event to be handled by a state-machine. It looks like this:

```
class QEvt {
    public:
        //! public constructor (dynamic event)
        QEvt(QSignal const s) // poolId_/refCtr_ intentionally uninitialized
          : sig(s) {}
        virtual ~QEvt() {}

    public:
        QSignal sig; //!< signal of the event instance

    private:
        uint8_t poolId_;          //!< pool ID (0 for static event)
        uint8_t volatile refCtr_; //!< reference counter
    };
```

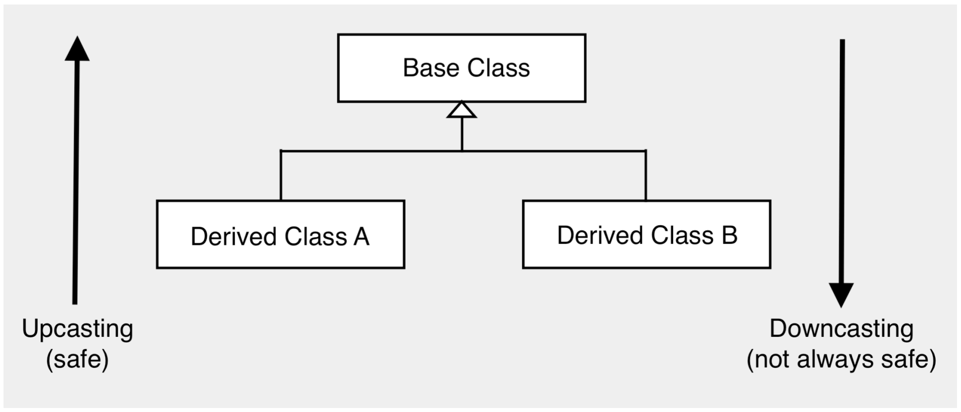
Basically QEvt carries a member named *sig* to indicate what type of event it is, along with some book-keeping information for garbage collection (such as poolId_ and refCtr_).

QEvt suffices for basic operations of a state-machine. However for robust system design we often need more information to come along with an event. For example we would need a sequence number to match response to request, as well as the source and destination of an event. Hence we derive the subclass Evt from QEvt to add those new members:

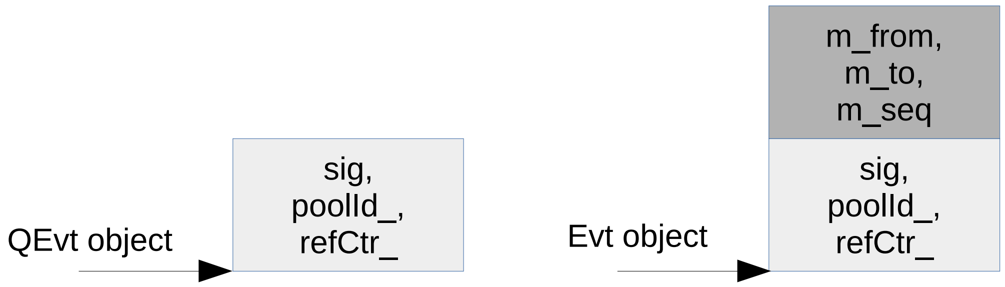
```
class Evt : public QP::QEvt {
public:
    static void *operator new(size_t s);
    static void operator delete(void *evt);

    Evt(QP::QSignal signal, Hsmn to, Hsmn from = HSM_UNDEF, Sequence seq = 0) :
        QP::QEvt(signal), m_to(to), m_from(from), m_seq(seq) {}
    ~Evt() {}

    Hsmn GetTo() const { return m_to; }
    Hsmn GetFrom() const { return m_from; }
    Sequence GetSeq() const { return m_seq; }

protected:
    Hsmn m_to;
    Hsmn m_from;
    Sequence m_seq;
};
```

We can see the members *m_to* (target), *m_from* (destination) and *m_seq* (sequence number) have been added, along with their accessor functions and a modified constructor.

Since Evt inherits from its base class QEvt, it automatically gets the members sig, poolId_ and refCtr_, even though they are not explicitly defined inside Evt. A memory view of both objects are shown below. As we can see, it is safe to convert/cast a pointer to a derived class (e.g. Evt object) to a pointer to a base class (e.g. QEvt object). This is called *upcasting*. It works because the memory layout of the bottom/base part of the Evt object looks identical to that of an actual QEvt object.



