Modules and Object-Oriented Programming in C

There are many ways to modularize code. The C language supports modularization through specific language features, and through programming patterns and techniques. We describe a progression of such patterns along with the relevant language feature. The goal of this analysis is not only to better understand modularization techniques in C, but also to understand how modules and objects are implemented in general.

Consider for example a *container* module with the following abstract interface:

Initialize
: creates and/or initializes the container, possibly allocating resources

Add x
: adds element x to the container

Delete x
: removes element x from the container

Search x
: returns true when x is in the container

Shutdown
: destroys the container and clears all allocated resources.

This is a very basic interface for a set-type container. We will call this a *db* interface.

How can we implement this module?

Singleton Object

One way to implement the container is to implement a single *db* object with the following C interface declared in a header file called, say, `db.h`:

```c
/* db.h */
typedef int key_t;

int db_init();                          /* return Boolean: success/er
int db_clear();                         /* return Boolean: success/er
```

```
int db_add(key_t k);                    /* return Boolean: success/er
int db_delete(key_t k);                 /* return Boolean: success/er
int db_search(key_t id);                /* return Boolean: key found/
```

Now suppose you want to extend this module to make the set persistent. You could do that by simply offering more interface functions, perhaps in another header file `db_persistence.h`:

```
#include "db.h"

int db_save(const char * filename);   /* return Boolean: success/er
int db_load(const char * filename);   /* return Boolean: success/er
```

These are the header files that expose the application-programmer interface. Of course, the implementation file could have a lot more data structures and functions, which can be properly modularized in that implementation file.

From the perspective of the user of the set-container module, everything is done through those interface functions. For example:

```
#include "db.h"
#include "db_persistence.h"

int main() {
    db_init();
    db_add(100);
    db_add(200);
    db_add(300);

    db_save("numbers.db");
    db_clear();
}
```

This is all very straightforward. However, we can only have *one* set at a time. What if an application needs to keep two sets? We now discuss ways to allow an application to define and use multiple sets.

Object with Fixed Functions (Interface)

If the application needs to use more than one set, then we must implement a sort of

set *object*. The following code exemplifies the most direct way of doing that in C:

```
typedef int key_t;

struct db;

struct db * db_create();                    /* return non-null if
void db_destroy(struct db *);

int db_add(struct db * obj, key_t k);       /* return Boolean: su
int db_delete(struct db * obj, key_t k);    /* return Boolean: su
int db_search(struct db * obj, key_t id);   /* return Boolean: ke
```

A user application can therefore create sets using the constructor function `db_create`, and then use a set object by passing the pointer to that object to the interface functions. For example:

```
#include "db.h"
#include "db_persistence.h"

int main() {                    /* WARNING: example, no error checks
    struct db * s = db_create();
    db_add(s, 100);             /* in Java: s.add(100) */
    db_add(s, 200);
    db_add(s, 300);
    db_save(s, "numbers.db");
    db_destroy(s);
}
```

This design gives the user an *opaque* structure. This is good in terms of modularization, but it has a disadvantage: the user can only have objects with explicit storage, that is, objects allocated and deallocated explicitly (on the heap). In other words, the application does not see the definition of `struct db`, so the application can not ever declare an object of that type.

If the module designer wants to allow applications to use objects with an implicit or static storage (allocated on the stack), then the structure must be visible to the application. For example:

```
typedef int key_t;

struct db {
```

```
    unsigned int size;
    key_t * elements;
};

int db_init(struct db * obj);                   /* return Boolean: suc
void db_clear(struct db * obj);

int db_add(struct db * obj, key_t k);           /* return Boolean: suc
int db_delete(struct db * obj, key_t k);        /* return Boolean: suc
int db_search(struct db * obj, key_t id);       /* return Boolean: key
```

In this case, the application is responsible for the allocation of objects, but beyond that, the module implementation provides all the necessary functions to initialize, clear, and then use the object. For example:

```
#include "db.h"
#include "db_persistence.h"

int main() {                    /* WARNING: example, no error checks
    struct db s;
    db_init(&s);
    db_add(&s, 100);
    db_add(&s, 200);
    db_add(&s, 300);
    db_save(&s, "numbers.db");
    db_clear(&s);
}
```

Both these object schemes (or “styles”)—one with an opaque structure, one with a visible structure—are also quite easy to use and implement. They support multiple objects, and they support some extensions with additional functionalities as alluded to in the example with the `db_persistence.h` header.

