

# Patterns in C – Part 1

By Adam Petersen <adampetersen75@yahoo.se>

Over the last ten years, the pattern format has gained a tremendous popularity as *the* format used for capturing experience. One of the reasons for this popularity is the unique success of the classic book *Design Patterns* [1] by the Gang of Four. The *Design Patterns* [1] book definitively served the community by spreading the word about patterns.

Today, patterns in the software industry aren't limited to design; there exists a broad range of patterns, covering analysis patterns, patterns for organizations, patterns for testing, etc.

As most patterns are described in the context of an object oriented design, one is easily lead to believe that patterns require a language with support for object orientation. By browsing a popular online bookstore, I noticed a lot of language specific pattern literature: design patterns in Java, C#, Smalltalk and other popular object oriented languages. But, where is the one targeting the unique implementation constraints and techniques for the C language? Isn't it possible to use patterns in the development of C programs or doesn't it add any benefits?

An important thing to realize about patterns is that they are neither a blueprint of a design, nor are they tied to any particular implementation. By those means, shouldn't it be possible to find mechanisms fitting the paradigm of C, letting C programmers benefit from the experience captured by patterns?

## What you will experience in this series

It is my belief that C programmers can benefit from the growing catalogue of patterns. This series will focus on the following areas:

- *Implementation techniques.* I will present a number of patterns and demonstrate techniques for implementing them in the context of the C language. In case I'm aware of common variations in the implementation, they will be discussed as well. The implementations included should however not by any means be considered as a final specification. Depending on the problem at hand, the implementation trade-offs for every pattern has to be considered.
- *Problem solved.* Patterns solve problems. Without any common problem, the "pattern" may simply not qualify as a pattern. Therefore I will present the main problem solved by introducing the pattern and provide examples of problem domains where the pattern can be used.
- *Consequences on the design.* Every solution implies a set of trade-offs. Therefore each article will include the consequences on the quality of the design by applying the pattern.

## ...and what you won't

- *Object oriented feature emulation.* The pattern implementations will *not* be based on techniques for emulating object oriented features such as inheritance or C++ virtual functions. In my experience, these features are better left to a compiler; manually emulating such techniques are obfuscating at best and a source of hard to track down bugs at worst. Instead, it is my intent to present implementations that utilizes the strengths of the abstraction mechanisms already included in the C language.
- *In depth discussion of patterns.* As the focus in these articles will be on the implementation issues in C, the articles should be seen as a complement to the pattern descriptions. By those means, this series will *not* include exhaustive, in depth treatment of the patterns. Instead I will provide a high-level description of the pattern and reference existing work, where a detailed examination of the pattern is found.

## Pattern Categories

The patterns described in this series will span the following categories.

- *Architectural patterns.* Frank Buschmann defines such a pattern as "a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them" [2].
- *Design patterns.* A design pattern typically affects the subsystem or component level. Most patterns described in this series will be from this category, including patterns described in the classic *Design Patterns* [1] book.
- *Language level patterns.* This is the lowest level of the pattern-categories, also known as idioms. A language level pattern is, as its name suggests, mainly unique to one particular programming language. One simple, classic example is the strcpy version from Kernighan and Ritchie [3].

```
void strcpy(char *s, char *t)
{
    while(*s++ = *t++);
}
```

## The Foundation

Our journey through the patterns will start with a language level pattern that decouples interface from implementation, thus improving encapsulation and providing loose dependencies.

This pattern will lay the foundation for many of the sub-sequent parts of this series.

## FIRST-CLASS ADT Pattern

It's getting close to the project deadline as the project manager rushes into your office. "They found some problem with your code", he says with a stressed voice. "According to the test-team, you cannot register more than 42 orders for a given customer. Sounds strange, doesn't it?"

Darn. You knew it. Those hardcoded limits. "Oh, I'll have a look at it", you reply softly. "Fine, I expect the problem to be solved tomorrow", the manager mumbles as he leaves your office.

"No problem", you reply, well confident that the design of the customer routines are highly modular and clearly implemented (after all, you've implemented it yourself).

