RECIPE:
Simple Encapsulation and Inheritance in C

Encapsulation

Encapsulation is the ability to package data with functions into classes. This concept should actually come as very familiar to any C programmer because it's quite often used even in the traditional C. For example, in the Standard C runtime library, the family of functions that includes fopen(), fclose(), fread(), and fwrite() operates on objects of type FILE. The FILE structure is thus **encapsulated** because client programmers have no need to access the internal attributes of the FILE struct and instead the whole interface to files consists only of the aforementioned functions. You can think of the FILE structure and the associated C-functions that operate on it as the FILE **class**. The following bullet items summarize how the C runtime library implements the FILE "class":

- Attributes of the class are defined with a C struct (the FILE struct).

- Methods of the class are defined as C functions. Each function takes a pointer to the attribute structure (FILE *) as an argument. Class methods typically follow a common naming convention (e.g., all FILE class methods start with prefix f).

- Special methods initialize and clean up the attribute structure (fopen() and fclose()). These methods play the roles of class constructor and destructor, respectively.

This is exactly how QF/C implements classes. For instance, the following snippet of QF/C code declares the QActive (active object) "class". Please note that all class methods start with the class prefix ("QActive" in this case) and all take a pointer to the attribute structure as the first argument "me":

```
typedef struct QActiveTag QActive;              /* Active Object base class */
struct QActiveTag {
    QHsm super_;    /* protected member super (inheritance from class QHsm) */
    . . .
    uint8_t prio__;                /* private priority of the active object */
};

/* public methods */
int QActive_start(QActive *me, uint8_t prio,
                  QEvent *qSto[], uint16_t qLen,
                  void *stkSto, uint32_t stkSize,
                  QEvent const *ie);
void QActive_postFIFO(QActive *me, QEvent const *e);
void QActive_postLIFO(QActive *me, QEvent const *e);

/* protected methods ...*/
void QActive_ctor_(QActive *me, QPseudoState initial);
void QActive_xtor_(QActive *me);
void QActive_stop_(QActive *me);    /* stopps thread; nothing happens after */
void QActive_subscribe_(QActive const *me, QSignal sig);
```

Most object-oriented designs distinguish the following levels of protection.

- Private — accessible only from within the class

- Protected — accessible only by the class and its subclasses

- Public — accessible to anyone (the default in C)

The recommended convention is to use the double-underscore suffix (`prio__`) to indicate private attributes and the single-underscore suffix (`super_`, `QActive_subscribe_()`) to indicate protected members. Public members do not require trailing underscores (`QActive_start()`).

Inheritance

Inheritance is the ability to define new classes based on existing classes in order to reuse and organize code. QF/C implements single **inheritance** by literally embedding the parent class attribute structure as the first member of the child class structure. As shown in the following figure, this arrangement lets you treat any pointer to the Child class as a pointer to the Parent:
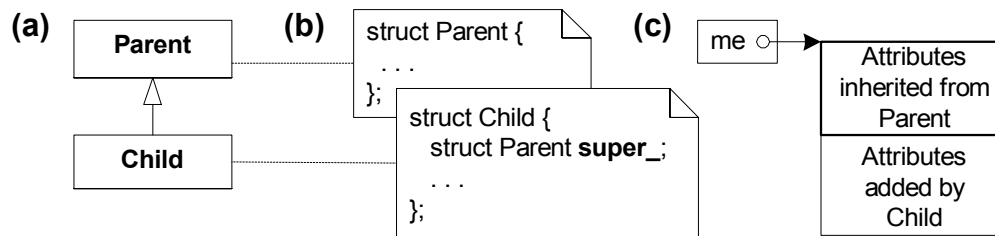
In particular, such memory alignment of the `Child` attributes with the `Parent` attributes allows you to always pass a pointer to the `Child` class to a C function that expects a pointer to the `Parent` class. (To be strictly correct in C, you should explicitly upcast this pointer.) Therefore, all methods designed for the `Parent` class are automatically available to `Child` classes; that is, they are **inherited**.

For example, in the code snippet from the previous section class `QActive` inherits from class `QHsm`. Please note the first protected attribute "`super_`" of type `QHsm` in the `QActive struct` definition.

Well, and that's all what it is to it. Quantum Platform code does not go with object-orientation beyond these two simple patterns. If you are interested in a more advanced OOP in C, including implementation of polymorphism, please refer to the recipe "C+ Object-Oriented Programming in C".