However, notice that those extensions must be based on the same structure. Also, the extensions can provide additional functionalities but can not extend the same functionality, for example by providing a new specialized implementation of an existing functionality (e.g., a better search function).

In other words, both styles allow for multiple objects, but there is effectively a single *class* of objects. We now extend these two schemes further to support multiple classes with dynamic binding and inheritance.

Objects with an Explicit Switch Between Implementations

We start with a solution that is sometimes useful and reasonable, but that is also not so extensible and also possibly not very efficient.

For this and the following examples, consider the same interface with *add*, *delete*, and *search*, and imagine a number of implementations. For example, consider one implementation backed by a linked list, one by an array, and one by a hash table.

The public and generic interface would be the same as above, plus specialized constructors for each of the implementations:

```
typedef int key_t;

struct db;

struct db * db_list_create();
struct db * db_array_create();
struct db * db_hashtable_create();

void db_destroy(struct db *);

int db_add(struct db * obj, key_t k);
int db_delete(struct db * obj, key_t k);
int db_search(struct db * obj, key_t id);
```

However, the implementation would have to accommodate all three implementations. One way to do that would be to use a *union* of the three structures to store the object, plus an explicit switch mechanism. In essence, the structure would explicitly denote the implementation type, and then based on that type, the implementation would switch between the three specialized implementations.

For example, the object structure could be defined as follows:

```
enum db_class { DB_LIST, DB_ARRAY, DB_HASHTABLE };

/* possibly #included from, say, "db_array.h" */
struct db_list {
    struct db_list * prev;
    struct db_list * next;
    key_t value;
```

```
};

/* possibly #included from, say, "db_array.h" */
struct db_array {
    key_t * values;
    unsigned int size;
    unsigned int allocated_size;
};

/* possibly #included from, say, "db_hashtable.h" */
struct db_hashtable {
    key_t * table;
    unsigned int size;
    unsigned int table_size;
};

struct db {
    enum db_class type;
    union {
        struct db_list list;
        struct db_array array;
        struct db_hashtable hashtable;
    };
};
```

Then we implement each specialized constructor in the obvious way. For example:

```
struct db * db_list_create() {
    struct db * obj = malloc(sizeof(struct db));
    if (obj) {
        obj->type = DB_LIST;
        obj->list->prev = &(obj->list); /* sentinel */
        obj->list->next = &(obj->list); /* sentinel */
    }
    return obj;
};
```

Then each method would have an explicit switch based on the actual object type:

```
int db_list_add(struct db_list * obj, key_t k) {
    /* specialized implementation */
}
```

```
int db_array_add(struct db_array * obj, key_t k) {
    /* specialized implementation */
};

int db_hashtable_add(struct db_hashtable * obj, key_t k) {
    /* specialized implementation */
};

int db_add(struct db * obj, key_t k) {
    switch(obj->type) {
    case DB_LIST: return db_list_add(&(obj->list), k);
    case DB_ARRAY: return db_array_add(&(obj->array), k);
    case DB_HASHTABLE: return db_hashtable_add(&(obj->hashtable), k
    }
}
```

We can use this solution as follows:

```
int main() {
    struct db * x = db_create_list();
    struct db * y = db_create_array();

    db_add(x, 10);
    db_add(y, 20);
    db_search(x, 20);
    db_search(y, 20);

    db_destroy(x);
    db_destroy(y);
}
```

This solution can just as well be used with a visible structure and therefore with objects of static or automatic storage. This solution also allows for extensions, although the extensions require modification throughout, from the types identifiers (`enum db_class`) to the top-level object structure (`struct db`) and of the generic methods that must now dispatch the call to more specialized extensions. All of this might be okay for a specific application, but it is a major problem if we want to have a general modularity mechanism.

Another potential disadvantage is that the generic methods are simple switches that end up calling another method, which can be a useless inefficiency. Also, the space complexity of the structure is that of the largest implementation.

We now discuss ways to solve all these problems one by one.