You launch your favorite code-editor and open a file with the following content:

### Listing 1 : Interface in `Customer.h`

```
/* Include guards and include files omitted. */

#define MAX_NO_OF_ORDERS 42

/* Internal representation of a customer. */
typedef struct
{
    const char* name;
    Address address;
    size_t noOfOrders;
    Order orders[MAX_NO_OF_ORDERS];
} Customer;

void initCustomer(Customer* theCustomer, const char* name,
                 const Address* address);

void placeOrder(Customer* customer, const Order* order);

/* A lot of other related functions... */
```

A quick glance reveals the problem. Simply increasing `MAX_NO_OF_ORDERS` would do, wouldn't it? But what's the correct value for it? Is it 64, 128, maybe even 2048 or some other magic number? Should customers with one, single order allocate space for, let's say, 2047 non-existing orders?

As you think of it, you realize that the current solution doesn't scale well enough. Clearly, you need another algorithm. You recall that a linked list exists in the company's code library. A linked list must do the trick. However, this means changing the internal structure of the `Customer`.

No problem, it looks like you thought of everything; the clients of the customer module simply use the provided functions for all access of the customer structure. Updating those functions should be enough, shouldn't it?

## Information hiding

Well, in an ideal world the change would be isolated to the one, single module. Given the interface above, clients depend upon the internal structure in at least one way.

At worst, the clients alter the internals of the data structure themselves leading to costly changes of all clients.

This can be prevented by frequent code-inspections and programmer discipline. In any case, we still have the compile-time dependencies and after changes, a re-compile of all clients is required and the compilation time may be significant in large systems.

The FIRST-CLASS ADT pattern will eliminate both dependency problems. The pattern provides us with a method of separating interface from implementation, thus achieving true information hiding.

## Definition of a FIRST-CLASS ADT

ADT stands for Abstract Data Type and it is basically a set of values and operations on these values. The ADT is considered first-class if we can have many, unique instances of it.

Sounds close to the interface listed in the introductory example above, doesn't it? However, the data type in the example is not abstract as it fails to hide its implementation details. In order to make it truly abstract, we have to utilize a powerful feature of C – the ability to specify incomplete types.

## Incomplete Types

The C standard [4] allows us to declare objects of incomplete types in a context where their sizes aren't needed. In our example implementation, we are interested in one property of incomplete types – the possibility to specify a pointer to an incomplete type (please note that the pointer itself is not of an incomplete type).

### Listing 2 : Pointer to an incomplete type

```
/* A pointer to an incomplete type (hides the implementation details). */
typedef struct Customer* CustomerPtr;
```

Instances of this pointer will serve as a handle for the clients of a FIRST-CLASS ADT. This mechanism enforces the constraint on clients to use the provided interface functions because there is no way a client can access a field in the `Customer` structure (the C language does not allow an incomplete type to be de-referenced).

The type is considered complete as soon as the compiler detects a subsequent specifier, with the same tag, and a declaration list containing the members.

### Listing 3 : Completing an incomplete type

```
/* The struct Customer is an incomplete type. */
typedef struct Customer* CustomerPtr;

/* Internal representation of a customer. */
struct Customer
{
    const char* name;
    Address address;
    size_t noOfOrders;
    Order orders[42];
};

/* At this point, struct Customer is considered complete. */
```

## Object Lifetime

Before we dive into the implementation of an ADT, we need to consider object creation and destruction.

As clients only get a handle to the object, the responsibility for creating it rests upon the ADT. The straightforward approach is to write one function that encapsulates the allocation of an object and initializes it. In a similar way, we define a function for destructing the object.

### Listing 4 : Interface to the ADT, Customer.h

```
/* Includes and include guards as before... */

/* A pointer to an incomplete type (hides the implementation details). */
typedef struct Customer* CustomerPtr;

/* Create a Customer and return a handle to it. */
CustomerPtr createCustomer(const char* name, const Address* address);

/* Destroy the given Customer. All handles to it will be invalidated. */
void destroyCustomer(CustomerPtr customer);
```

### Listing 5 : Implementation of the ADT in Customer.c

```
#include "Customer.h"
#include <stdlib.h>

struct Customer
{
    const char* name;
    Address address;
    size_t noOfOrders;
    Order orders[42];
};

CustomerPtr createCustomer(const char* name, const Address* address)
{
    CustomerPtr customer = malloc(sizeof * customer);

    if(customer)
    {
        /* Initialize each field in the customer... */
    }

    return customer;
}

void destroyCustomer(CustomerPtr customer)
{
    /* Perform clean-up of the customer internals, if necessary. */
    free(customer);
}
```

The example above uses `malloc` to obtain memory. In many embedded applications, this may not be an option. However, as we have encapsulated the memory allocation completely, we are free to choose another approach. In embedded programming, where the maximum number of needed resources is typically known, the simplest allocator then being an array.

