

C Patterns on Data Lifetime and Ownership

CHRISTOPHER PRESCHERN, B&R Industrial Automation GmbH

Design patterns are well known for object-oriented languages. However, when it comes to procedural programming languages, like for example C, then most of the knowledge documented in form of design patterns cannot be used, because these patterns focus on object-oriented languages and in C you do not have mechanisms like a constructor, a destructor, inheritance, or public and private variables in form of memory dedicated to an object. Despite of that, also C programmers group their functionality into object-like elements which have their own dedicated memory and in C the programmers then have to put special attention on the lifetime and ownership of these object-like elements. This paper shows in form of patterns how C programmers can do that. The pattern **STATELESS SOFTWARE-MODULE** describes, how related functions, which don't share any information between the function calls, can be put together. **SOFTWARE-MODULE WITH GLOBAL STATE** describes, how related functions can be put together as soon as they share information. **CALLER-OWNED INSTANCE** describes how multiple object-like elements can be created and distributed to different callers. **SHARED INSTANCE** describes how the same can be achieved with additionally making it possible to share object-like elements between callers. The four presented patterns are throughout the paper applied to a running code example, to give hands-on advice on how C programmers can apply these patterns.

How to structure a program is a central question for a programmer or a software architect. For object-oriented languages, this question is answered to a great extent by the Gang of Four design patterns [Gamma et al. 1994]. Design patterns provide a programmer with best practices on how objects should interact and on which object owns which other kinds of objects. Also, design patterns show, how such objects can be grouped together.

However, if we have a look at procedural programming languages like C, most of these design patterns cannot be implemented in the way described by the Gang of Four. There are no native object-oriented mechanisms in C. It is possible to emulate inheritance or polymorphism also in the C programming language like described in [Schreiner 2011] [Goedicke et al. 2000]; however, that might not be the first choice, because it makes things unfamiliar for programmers who are used to programming C and who are not used to program with object-oriented languages like C++ and concepts like inheritance and polymorphism. Such programmers might want to stick to their native C programming style which they are used to. When doing that, not all object-oriented design patterns guidance is usable or at least the specific implementation of the idea presented in a design patterns is not provided for non-object-oriented programming languages.

Here, this paper comes in. This paper fills this gap by providing patterns on how to structure your C program with object-like elements. For these objects-like elements, the patterns put special focus on who is responsible for creating and destroying them - in other words they put special focus on lifetime and ownership. That topic is especially important for C, because C has no destructor and no garbage collection mechanism and thus special attention has to be put on cleanup of resources.

But what is an “object-like element” and what is the meaning of it for the C programming language? The term object is well defined for object-oriented programming languages, however, for non-object-oriented programming languages it is not clear what the term object means. The following provides an explanation of the equivalent for this term which will be used throughout this paper:

In object-based programming, an object describes a related set of data which has an identity and properties and which is used to store representations of things found in the real world. In object-oriented programming, an object additionally has the capability of polymorphism and inheritance. The objects-like elements described throughout this paper do not address object-oriented principles. They only address object-based principles and are closer to the following definition of the term object in the C programming language:

“An object is a named region of storage.” [Kernighan and Ritchie 1988]

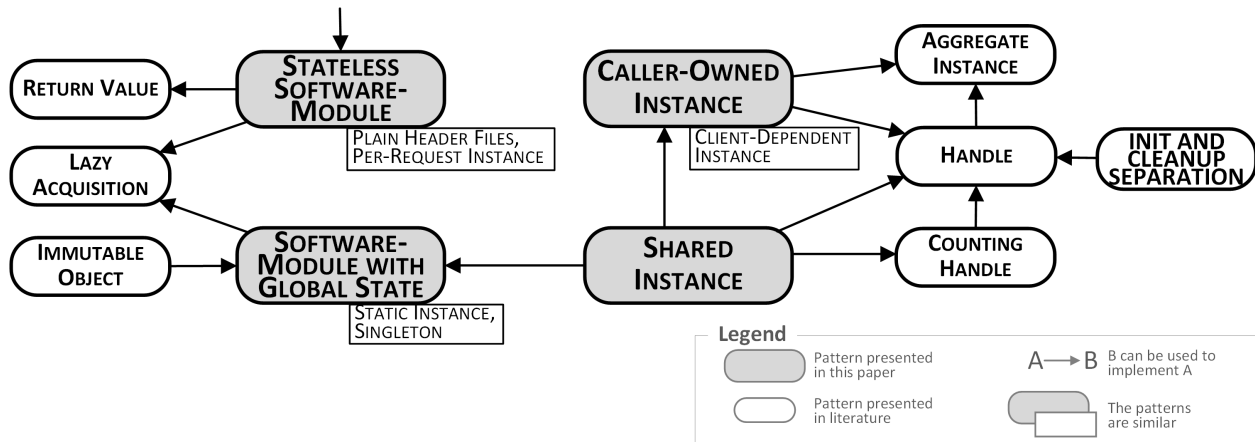
In the following, the paper will not use the term object anymore, because that term is usually associated with object-oriented programming languages. Instead, in this paper such an object-like element simply is an instance of a data structure and will furthermore be called **instance**.