Non-Union, Specialized Object Structures

We start by discussing alternative object structures. Consider again the main union structure:

```
struct db {
    enum db_class type;
    union {
        struct db_list list;
        struct db_array array;
        struct db_hashtable hashtable;
    };
};
```

Notice that this structure has a common component (`type`) followed by a specialized component (`list`, `array`, or `hashtable`). An alternative would be to define the three alternative structures, each one containing the common structure as their first element, like so:

```
enum db_class { DB_LIST, DB_ARRAY, DB_HASHTABLE };
struct db {                    /* common elements */
    enum db_class type;
};

struct db_list {
    struct db db;
    struct db_list * prev;
    struct db_list * next;
    key_t value;
};

struct db_array {
    struct db db;
    key_t * values;
    unsigned int size;
    unsigned int allocated_size;
};

struct db_hashtable {
```

```
    struct db db;
    key_t * table;
    unsigned int size;
    unsigned int table_size;
};
```

With these structures, we create each object–through the class-specific constructor–by allocating an object of the subclass (e.g., `db_list`), and then use a pointer to that object as a pointer to an object of class `db`. This is a common pattern in C, and it is a perfectly correct implementation, since the language guarantees that a pointer to an aggregate object (struct or array) is also a valid pointer to the first element of the aggregate. Therefore, we implement a specialized constructor as follows:

```
struct db * db_list_create() {
    struct db_list * obj = malloc(sizeof(struct db_list));
    if (obj) {
        obj->db.type = DB_LIST;
        obj->db.name = "Antonio";
        obj->prev = &(obj->list); /* sentinel */
        obj->next = &(obj->list); /* sentinel */
    }
    return &(obj->db);              /* == obj */
};
```

It is worth repeating and emphasizing the main technical underpinning of this solution. In C, an aggregate object can be used as the first element of the aggregate, and the same relation exists between pointers to the element and the aggregate object. In particular, *a pointer to an aggregate object (struct or array) is also a valid pointer to the first element of the aggregate object.*

For example, below we define a `struct date` that contains an `int day` object as its first element. This means that a pointer to a `struct date` is also a valid pointer to its `int day` element, and vice-versa, a pointer to the `int day` element (of a properly allocated `struct date` object) is a valid pointer to the whole structure.

```
struct date {
    int day;
    int month;
    int year;
};

int * new_day() {
```

```
    struct date * d = malloc(sizeof(struct date));
    d->day = 16;
    d->month = 12;
    d->year = 2019;
    return &(d->day);          /* == d */
}

int main() {
    int * p = new_day();
    *p = 17;
    struct date * d = (struct date *)p;
    printf("%d/%d/%d", d->day, d->month, d->year);
}
```

Going back to our polymorphic class implementation, the correspondence between the pointers allows us to call a generic "switch" API method by passing a pointer to the specialized object, which would be statically considered a pointer to a generic (superclass) object:

```
int db_list_add(struct db * obj_db, key_t k) {
    struct db_list * obj = (struct db_list *)obj_db;
    /* specialized implementation */
}

int db_array_add(struct db_array * obj_db, key_t k) {
    struct db_array * obj = (struct db_array *)obj_db;
    /* specialized implementation */
};

int db_hashtable_add(struct db * obj_db, key_t k) {
    struct db_hashtable * obj = (struct db_hashtable *)obj_db;
    /* specialized implementation */
};

int db_add(struct db * obj, key_t k) {
    switch(obj->type) {
    case DB_LIST: return db_list_add(obj, k);
    case DB_ARRAY: return db_array_add(obj, k);
    case DB_HASHTABLE: return db_hashtable_add(obj, k);
    }
}
```

This solution is still using an explicit switch between the various implementation, which is inefficient and cumbersome with further extensions. We now see how we can also solve this problem.

Objects with a Virtual Interface

A more general and elegant way to solve the problem of switching between implementations is to use function pointers. There are various ways to use function pointers. We now develop two such methods. We apply these methods to the specialized structures of the last example, although the same techniques can be used with a union structure.

Per-Object Virtual Table

Instead of switching based an an explicit type variable, one can simply use indirect calls through function pointers, and the most obvious way to do that is to store those pointers directly in the object. More specifically, since they are part of the generic API (superclass), we can put them in the superclass object:

```
struct db {
    void (*destroy)(struct db *);
    int (*add)(struct db * obj, key_t k);
    int (*delete)(struct db * obj, key_t k);
    int (*search)(struct db * obj, key_t id);
};

struct db_list {
    struct db db;
    struct db_list * prev;
    struct db_list * next;
    key_t value;
};

struct db_array {
    struct db db;
    key_t * values;
    unsigned int size;
    unsigned int allocated_size;
};

struct db_hashtable {
    struct db db;
```

```
    key_t * table;
    unsigned int size;
    unsigned int table_size;
};
```