It is important to note that the reverse, called *downcasting*, is not necessarily safe. That is we cannot blindly convert/cast a pointer to a base class to a pointer to a derived class. We *must* be sure that the object pointed to by the base class pointer is indeed a derived class object. In embedded system, RTTI (Run-time Type Identification) is not encouraged due to its overhead. The type of an object is typically identified by an explicit data member of the base class. In the case of QEvt, it is the member *sig* that help identify the actual class of an event object.



It is programmers' responsibility to ensure correctness when performing downcasting. We have to let the compiler know that we are absolutely sure by applying a *static_cast*<>operator explicitly. In the following example, the code checks for a timer event with the macro IS_TIMER_EVT(e->sig) and downcast the event object e from the base class *QEvt* to the timer event class *Timer*.

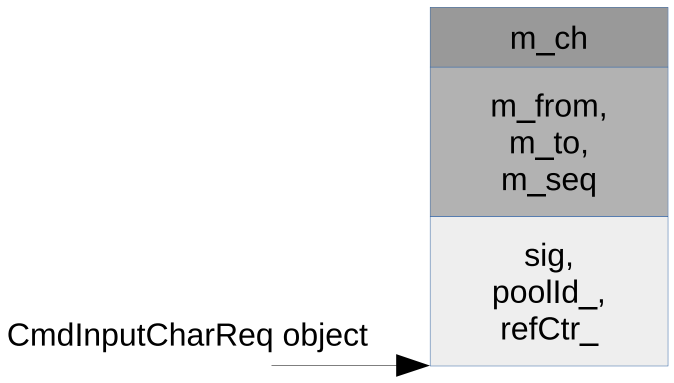
```
void Active::dispatch(QEvt const * const e) {
    if (IS_TIMER_EVT(e->sig)) {
        Timer const *timerEvt = static_cast<Timer const *>(e);
        hsmn = timerEvt->GetHsmn();
    }
    ...
}
```

3.2 Additional Level of Inheritance

With the derived class Evt, common attributes such as the source, destination and sequence number are now included in an event object, along with the event type (sig) already defined in the base class Evt. This satisfies many use cases. However it is not difficult to foresee that certain specific event types need to carry with them additional parameters.

For example, an event passing a received character to the command input handler obviously need to contain the character to be handled. We achieve this by further deriving CmdInputCharReq from Evt as:

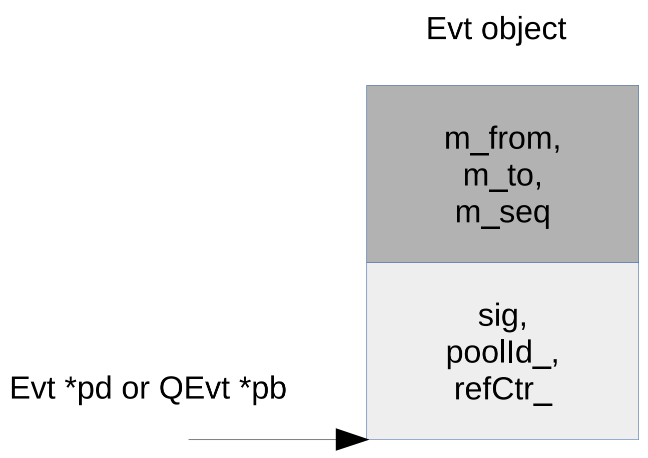
```
class CmdInputCharReq : public Evt {
public:
    CmdInputCharReq(Hsmn to, Hsmn from, char ch) :
        Evt(CMD_INPUT_CHAR_REQ, to, from), m_ch(ch) {}
    char GetCh() const { return m_ch; }
private:
    char m_ch;       // Character to be processed.
};
```



4 Polymorphism

4.1 Pointer to Base Class (Upcasting)

Before we discuss polymorphism, let's review a previous example illustrating upcasting. Previously we have shown how to upcast a pointer to a derived class to a pointer to a base class. We have QEvt as the base class and Evt as the derived class. Given a pointer to an Evt object, we can safely refer to the same object with a pointer to QEvt. This is illustrated in the figure below, in which the Evt object is safely referred to by either *pd* (pointer to derived class) or *pb* (pointer to base class).



