Module 2

Design and Optimization of Embedded and Real-time Systems

1 C++ for Embedded Systems

According to a survey by IEEE Spectrum (Top Programming Languages - IEEE Spectrum), the 10 most popular programming languages in 2021 are:

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

C and C++ have kept their positions in the top 5 after all these years. 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 this course, we only use a minimal set of C++ features 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 (with the memory pool being a chunk of static memory buffer).

1.1.1 Why Static?

First let’s look at when dynamic memory allocation could be useful. Here are two cases:

1. You don't know how big your data structure (vector, map, etc) is at compile-time, so you want it to be able to grow or sink as needed at run-time.
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.
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 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. Despite a higher overhead, shared_ptr’s are generally preferred to raw pointers (new/delete).

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

1.1.2 Memory Pool

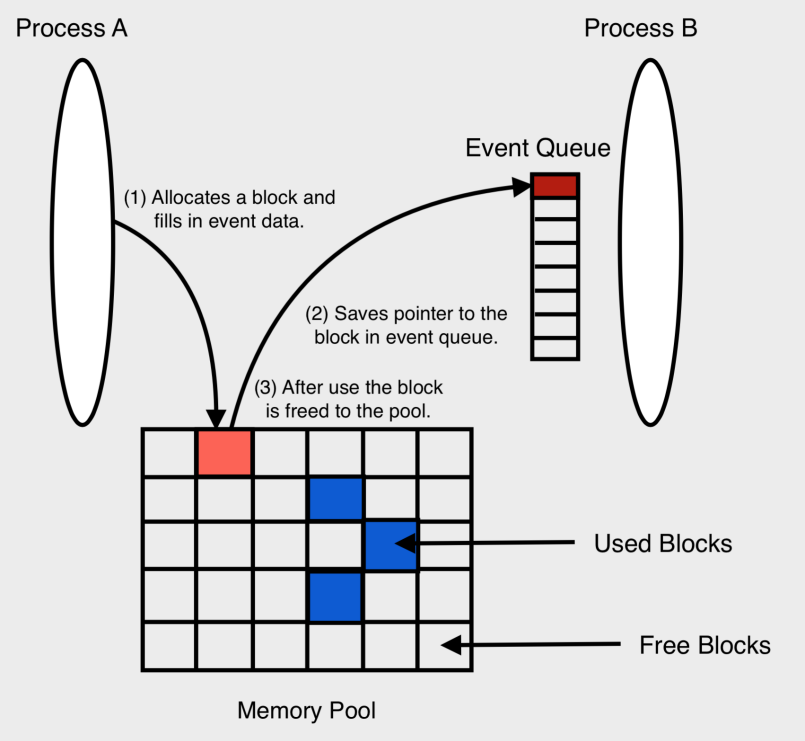
There are cases when dynamic memory is desirable. Examples include passing events between two objects via an event queue, and handling messages sent or received over the network. The main reason is that events/messages are dynamic in nature as they may arrive in any number at any time.

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.)



1.2 Standard Template Library (STL)

Another decision to make is whether to use STL or not. In the past when we worked with 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.

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.

In general STL could be used judicially. It is not used in this course because:

1. I have developed my own classes for common data types such as pair, map, string, array, pipe and fifo to illustrate basic OOP concepts and static memory management.
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 or allowed).

1.3 C++ Exception

C++ exception should be avoided in embedded systems. Exception used to impose a rather hefty run-time cost even when exception did not occur. Recent optimization may have made it less a 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 (e.g. using fault states) in which the fault handling 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 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. It may cause code bloat if not used carefully.

Since STL depends on template, we should be comfortable with it so we can use it when necessary. In my projects, template is used to implement some basic container types (e.g. a pipe).

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.

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.
3. New features add complexity to the syntax and rules of C++ if they were not complex to begin with.

   As it turns out, certain embedded applications may not benefit from those new features. For example, a system that relies on static memory allocation sees little use of *shared_ptr*. An asynchronous event-driven system won't need *promises* and *futures* added in C++1x.