Such instances do not stand by themselves, but instead they usually come with related pieces of code which makes it possible to operate on the instances. This code is usually put together into a set of H-files for its interface and a set of C-files for its implementation. Throughout this paper, the sum of all this related code, which often defines, similar to object-oriented classes, the operations which can be performed on an instance, will be called **software-module**.

When programming C, the described instances of data are usually implemented as abstract data types (e.g. having an instance of a struct with functions accessing the struct members). An example for such an instance is the C stdlib FILE struct which stores information like the file pointer or the position in the file. The corresponding software-module is the `stdio.h` API and its implementation of functions like `fopen` and `fclose` which provide access to the FILE instances.

The patterns in this paper cover the topic of how to provide access to instances and on who has ownership of these instances. A very similar topic is covered by a subset of the patterns from the book “Remoting Patterns” [Voelter et al. 2007]. The book “Remoting Patterns” presents patterns for building distributed object middleware and three of these patterns focus on lifetime and ownership of objects created by remote servers. Compared to that, the patterns presented in this paper focus on a different context. They are not patterns for remote systems, but for local procedural programs. They focus on C programming, but can also be used for other procedural programming languages. Still, some of the underlying ideas in the patterns are very similar to the “Remoting Patterns”.

The following figure and table show the patterns presented in this paper as well as related patterns from literature. The white-colored patterns are patterns from literature. The gray-colored patterns are presented in this paper and to make the patterns easier to grasp, their application is in the following shown with a running example.



PATTERN NAME	PATTERN SOLUTION SKETCH
--------------	-------------------------

STATELESS SOFTWARE-MODULE	Put all related functions together into one API. The functions don't share state information, so don't use global variables or parameters to transport information between the functions
SOFTWARE-MODULE WITH GLOBAL STATE	Put all related functions together into one API and put all function implementations in one single C file. Use global variables to communicate state information between the functions.
CALLER-OWNED INSTANCE	Put all related functions together into one API and pass an instance (a struct) between the functions to share information. Each caller gets his own instance.
SHARED INSTANCE	Put all related functions together into one API and pass an instance (a struct) between the functions to share information. Callers can share instances.
STATIC INSTANCE	A server application provides a number of previously-known remote object instances. Provide static instances independent of the client's state and lifecycle [Voelter et al. 2007].
PER-REQUEST INSTANCE	A server application serves remote objects that are accessed by a large number of clients. Therefore, let the distributed object middleware activate a new servant for each invocation. This servant handles the request, returns the result, and is then deactivated. [Voelter et al. 2007]
CLIENT-DEPENDENT INSTANCE	In situations in which the application logic of the remote object extends the logic of the client, it becomes important to think about where to put the common state of both. Therefore, provide remote objects whose lifetime is controlled by the clients. [Voelter et al. 2007]
SINGLETON	Creates and provides access to an object of a unique thing in your system. [Gamma et al. 1994]
PLAIN HEADER FILES	Provide function declarations for any functionality you want to provide to your caller in your headerfile. Hide any internal functions, internal data, and your internal implementation in your implementation file and don't provide this implementation file to the caller [Preschern 2016].
HANDLE	Create an abstract parameter (=handle) in your API and pass this parameter to all your functions. Your functions know how to interpret (e.g. cast) this parameter. The functions then share their state information or data via this handle [Coplien 1998] [Preschern 2016].
INIT AND CLEANUP SEPARATION	Put initialization and cleanup into separate functions. Pass a struct to these functions to keep track of allocated resources [Preschern 2015].
RETURN VALUE	Use the return value of a C function to transport a copy of the result [Preschern 2018].
LAZY ACQUISITION	Implicitly initialize the object or data the first time it is used [Kirchner and Jain 2004].
IMMUTABLE OBJECT	Create an object in the static memory. Fill data into this object at compile-time or at boot-time. Return a pointer to this object to callers who want to access this data and make sure that the callers cannot modify the data [Henney 2000] [Preschern 2018].
AGGREGATE INSTANCE	Define a context object which contains all the related information which you want to share. Define this context object in the interface of the software-module which provides this information and let the user directly access information from its members [Zdun 2005] [Preschern 2018].
COUNTING HANDLE	Have a handle object through which the user works on the shared object. The handle encapsulates the responsibility for tracking references to the shared object and for its deletion [Henney 2001].