## Listing 6 : Example of a static approach to memory allocation

```
#define MAX_NO_OF_CUSTOMERS 42
static struct Customer objectPool[MAX_NO_OF_CUSTOMERS];
static size_t numberOfObjects = 0;

CustomerPtr createCustomer(const char* name, const Address* adress)
{
    CustomerPtr customer = NULL;

    if(numberOfObjects < MAX_NO_OF_CUSTOMERS)
    {
        customer = &objectPool[numberOfObjects++];

        /* Initialize the object... */
    }

    return customer;
}
```

In case de-allocation is needed, an array will still do, but a more sophisticated method for keeping track of “allocated” objects will be needed. However, such an algorithm is outside the scope of this article.

## Copy Semantics

As clients only use a handle, which we have declared as a pointer, to the ADT, the issue of copy semantics boils down to pointer assignment. Whilst efficient, in terms of run-time performance, copies of a handle have to be managed properly; the handles are only valid as long as the real object exists.

In case we want to copy the real object, and thus create a new, unique instance of the ADT, we have to define an explicit copy operation.

## Dependencies managed

With the interface above, the C language guarantees us that the internals of the data structure are encapsulated in the implementation with no possibility for clients to access the internals of the data structure.

Using the FIRST-CLASS ADT, the compile-time dependencies on internals are removed as well; all changes of the implementation are limited to, well, the implementation, just as it should be. As long as no functions are added or removed from the interface, the clients do not even have to be re-compiled.

## Consequences

The main consequences of applying the FIRST-CLASS ADT pattern are:

1. *Improved encapsulation.* With the FIRST-CLASS ADT pattern we decouple interface and implementation, following the recommended principle of programming towards an interface, not an implementation.
2. *Loose coupling.* As illustrated above, all dependencies on the internals of the data structure are eliminated from client code.
3. *Controlled construction and destruction.* The FIRST-CLASS ADT pattern gives us full control over the construction and destruction of objects, providing us with the possibility to ensure that all objects are created in a valid state. Similarly, we can ensure proper de-allocation of all elements of the object, provided that client code behaves correctly and calls the defined destroy-function.
4. *Extra level of indirection.* There is a slight performance cost involved. Using the FIRST-CLASS ADT pattern implies one extra level of indirection on all operations on the data structure.
5. *Increased dynamic memory usage.* In problem domains where there may be potentially many instances of a quantity unknown at compile-time, a static allocation strategy cannot be used. As a consequence, the usage of dynamic memory tends to increase when applying the FIRST-CLASS ADT pattern.

## Examples of use

The most prominent example comes from the C language itself or, to be more precise, from the C Standard Library – **FILE**. True, **FILE** isn't allowed by the standard to be an incomplete type and it may be possible to identify its structure, buried deep down in the standard library. However, the principle is the same since the internals of **FILE** are implementation specific and programs depending upon them are inherently non-portable.

Sedgewick[5] uses FIRST-CLASS ADT to implement many fundamental data structures such as linked-lists and queues.

This pattern may prove useful for cross-platform development. For example, when developing applications for network communication, there are differences between Berkeley Sockets and the Winsock library. The FIRST-CLASS ADT pattern provides the tool for abstracting away those differences for clients. The trick is to provide two different implementations of the ADT, both sharing the same interface (i.e. include file).

## Next time

We will climb one step in the pattern categories and investigate a pattern from the *Design Patterns* [1] book. The next pattern may be useful for controlling the dynamic behavior of a program and in eliminating complex conditional logic.

## References

1. Gamma, E., Helm, R., Johnson, R., and Vlissides, J., “Design Patterns”, Addison-Wesley
2. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M., “POSA, A System of Patterns, Volume 1”, Wiley
3. Kernighan, B., and Ritchie, D., “The C Programming Language”, Prentice Hall
4. ISO/IEC 9899:1999, The C standard
5. Sedgewick, R., “Algorithms in C, Parts 1-4”, Addison-Wesley

## Acknowledgements

Many thanks to Drago Krznaric and Andre Saitzkoff for their feedback.