2 Encapsulation

2.1 C Structure

Before OOP (Object-Oriented Programming), we use structure to group related data together. For example if we want to represent properties of a vehicle, we would have

```
enum {
    VIN_SIZE = 18,
};

typedef struct {
    char vin[VIN_SIZE];
    uint32_t seats;
    uint32_t mpg;
} Vehicle;
```

Functions that operate on a structure are separated from its 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:

```
float RunningCost(Vehicle const *v, uint32_t miles, float gasPrice) {
    return (miles / (v->mpg)) * gasPrice;
}
```

2.2 C++ Class

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 car or 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. As a start, let's get back to our *Vehicle* example.

Using class, the *Vehicle* class looks like this:

```
class Vehicle {
public:
    enum {
        VIN_SIZE = 18,
    };
    Vehicle(char const *vin, uint32_t seats, uint32_t mpg) :
        m_seats(seats), m_mpg(mpg) {
        STRBUF_COPY(m_vin, vin);
    }
    virtual ~Vehicle() {}
    char const *GetVin() const { return m_vin; }
    uint32_t GetSeats() const { return m_seats; }
    uint32_t GetMpg() const { return m_mpg; }
    virtual void GetInfo(char *buf, uint32_t bufSize) const {
        snprintf(buf, bufSize, "To be implemented in derived classes");
    }
    float RunningCost(uint32_t miles, float gasPrice) const {
        TEST_CODE_ASSERT(m_mpg);
        return (miles/m_mpg)*gasPrice;
    }
protected:
    char m_vin[VIN_SIZE];
    uint32_t m_seats;
    uint32_t m_mpg;
};
```

Usage:

```
Vehicle car1("CAR1", 5, 35);
Vehicle car2("CAR2", 7, 24);
console.Print("CAR1: A 100-mile trip costs $%f\n\r", car1.RunningCost(100, 2.89));
console.Print("CAR2: A 100-mile trip costs $%f\n\r", car2.RunningCost(100, 2.89));
```

Results:

```
CAR1:   A 100-mile trip costs $5.780000
CAR2:   A 100-mile trip costs $11.560000
```

Here are some key points:

1. A class is defined by the keyword *class*.

A class is like a structure that combines related *data members* together. In addition to that, a class can contain *methods* (*member functions*) that operate on its data members.

2. A class acts like a new *data type.* You can use it to build custom types (as complex as you can imagine) on top of those built-in C++ types (bool, int, float, std::string, etc.)

3. Once you have defined your class, you can *instantiate* (*create*) *objects* (*instances*) from it, in a similar way as you define an integer variable from the built-in type "int".

   Each object has its own copy of *data members* in memory (SRAM). In the example above, car1 and car2 have different values of number of seats (m_seats) and gas mileage (m_mpg).

   Note – *Member functions* exist in code memory (Flash) and there is only one copy shared by all instances of the class.

4. The *constructor* (a member function named as the class name without return type) is called when an object of the class is instantiated. The constructor is responsible for initializing data members of the object (e.g. with parameters passed in to the constructor).

   The constructor ensures an object is properly initialized when it is created. How? The compiler checks (at compile-time) if all the required parameters are present when an object is instantiated. The constructor can verify the parameters (e.g. range checking) at run-time in its function body. This is a major advantage over C struct with which you assign fields individually (easy to result in partially initialized structures).

5. The destructor (a member function named as the class name with a '~' in front) performs cleanup when an object is deleted (for dynamically allocated ones) or get out of scope (for statically allocated ones). Usually it frees up any resources allocated in the constructor (e.g. deallocating memory buffers or closing files, etc.)

6. Accessibility control with *public*, *protected* and *private.*

   "*public*" members are accessible anywhere in the program, just like members of a C struct.

   "*private*" members are only accessible by member functions of the *same class*.

   "*protected*" members behave like "private" with the exception that they are also accessible by methods of derived classes (see Inheritance section).