The following running example shows the application of the presented patterns throughout the paper:

Running example:

You want to implement a device driver for your Ethernet network interface card. The Ethernet network interface card is installed on the operating system your software runs on, so you can use the POSIX socket functions to send and receive network data.

You want to build some abstraction for your user, because you want to provide an easier way to send and receive data compared to socket functions and because you want to add some additional features to your Ethernet driver. Thus you want to implement something that encapsulates all the socket details.

To achieve that, start with a simple STATELESS SOFTWARE-MODULE.

STATELESS SOFTWARE-MODULE



Context:

You want to provide functions with related functionality to a caller. The functions don't operate on common data shared between the functions and they don't require preparation of resources, like memory which has to be initialized previous to the function call.

Problem:

You want structure your code with related functionality and you want to make that functionality for the caller as easy as possible to use.

You want to make it as simple as possible for the caller to access your functionality, the caller should not have to deal with initialization and cleanup aspects of the functions, and the caller should not be confronted with implementation details.

You don't require the functions to be very flexible regarding future changes while maintaining backwards compatibility - instead the functions should only provide a simple to use abstraction for accessing the implemented functionality.

Solution:

Put all related functions into one headerfile and provide this headerfile to the caller as the interface to your software-module. The functions provided in the headerfile are atomic and independent from one another.

No communication and no sharing of state information takes place between the functions and also no storing of state information takes place between function calls. That means the functions calculate a result or perform an action which does not depend on other function calls in the API and does not depend on previous function calls. The only communication that takes place is between the caller and the one called function at a time (e.g. in form

of RETURN VALUES). Still, the functions are related and because of that they are put together into one API. Being related means that the function are usually applied together by a caller (see interface seggregation principle [Martin 2018]) and that if they change, they change for the same reason (see common closure principle [Martin 2018]).

Put the function declaration in one headerfile (as suggested in PLAIN HEADER FILES), but put the implementations of the functions into separate C-files (but maybe into the same directory). As the functions are related, because they logically belong together, but they do not share a common state or influence each others state, there is no need to share information between the functions via global variables or to encapsulate this information by passing instances between the functions. That's why even each single function implementation can be put into a separate C-file.

If a function requires any resources, such as heap memory for example, then the resources have to be handled transparently for the caller. They have to be acquired, implicitly initialized before they are used as suggested by LAZY ACQUISITION, and released within the function call.

Code Example

CALLER

```
int ret = function(42);
```

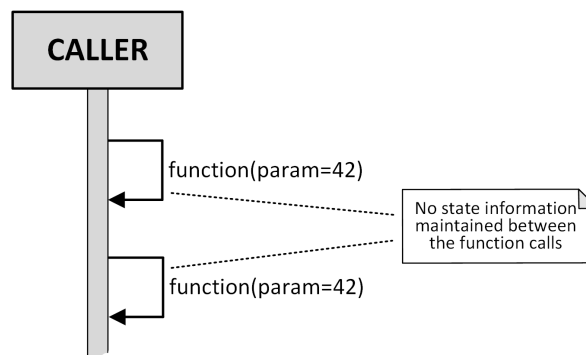
API

```
int function(int param);
```

IMPLEMENTATION

```
int function(int param)
{
    int result;
    /* calculate result only depending on 'param'
       and not requiring any state information */
    return result;
}
```

The caller calls `function` and retrieves a copy of the result. When calling the function twice with same input parameters, the function would deliver the exact same result as no state information is maintained in the STATELESS SOFTWARE-MODULE and as in this special case, also no other function which holds state information is called.



Consequences:

You have a very simple interface and the caller does not have to cope with initializing or cleaning up anything for your software-module. The caller can simply call one of the functions independently of previous function calls and independently of other parts of the program, like for example other threads, which concurrently access the software-module. Having no state information makes it much easier to understand what a function does.

The caller does not have to cope with questions about ownership, because there is nothing to own - the functions have no state. The resources required by the function are allocated and cleaned up within the function call and are thus transparent to the caller.