Notice that the common structure (superclass) no longer needs to store a type attribute. The implementation of each specialized constructor is straightforward:

```
void db_list_destroy(struct db * obj_db) {
    struct db_list * obj = (struct db_list *)obj_db;
    /* specialized implementation */
}

int db_list_add(struct db * obj_db, key_t k) {
    struct db_list * obj = (struct db_list *)obj_db;
    /* specialized implementation */
}

int db_list_delete(struct db * obj_db, key_t k) {
    struct db_list * obj = (struct db_list *)obj_db;
    /* specialized implementation */
}

int db_list_search(struct db * obj_db, key_t k) {
    struct db_list * obj = (struct db_list *)obj_db;
    /* specialized implementation */
}

struct db * db_list_create() {
    struct db_list * obj = malloc(sizeof(struct db_list));
    if (obj) {
        obj->db.destroy = db_list_destroy;
        obj->db.add = db_list_add;
        obj->db.delete = db_list_delete;
        obj->db.search = db_list_search;

        obj->prev = obj; /* sentinel */
        obj->next = obj; /* sentinel */
    }
    return &(obj->db);
};
```

With this structure, each generic method simply invokes the corresponding function. For example, `db_add` would look like this:

```
int db_add(struct db * obj, key_t k) {
    return obj->add(obj, k);
};
```

In fact, it is clear that, with this structure, one might simply make the API publicly visible to the user, by including the definition of `struct db` in the public header `db.h`, and then let the user simply call the (indirect) method.

```
#include "db.h"

int main() {
    struct db * l = db_list_create();
    /* ... */
    l->add(l, k);
};
```

Notice that this method to implement specialized implementations of a common API can also be used for *singleton* objects as follows:

```
struct db {
    void (*destroy)();
    int (*db_add)(key_t k);
    int (*db_delete)(key_t k);
    int (*db_search)(key_t id);
};
```

The indirect calls for each specific implementation that depend on the dynamic type of an object (or singleton) are also called *virtual calls*. Therefore, the table of function pointers that implement those calls (per-object or singleton) is also called a *virtual table*.

Per-Class Virtual Table

Having a virtual table within each object is straightforward and efficient in terms of time, since it allows for an immediate call for each method. A call here is straightforward and immediate in the sense that the user can write an expression like `object->method()`, but notice that the call is still *indirect* through a function pointer. However, any non-trivial API is likely to have several virtual functions, which is inefficient in terms of space and also for the constructor. Notice in fact that all objects

of the same class (specialized subclass) will hold an identical virtual table, replicated and *stored within each object* and that must also be initialized for each object. This suggest another approach: we can have a *single, per-class table*, and each object can hold a pointer to that table. This can be done as follows:

```
struct db_vtable {
    void (*destroy)(struct db *);

    int (*db_add)(struct db * obj, key_t k);
    int (*db_delete)(struct db * obj, key_t k);
    int (*db_search)(struct db * obj, key_t id);
};

struct db {
    struct db_vtable * vtable;
};

struct db_list {
    struct db db;
    struct db_list * prev;
    struct db_list * next;
    key_t value;
};

struct db_array {
    struct db db;
    key_t * values;
    unsigned int size;
    unsigned int allocated_size;
};

struct db_hashtable {
    struct db db;
    key_t * table;
    unsigned int size;
    unsigned int table_size;
};
```

Notice that the only difference for now is that we have a pointer to a virtual table, instead of the virtual table itself, within each object.

The implementation of a specialized subclass would look like the following:

```
void db_list_destroy(struct db * obj_db) {
    struct db_list * obj = (struct db_list *)obj_db;
    /* specialized implementation */
}

int db_list_add(struct db * obj_db, key_t k) {
    struct db_list * obj = (struct db_list *)obj_db;
    /* specialized implementation */
}

int db_list_delete(struct db * obj_db, key_t k) {
    struct db_list * obj = (struct db_list *)obj_db;
    /* specialized implementation */
}

int db_list_search(struct db * obj_db, key_t k) {
    struct db_list * obj = (struct db_list *)obj_db;
    /* specialized implementation */
}

static struct db_vtable list_vtable = {
    .destroy = db_list_destroy,
    .add = db_list_add,
    .delete = db_list_delete,
    .search = db_list_search,
};

struct db * db_list_create() {
    struct db_list * obj = malloc(sizeof(struct db_list));
    if (obj) {
        obj->db.vtable = &list_vtable;

        obj->prev = obj; /* sentinel */
        obj->next = obj; /* sentinel */
    }
    return &(obj->db);
};
```