   It's a good practice to keep data members *non-public* (i.e. *private* or *protected),* which promotes *data hiding*. When needed, public methods (called getters/setters) are provided to *get* or *set* hidden data members. They provide a layer of isolation which makes it easier to modify (in a single place) when needed.

7. (Non-static) *member functions* must be called *on* an object of the same class using the dot operator (or using the arrow operator on a pointer to an object).

```
// RunningCost is called on the object car1 using dot operator.
Vehicle car1("CAR1", 5, 35);
float cost = car1.RunningCost(100, 2.89);

// RunningCost is called on the pointer pCar2 using arrow operator.
Vehicle *pCar2 = new Vehicle("CAR2", 7, 24);
cost = pCar2->RunningCost(100, 2.89);
delete pCar2;
```

The pointer to the object *on which* a member function is called is implicitly passed in to the member function as the first parameter named "*this*" (called the "*this pointer*"). It is through *this* pointer that a member function can access data members of the object or call other member functions on the object.

For example, this is the original Vehicle::RunningCost() method in the code:

```
// Note – Vehicle:: is the scope resolution operator, indicating that
//        RunningCost is a member function of Vehicle.
float Vehicle::RunningCost(uint32_t miles, float gasPrice) const {
    TEST_CODE_ASSERT(m_mpg);
    return (miles/m_mpg)*gasPrice;
}
```

Under the hood, the actual implementation is:

```
// Note – The first parameter "this" is implicitly added by the compiler.
float Vehicle::RunningCost(Vehicle this, uint32_t miles, float gasPrice) const {
    TEST_CODE_ASSERT(this->m_mpg);
    return (miles/this->m_mpg)*gasPrice;
}
```

In the first example above, "*this*" is equal to *&car1* (i.e. it is a pointer to the object *car1*).
In the second example, "*this*" is equal to *pCar2* (which is a pointer to the object itself).

3 Inheritance

3.1 Special Kinds of Vehicles

One way of building more complex objects from simpler ones is to use *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.

Inheritance represents an "*is-a*" relationship, meaning that a derived class *is a* special kind of the base class. Continuing with our vehicle example above, we can use the class Vehicle as a base class to represent any kinds or vehicles. From it we derived a special kind of vehicles named Sedan and another kind called Suv. Note that a sedan or an SUV is still a vehicle.

```
class Sedan : public Vehicle {
public:
    Sedan(char const *vin, uint32_t seats, uint32_t mpg, bool sporty) :
        Vehicle(vin, seats, mpg), m_sporty(sporty) {}
    bool IsSporty() const { return m_sporty; }
protected:
    bool m_sporty;
};

class Suv : public Vehicle {
public:
    Suv(char const *vin, uint32_t seats, uint32_t mpg, bool roofRack) :
        Vehicle(vin, seats, mpg), m_roofRack(roofRack) {}
    bool hasRoofRack() const { return m_roofRack; }
protected:
    bool m_roofRack;
};
```

Here are the key points:

1. The line "`class Sedan : public Vehicle`" indicates that the class Sedan is derived from the base class Vehicle.

   We can call *Sedan* a *subclass*, *derived class* or *child class*. We can call *Vehicle* a *superclass*, *base class* or *parent class*.

2. *Sedan* inherits all the data members and methods of *Vehicle*. In addition, it defines a new member named *m_sporty*. Here we assume being sporty or not is a special property only applicable to sedans but not to other kinds of vehicles.

   Similarly, *Suv* adds a property named *m_roofRack*. Again we assume having a roof rack or not is a special property only applicable to SUVs.

3. The constructor of a derived class is responsible for calling the constructor of its immediate base class and passing to it any required parameters.

   The constructor is also responsible for initializing data members of the derived class itself (similar to how the constructor of *Vehicle* initializes its own data members).

   It is important to note that the order of initialization begins from the very base class through any intermediate base classes before any data members of this derived class.