In the code fragment below, we create an Evt object (*DONE* event) and point to it with *pd* which is a pointer to the derived class (Evt). In the next line the assignment of *pd* to *pb* automatically upcast *pd* to a pointer to the base class (QEvt).

```
Evt *pd = new Evt(DONE, GET_HSMN());
QEvt *pb = pd;
```

What is the advantage of upcasting? Imagine that the event framework provided by QP has some useful functions to work on events. Those functions know about QEvt since QEvt is part of QP but they have absolutely no ideas about our derived class Evt (in which we added source, destination and sequence number). Those event functions in QP are generic and should apply to all kinds of events derived from QEvt. All those functions care about are members of the base class like *sig*, *poolId_* and *refCtr_* that are inherited by any derived classes, but they don't care about members that we added (i.e. *m_from*, *m_to* and *m_seq*). With upcasting, we can safely and conveniently use those generic functions as is for our own event classes derived from QEvt.

Here is an example of such generic function provided by QP:

```
bool QActive::post_(QEvt const * const e, uint_fast16_t const margin);
```

post_() takes a parameter *e* which is a pointer to the base class for all events (i.e. QEvt). It puts *e* into the event queue of an active object regardless to the actual type of event pointed to by *e*. Here is an example of how *post_()* is used in our application framework:

```
void Fw::Post(Evt const *e) {
    QActive *act = m_hsmActMap.GetByIndex(e->GetTo())->GetValue();
    if (act) {
        act->post_(e, 0);
    } else {
        QF::gc(e);
    }
}
```

So far we have not touched polymorphism yet, since those generic functions work in the *same* way for all the derived classes. They are very useful for being generic. However there will be situations in which we want the way they work to depend on the actual derived classes of the objects passed in to those functions. Virtual functions will help us achieve this.

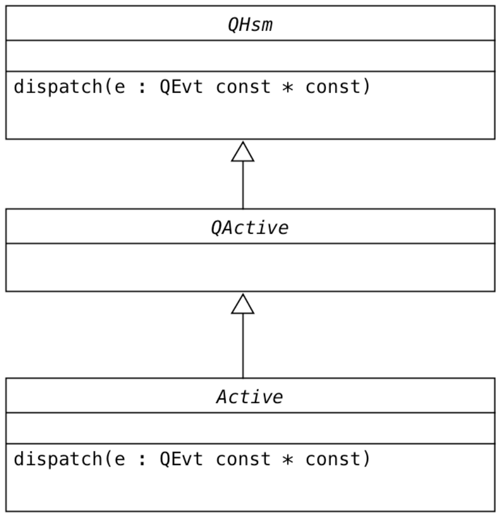
4.2 Virtual Functions

A virtual function is a member function declared with the keyword *virtual* in a base class. Like any member functions of a base class, a virtual function can be overridden by a derived class. The unique feature of being virtual is that when it is invoked on an object via a pointer to the base class, the code at run-time will call the overriding function in the *actual* derived class of the object if one is defined. If a virtual function is *not* overridden by the derived class, the version in the base class or one of the intermediate base classes (if any) will be invoked.

For example in QP there is a base class for hierarchical state-machines named QHsm. It has a virtual member function *dispatch()* which dispatches an event to the state-machine.

```
class QHsm {
public:
    virtual void dispatch(QEvt const * const e);
    …
}
```

Being *virtual*, it allows a derived class to override the base version with a new version to do different things in the function. In our project we have our framework *active object* class *Active* derived from the QP-provided *QActive* which is in turn derived from *QHsm*. *Active* represents a more sophisticated hierarchical state-machine than the basic ones provided by QP.


QHsm
dispatch(e : QEvt const * const)
QActive
Active
dispatch(e : QEvt const * const)

As shown in the class diagram above, we override QHsm::dispatch() with Active::dispatch(). However we do not want to completely rewrite dispatch() since the base version has already implemented the complex logic of of a hierarchical state-machine. We only want to supplement it with additional features such as supporting different types of events, orthogonal regions, etc. It turns out we can call the base version of the virtual function within the derived version by using a scope resolution operator (::), i.e. QHsm::dispatch().