However, not all functionality can be provided with such a simple interface. If the functions within an API share state information or data (e.g. one has to allocate resources required by another), then a different approach, like a SOFTWARE-MODULE WITH GLOBAL STATE or a CALLER-OWNED INSTANCE has to be taken in order to share this information.

Known Uses and References:

These types of related functions gathered into one API are found each time that the function within the API do not require shared information or state information.

- The `sin` and `cos` functions from `math.h` are provided in the same headerfile and they calculate their results solely based on the function input. They do not maintain state information and each call with the same input produces the same output.
- The `string.h` functions `strcpy` or `strcat` do not depend on one another. They don't share information, but they belong together and are thus part of a single API.
- The Windows headerfile `VersionHelpers.h` provides information about which Microsoft Windows version is currently running. Functions like `IsWindows7OrGreater` or `IsWindowsServer` provide related information, but still the functions share no information and are independent from one another.
- The Linux headerfile `parser.h` comes with functions like `match_int` or `match_hex`. These functions try to parse an integer or a hexadecimal value from a substring. The functions are independent from one another, but still they belong together into the same API.
- The source code of the Nethack game also has many applications of this pattern. For example, the `vision.h` headerfile includes functions to calculate, whether the player is able to see specific other items on the game map. The functions `couldsee(x, y)` and `cansee(x, y)` calculate whether the player has clear line of sight to the other item and whether the player additionally faces that other item. Both functions are independent from one another and don't share state information.
- A variant of this pattern with more focus on API flexibility is presented as the PLAIN HEADER FILES pattern in [Preschern 2016].
- The pattern PER-REQUEST INSTANCE [Voelter et al. 2007] describes that a server in a distributed object middleware should activate a new servant for each invocation and that it should, after the servant handles the request, return the result and deactivate the servant. Such a call to a server maintains no state information and is similar to calls in STATELESS SOFTWARE-MODULES, but with the difference that STATELESS SOFTWARE-MODULES don't cope with remote entities.

Running Example:

Your first device driver looks like this:

API

```
void sendByte(char data,
              char* destination_ip);
char receiveByte();
```

IMPLEMENTATION

```
void sendByte(char data,
              char* destination_ip)
{
    /* open socket to destination_ip,
       send data via this socket
       and close the socket */
}

char receiveByte()
{
    /* open socket for receiving data,
       wait some time and
       return received data */
}
```

The user of your Ethernet driver does not have to cope with implementation details like how to access sockets and can simply use the provided API. Both of the functions in this API can be called at any time independently from one another and the caller can obtain data provided by the functions without having to cope with ownership and with freeing resources. Using this API is very simple, but also very limited.

Next, you want to provide additional functionality for your driver. You want to make it possible for the user to see whether the Ethernet communication works fine and thus, you want to provide statistics showing the number of sent or received packets. However, with a simple STATELESS SOFTWARE-MODULE, you cannot achieve that, because you have no retained memory for storing state information from one function call to another.

To achieve that, you need a SOFTWARE-MODULE WITH GLOBAL STATE.

SOFTWARE-MODULE WITH GLOBAL STATE

**Context:**

You want to provide functions with related functionality to a caller. The functions do operate on common data shared between them and they might require preparation of resources, like memory which has to be initialized previous to using your functionality. However, the functions do not require any caller-dependent state information.

Problem:

You want structure your code with related and state-sharing functionality and you want to make that functionality for the caller as easy as possible to use.

You want to make it as simple as possible for the caller to access your functionality, the caller should not have to deal with initialization and cleanup aspects of the functions, and the caller should not be confronted with implementation details. The caller should not necessarily realize that the functions access common data.

You don't require the functions to be very flexible regarding future changes while maintaining backwards compatibility - instead the functions should only provide a simple to use abstraction for accessing the implemented functionality.

Solution:

Put all related functions into one headerfile and provide this headerfile to the caller as the interface to your software-module. In the implementation of your software-module, have one global instance and let your functions share common resources via this instance.

The caller does not necessarily realize that the functions operate on common resources. The initialization and lifetime of the resources is transparently managed in the software-module and is independent from the lifetime of its callers.

Put the function declaration in one headerfile (as suggested in PLAIN HEADER FILES) and put all the implementations for your software-module into one single C-file. In this C-file, have a global instance (a file-global static struct or several file-global static variables), which holds the common shared resources that should be available for your function implementations. Your function implementations can then access these shared resources similar to private variables in object-oriented programming languages.