   The order of initiation of data members follows the order in which they are defined in the class body.

3.2 More Specialization

Now that we have derived *Sedan* from *Vehicle*, we can further create special kinds of sedans through a second level of inheritance, i.e. by deriving subclasses from *Sedan*. Similarly we can create special kinds of SUVs by deriving subclasses from *Suv*.

For example, we are a car dealership and we sell two models of vehicles, namely Model A which is a sedan and Model B which is an SUV. We can derive the class *ModelA* from *Sedan* and *ModelB* from *Suv*:

```
class ModelA : public Sedan {
public:
    ModelA(char const *vin) :
        Sedan(vin, 5, 35, false) {}
    void GetInfo(char *buf, uint32_t bufSize) const {
        snprintf(buf, bufSize, "%s: ModelA sedan, seats=%lu, mpg=%lu, sporty=%d",
                 m_vin, m_seats, m_mpg, m_sporty);
    }
};

class ModelB : public Suv {
public:
    enum {
        UPGRADE_SIZE = 128
    };
    ModelB(char const *vin, char const *upgrade = "") :
        Suv(vin, 7, 24, true) {
        STRBUF_COPY(m_upgrade, upgrade);
    }
    void GetInfo(char *buf, uint32_t bufSize) const {
        snprintf(buf, bufSize, "%s: ModelB SUV, seats=%lu, mpg=%lu, roofRack=%d,
                 upgrade='%s'", m_vin, m_seats, m_mpg, m_roofRack, m_upgrade);
    }
private:
    char m_upgrade[UPGRADE_SIZE];
};
```

Here are the key points:

1. Like before, the line "`class ModelA : public Sedan`" indicates that *ModelA* is derived from its base class *Sedan*. We call *Sedan* an *intermediate base class* since it in turn is derived from its own base class *Vehicle*.

2. The constructor of *ModelA* calls the constructor of its immediate base class *Sedan*, passing to it any required parameters. Note that some of the parameters to the constructor of *Sedan* are "fixed" (e.g number of seats and gas mileage) since those parameters are *known* properties of this special model (Model A) of sedans. However since each individual vehicle has its unique VIN (Vehicle ID), the parameter *vin* must be passed in by the caller instantiating the object.

   Here we don't need to worry about calling the constructors of any lower level base classes, since the immediate base class (*Sedan*) will take care of any lower level construction.

3. *ModelA* (likewise for *ModelB*) overrides the method *GetInfo()* that has already been defined in its base class *Vehicle*. Overriding happens when a method of a derived class has the same signature (function name, parameters and return type) as a method of a base class.

   When GetInfo() is called on a ModelA object, the version ModelA::GetInfo() is invoked. When

GetInfo() is called on a Vehicle object, the version Vehicle::GetInfo() is invoked. (For now, let's ignore the "*virtual*" keyword.) See this example:

Usage:

```
Vehicle car1("VEHICLE", 5, 35);
ModelA car2("MODEL_A");
char info[100];
car1.GetInfo(info, sizeof(info));
console.Print("Vehicle::GetInfo() is called: %s\n\r", info);
car2.GetInfo(info, sizeof(info));
console.Print("ModelA::GetInfo() is called: %s\n\r", info);
```

Results:

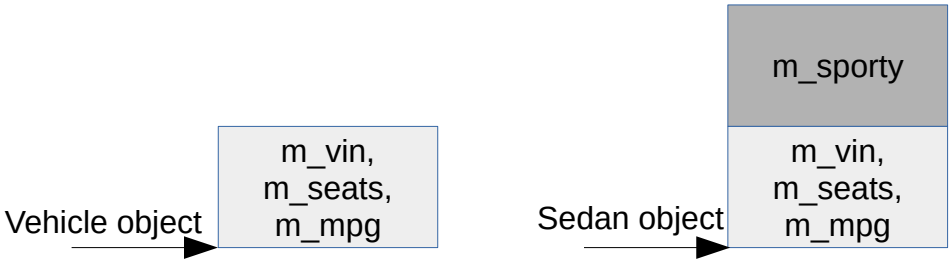
```
Vehicle::GetInfo() is called: To be implemented in derived classes

ModelA::GetInfo() is called: MODEL_A: ModelA sedan, seats=5, mpg=35, sporty=0
```

4. In *ModelB*, which is derived from *Suv*, we further specialize it with a new property named *m_upgrade*. Here we assume upgrade options are only available to Model B; otherwise we would have added this property to its base class.