With this structure, each generic method invokes the corresponding function by accessing the corresponding function pointer through the pointer to the virtual table. For example, `db_add` would look like this:

```
int db_add(struct db * obj, key_t k) {
    return obj->vtable->add(obj, k);
};
```

Now that we have a virtual table and a pointer to that table, we can apply the same extension (subclass) mechanism *to the virtual table*. So, imagine we have a pair of methods defined in the `list` subclass, for example `begin` and `end`, which can be used to iterate through the elements in the set. These could be defined in the public API of as follows:

```
typedef struct db_list * db_list_iterator;

db_list_iterator db_list_begin(struct db * obj);
db_list_iterator db_list_end(struct db * obj);
```

We can also imagine that these two methods might be further extended in subclasses of `list`. So, we define a virtual table of the `list` class as an extension of the virtual table of the generic `db` superclass. We do that exactly as we did for the object attributes, that is, by defining a virtual-table structure for the subclass that *contains* as its first element the virtual table structure of the superclass:

```
/* possibly #included from "db_implementation.h"
*/
struct db_vtable {
    void (*destroy)(struct db *);

    int (*db_add)(struct db * obj, key_t k);
    int (*db_delete)(struct db * obj, key_t k);
    int (*db_search)(struct db * obj, key_t id);
};

struct db {
    struct db_vtable * vtable;
};

/* in this list implementation header (e.g., list_implementation.h)
*/
struct db_list_vtable {
    struct db_vtable super;

    db_list_iterator (*begin)(struct db * obj);
    db_list_iterator (*end)(struct db * obj);
```

```
};
```

This goes along with the following constructor:

```
db_list_iterator list_begin(struct db * obj) {
    struct db_list * obj = (struct db_list *)obj_db;
    /* specialized implementation */
}

db_list_iterator list_end(struct db * obj) {
    struct db_list * obj = (struct db_list *)obj_db;
    /* specialized implementation */
}

struct db_list_vtable list_vtable = {
    .super = {
        .destroy = list_destroy,
        .db_add = list_db_add,
        .db_delete = list_db_delete,
        .db_search = list_db_search,
    },
    .begin = list_begin,
    .end = list_end
};

struct db * db_list_create() {
    struct db_list * obj = malloc(sizeof(struct db_list));
    if (obj) {
        obj->db.vtable = &(list_vtable->super);
        obj->prev = obj; /* sentinel */
        obj->next = obj; /* sentinel */
    }
    return &(obj->db);
};
```

And the following virtual methods:

```
db_list_iterator db_list_begin(struct db * obj) {
    struct list_vtable * obj_vtable = (struct list_vtable *)obj->db
    return obj_vtable->begin(obj);
}
```

```
db_list_iterator db_list_end(struct db * obj) {
    struct list_vtable * obj_vtable = (struct list_vtable *)obj->db
    return obj_vtable->end(obj);
}
```

Exercise: DB

The architecture with per-class virtual tables is how object-oriented languages like Java and C++ implement objects. In Java, all methods are virtual. In C++ you have to explicitly declare a method as virtual, otherwise the default declaration and the resulting implementation is that of a static (non-virtual) invocation. You might consider the pros and cons of this design choice.

Implement five classes in C as defined in the following Java code:

```
class db {
    public abstract int db_add(int k);
    public abstract int db_delete(int k);
    public abstract int db_search(int id);
};

class named_db extends db {
    public String name;
};

class list_db extends named_db {
    /* private data members ... */
    public int db_add(int k) { /* ... */ }
    public int db_delete(int k) { /* ... */ }
    public int db_search(int id) { /* ... */ }
};

class array_db extends named_db {
    /* private and protected data members ... */
    public abstract int db_add(int k);
    public abstract int db_delete(int k);
    public abstract int db_search(int id);
};

class persistent_db extends list_db {
    /* private and protected data members ... */
```

```
    public int db_save(String filename) { /* ... */ }
    public int db_load(String filename) { /* ... */ }
};
```

Exercise: 2D Shapes

Design and implement in C a system of classes that implement a number of basic 2D shapes, including points, line segments, circles, rectangles, triangles.

The base abstract class, which should be called `shape`, should have methods `get_mbr` to get the minimum bounding rectangle; `get_area` to compute the total area of the shape; and `move(double dx, double dy)` that translates the shape by the given horizontal and vertical offsets `dx` and `dy`, respectively.