Below is the derived version of dispatch() showing how it modifies, but still reuses, the base version of the function to provide more sophisticated features.

```
void Active::dispatch(QEvt const * const e) {
    Hsmn hsmn;
    // Discard event if it is sent to an undefined HSM.
    if (!IS_EVT_HSMN_VALID(e->sig)) {
        return;
    }
    if (IS_TIMER_EVT(e->sig)) {
        Timer const *timerEvt = static_cast<Timer const *>(e);
        hsmn = timerEvt->GetHsmn();
    } else {
        Evt const *evt = static_cast<Evt const *>(e);
        hsmn = evt->GetTo();
    }
    if (hsmn == m_hsm.GetHsmn()) {
        QHsm::dispatch(e);
        // Handle all reminder events generated as a result of e.
        m_hsm.DispatchReminder();
    } else {
        HsmnReg *hsmnReg = m_hsmnRegMap.GetByKey(hsmn);
        if (hsmnReg && hsmnReg->GetValue()) {
            hsmnReg->GetValue()->dispatch(e);
        }
    }
}
```

Note – The above code is to illustrate the concept of virtual function. We don't need to understand the details of the logic here.

Ultimately we are ready to appreciate the beauty of polymorphism. This is what we have seen so far:

1. QActive is the basic active object derived from QHsm (hierarchical state-machine).
   Active is our more sophisticated active object derived from QActive.
2. QHsm has a virtual function named dispatch().
3. QActive *does NOT* override dispatch() but *inherits* it from QHsm.
4. Active overrides dispatch() to provide additional features while reuses the base version.

Having spent so much time talking about dispatch(), we haven't yet seen where it is called. Indeed, dispatch() is at the very core of QP. QP being an event-driven framework its most fundamental job is to process events. It gets an event from the event queue of an active object and dispatches it to the active object to handle. The code for this takes barely two lines as shown below:

```
void QXK_activate_(void) {
    QP::QActive *a;
    ...
    a = QP::QF::active_[p]; // obtain the pointer to the AO
    QP::QEvt const *e = a->get_();
    a->dispatch(e);
    QP::QF::gc(e);
    ...
}
```

With "*a"* pointing to the intermediate base class QActive and *dispatch(e)* being a virtual function, the call "*a->dispatch(e)"* invokes either:

1. The overriding function defined in the derived class of the active object pointed to by "a".

   If "a" points to an object of the derived class Active, the overriding function Active::dispatch() will be called to provide more sophisticated features.

2. The base version defined in QHsm through inheritance.

   If "a" points to QActive, since QActive does not override dispatch(), the inherited QHsm::dispatch() will be called to provide the basic features.

The beauty is the decision is made at run-time (called dynamic binding) and the code for that decision is generated by the C++ compiler. This frees up programmers' mind to focus on higher level concepts and logic. It also allows a library to support an open extension by its user application without modification to the library itself. By this principle I managed to extend QP's features without modifying QP itself, apart from bug fixes and unrelated minor changes.

Speaking of higher level concepts, we will soon launch ourselves above the coding level into the space of *design*. We will experience how statechart, a visual language, further frees up our mind to let us see things that are not obvious in source code, such as behaviors, interactions and reliability.

5 Project Layout

Refer to Project Explorer in Eclipse. Main directory structure is:

1. **qpcpp** – QP source code version 6.3.1 downloaded from https://www.state-machine.com.
2. **framework –** Custom application framework on top of QP.
3. **system** – Low-level libraries including CMSIS, STM32Cube HAL drivers, etc.

   Important files include *src/cortexm/exception_handlers.c* (exception handlers in C) and *src/cmsis/startup_stm32f401xe.S* (exception vectors).
4. **ldscripts –** Linker scripts.
5. **include** – Application specific include files.
   (a) app_hsmn.h – Definition of *HSM numbers* (or ID) and *priorities*.
   (b) bsp.h – Low-level board-support functions.
   (c) periph.h – Global STM32 peripheral configurations (GPIO, Timers, Clocks).
   (d) stm32f4xx_hal_conf.h – STM32Cube configurations.
   (e) stm32f4xx_it.h – Cortex-M and STM32 interrupt-service routines (ISRs.)
6. **src** - Application specific source files.
   (a) bsp.cpp – Low-level board-support functions.
   (b) main.cpp – Entry point of the application program.
   (c) periph.cpp – Global STM32 peripheral configurations (GPIO, Timers, Clocks).
   (d) stm32f4xx_it.cpp – Cortex-M and STM32 interrupt-service routines (ISRs.)
   (e) **Active object subdirectories –** Each subdirectory corresponds to an *active object*.

   For example, they include **System**, **Disp**, **Sensor**, **UserLed**, etc. For now, think of each active object as a thread. The internal structure of each active object subdirectory will be introduced later.