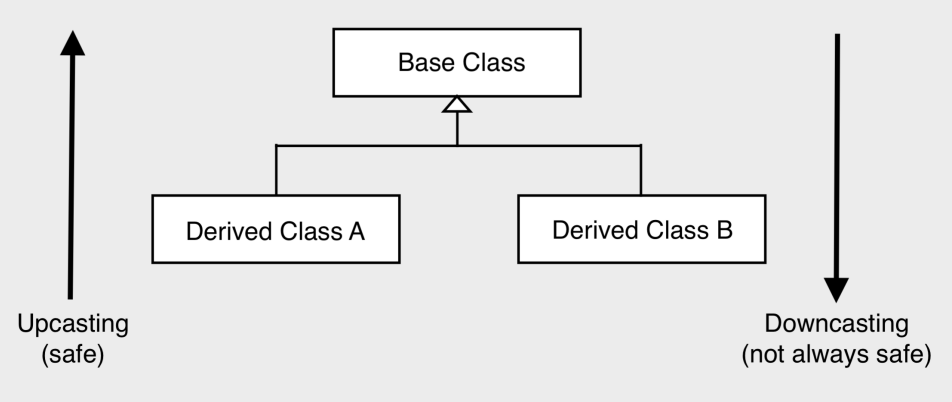
   This illustrates the concept of *programming by difference*. We factorize common properties and behaviors to base classes and implement only the differences in subclasses. In our vehicle example, we allow each model to customize its *info* string and add custom properties (e.g. upgrade options).

3.3 Casting

Since *Sedan* inherits from its base class *Vehicle*, it automatically gets the members *m_vin*, *m_seats* and *m_mpg*. On top of that it adds a new member *m_sporty*. A memory view of a Vehicle and a Sedan object looks like this:



As we can see, it is safe to convert/cast a pointer to a derived class object (*Sedan*) to a pointer to its base class object (*Vehicle*). This is called *upcasting* and the compiler automatically does it for us. It works because the memory layout of the bottom/base part of a derived object (*Sedan*) looks identical to that of its base class object (*Vehicle*).



It is important to note that the reverse, called *downcasting*, is not necessarily safe. That is, we cannot *always* convert/cast a pointer to a base class object to a pointer to a derived class object. We *must* be sure that the object pointed to by the base class pointer is indeed a derived class object.

The C++ feature called RTTI (Run-time Type Identification) helps but in embedded system RTTI is not encouraged due to its overhead. We can, however, encode the type information explicitly with a special data member (e.g. an enum) in the base class.

Since it is the programmers' responsibility to ensure type-correctness when performing downcasting, the compiler requires us to use a *static_cast<> operator* explicitly.

Example:

```
ModelA car("MODEL_A");
Vehicle *pv = &car;  // OK
console.Print("VIN = %s\n\r", pv->GetVin());

// Compile error:
// invalid conversion from 'APP::Vehicle*' to 'APP::ModelA*' [-fpermissive]
//ModelA *pa = pv;

ModelA *pa = static_cast<ModelA *>(pv);
console.Print("VIN = %s\n\r", pa->GetVin());
```

Results:

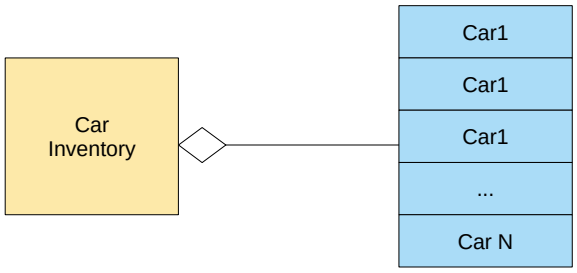
```
VIN = MODEL_A

VIN = MODEL_A
```

3.4 Composition

Apart from inheritance, *composition* is another way to build more complex objects from simpler ones, which is achieved by containing objects of other classes as its data members. Composition represents a *has-a* relationship.

Simple examples include a car contains (*has*) an engine, 4 wheels and 4 passenger doors. Continuing with our vehicle examples, we are going one level up to say an inventory of a car dealership contains a list of vehicles. We represent it with the class *CarInventory*:

```
class CarInventory {
public:
    enum {
        INFO_SIZE = 256,
    };
    CarInventory() = default;
    ~CarInventory() = default;
    void Add(Vehicle *v) {
        m_vehicles.push_back(v);
    }
    void Remove(char const *vin) {
        m_vehicles.erase(std::remove_if(m_vehicles.begin(), m_vehicles.end(),
            [&](Vehicle const *v) {
                  return STRING_EQUAL(v->GetVin(), vin);
            }), m_vehicles.end());
    }
    void Clear() {
        m_vehicles.clear();
    }
    void Show(Console &console, char const *msg = nullptr) {
        if (msg) {
            console.Print("%s\n\r", msg);
        }
        console.Print("Car inventory:\n\r");
        console.Print("==============\n\r");
        for (auto const &v: m_vehicles) {
            char info[INFO_SIZE];
            v->GetInfo(info, sizeof(info));
            console.Print("%s\n\r", info);
        }
    }
private:
    std::vector<Vehicle *> m_vehicles;
};
```



A vector of vehicles

Usage:

```
ModelA carA1("A001");
ModelA carA2("A002");
ModelA carA3("A003");
ModelB carB1("B001");
ModelB carB2("B002", "Autodrive");
ModelB carB3("B003", "Entertainment");
CarInventory inventory;
inventory.Add(&carA1);
inventory.Add(&carA2);
inventory.Add(&carA3);
inventory.Add(&carB1);
inventory.Add(&carB2);
inventory.Add(&carB3);
inventory.Show(console);
inventory.Remove("A002");
inventory.Remove("B002");
inventory.Remove("B004");
inventory.Show(console, "After deleting some cars...");
inventory.Clear();
inventory.Show(console, "After clearing all...");
```

Results:

```
Car inventory:
==============
A001: ModelA sedan, seats=5, mpg=35, sporty=0
A002: ModelA sedan, seats=5, mpg=35, sporty=0
A003: ModelA sedan, seats=5, mpg=35, sporty=0
B001: ModelB SUV, seats=7, mpg=24, roofRack=1, upgrade=''
B002: ModelB SUV, seats=7, mpg=24, roofRack=1, upgrade='Autodrive'
B003: ModelB SUV, seats=7, mpg=24, roofRack=1, upgrade='Entertainment'
After deleting some cars...
Car inventory:
==============
A001: ModelA sedan, seats=5, mpg=35, sporty=0
A003: ModelA sedan, seats=5, mpg=35, sporty=0
B001: ModelB SUV, seats=7, mpg=24, roofRack=1, upgrade=''
B003: ModelB SUV, seats=7, mpg=24, roofRack=1, upgrade='Entertainment'
After clearing all...
Car inventory:
==============
```

Here are the explanations:

1. The class *CarInventory* contains a list of vehicles using a C++ vector:

   ```
   std::vector<Vehicle *> m_vehicles;
   ```

   This is like a dynamic array that can grow or shrink as we add or remove items to/from it.

2. *CarInventory* only stores pointers to vehicle objects (*Vehicle* *) in the vector, rather than a copy of Vehicle objects themselves. Whenever we pass pointers around, we need to be extremely careful about ownership (i.e. which component owns the objects); otherwise we may run into dangling-pointer or memory-leak issues.

   In this example, we assume the vehicle objects referenced to by *CarInventory* are owned externally by its user, and therefore it does not delete them in its Remove() function or destructor.

   Note: we allocate the vehicle objects on the stack and they are deallocated automatically when they get out of scope (vs using *new* and *delete*). What would happen if *CarInventory* tries to free those pointers in its vector?

3. The Remove() function is what you would see in a modern C++ code with heavy uses of standard template libraries, lambda functions, etc. We will not use those features much in rest of this course.

4. When we create the vehicle objects, we create specific models, i.e. objects of the derived classes *ModelA* and *ModelB*. We pass in unique parameters to each call of the constructors so that each object has its own unique properties (attributes) such as its VIN, upgrade options, etc.

5. *CarInventory*, however, maintains pointers to those vehicle objects in a vector via *pointers to the base class* (i.e. *Vehicle* *) rather than pointers to specific derived classes (such as *ModelA* * or *ModelB* *). Why?

6. This is called *polymorphism* in object-oriented programming. It allows us to write generic code like *CarInventory* in our example. *CarInventory* does not care the exact derived classes of vehicle objects added to it, as long as they are derived from the same base class *Vehicle*. Recall that a compiler automatically converts a *pointer to a derived class object* to a *pointer to its base class object*. This is what happens in the following call:

   ```
   inventory.Add(&carA1);
   ```

7. In the *CarInventory::Show()* function, it loops through all the Vehicle objects referenced to by its vector and calls the *GetInfo()* method of each Vehicle object:

   ```
   // auto is deduced to “Vehicle *”
   for (auto const &v: m_vehicles) {
       char info[INFO_SIZE];
       v->GetInfo(info, sizeof(info));
       console.Print("%s\n\r", info);
   }
   ```

   Here it illustrates the purpose of the "*virtual*" keyword in the declaration of *GetInfo()* in the base class *Vehicle*. It defines *GetInfo()* as a *virtual function.*

8. 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/called on an object via a pointer to the base class, the code at run-time will call the overriding function (if any) defined in the *actual* derived class of the object. If a virtual function is *not* overridden by the derived class, the version in the base class will be invoked.

   As shown in the results output above, the invocation of *GetInfo()* via *Vehicle* * in the for-loop calls either *ModelA::GetInfo()* or *ModelB::GetInfo()* depending on the actual derived class of an object (e.g. carA1, carB1, etc) added to *CarInventory*.

9. If we now add a new model named *ModelC*, do we need to modify *CarInventory*? This is the essence of object-oriented programming.

   Can you think of other examples where polymorphism is useful?

4 Project Layout

Refer to Project Explorer in STM32CubeIDE. This is the main directory structure of our demo project *platform-stm32l475-0disco*:

1. **Inc** – 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) stm32l4xx_hal_conf.h – STM32Cube library configurations.
   (e) stm32l4xx_it.h – Cortex-M and STM32 interrupt-service routines (ISRs).
2. **Src** – Source files.
   (a) **app** – Active object folders, e.g. System, Disp, Sensor, Console, UartAct, etc. Each active object is equivalent to a thread. An active object is a state machine by itself, and may contain other parallel state machines (called regions).
   (b) **framework** – Application framework.
   (c) **qpcpp** – QP source code version 6.5.1 downloaded from https://www.state-machine.com.
   (d) **system** – Low-level libraries such as STM32Cube HAL drivers, component drivers, CMSIS, exception handlers.
   (e) bsp.cpp – Low-level board-support functions.

(f) main.cpp – Entry point of the application program.

(g) periph.cpp – Global STM32 peripheral configurations (GPIO, Timers, Clocks).

(h) stm32l4xx_it.cpp – Cortex-M and STM32 interrupt-service routines (ISRs.)

(i) syscalls.c and sysmem.c – Low-level system functions.

3. **Startup** – Startup assembly file (exception vector table, initialization code, etc).
4. ***.ld –** Linker scripts.