Within your software-module, the access to these file-global resources might have to be protected by synchronization primitives such as a Mutex to make it possible to have multiple callers from different threads. Make this synchronization within your function implementation, so that the caller does not have to deal with synchronization aspects.

If the resources have to be initialized, then you can initialize them at startup time or you can use LAZY ACQUISITION to initialize the resources right before they are needed. In any case, the initialization is done transparently for the caller.

Code Example

CALLER

```
int ret = function(42);
```

API

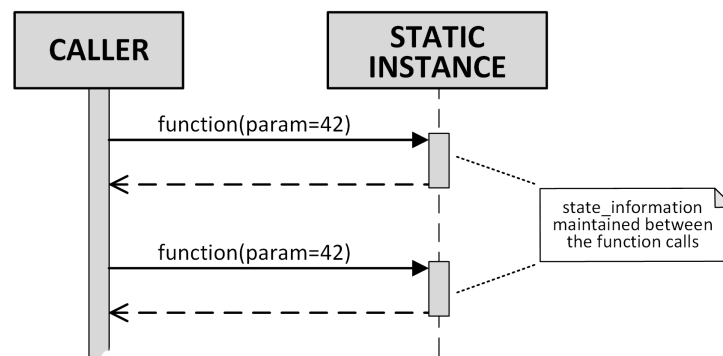
```
int function(int param);
```

IMPLEMENTATION

```
static int state_information;

int function(int param)
{
    int result;
    /* calculation of 'result' depending on
     * 'state_information' which might have been
     * built up with previous function calls */
    return result;
}
```

The caller calls `function` and retrieves a copy of the result. When calling the function twice with same input parameters, the function might deliver different results, because the function maintains state information.



Consequences:

Even though your functions share information or resources, they are available to the caller without requiring the caller to pass parameters containing this shared information and the caller is not responsible for allocating and cleaning up resources.

To achieve this sharing of information in your software-module, you implemented the C-version of a SINGLETON. Beware of the SINGLETON - literature documents many disadvantages of this pattern and often it is rather called an anti-pattern. Still, in C such SOFTWARE-MODULES WITH GLOBAL STATE are wide-spread, because it is quite easy to write the keyword `static` before a variable and as soon as you do that, you have your SINGLETON. In some cases that is ok. If your C-files are short, having file-global variables is quite similar to having private variables in object-oriented programming and if your functions do not require state information or do not operate in multi-threaded environments,

then you might be just fine. However if multi-threading and state information becomes an issue and if your implementation file becomes longer and longer, then you are in trouble and the SOFTWARE-MODULE WITH GLOBAL STATE is not a good solution anymore.

If your SOFTWARE-MODULE WITH GLOBAL STATE requires initialization, then you either have to initialize it at system startup, or you have to use LAZY ACQUISITION to initialize short before the use of resources, which has the drawback, that the duration for your function calls vary, because at the first call additional initialization code is implicitly called. In any case, the resource acquisition is performed transparently to the caller. The resources are owned by your software-module and thus the caller is not burdened with ownership of resources - he does not have to explicitly acquire or release the resources.

However, not all functionality can be provided with such a simple interface. If the functions within an API share caller-specific state information, then a different approach, like a CALLER-OWNED INSTANCE has to be taken.

Known Uses and References:

- The `string.h` function `strtok` splits a string into tokens. Each time the function is called, the next token for the string is delivered. In order to have the state information about which token to deliver next, the function uses static variables.
- With a Trusted Platform Module (TPM) one can accumulate hash values of loaded software. The corresponding function in the TPM-Emulator v0.7 code uses static variables to store this accumulated hash value.
- The `math` library uses a state for its random number generation. Each call of `rand` calculates a new pseudo random number based on the number calculated with the previous `rand` call. Initially `srand` has to be called in order to set the seed (the initial static information) for the pseudo random number generator called with `rand`.
- An IMMUTABLE OBJECT [Henney 2000] can be seen as part of a SOFTWARE-MODULE WITH GLOBAL STATE with the special case, that the IMMUTABLE OBJECT is not modified at runtime.
- The source code of the Nethack game stores information about items (swords, shields) in a static list defined at compile-time and provides functions to access this shared information.
- STATIC INSTANCE [Voelter et al. 2007] suggests to provide remote objects with lifetime decoupled from the lifetime of the caller. The remote objects can, for example, be initialized at startup time and then be provided to a caller when requested. SOFTWARE-MODULE WITH GLOBAL STATE presents the same idea of having static data, however, it is not meant for having multiple instances for different callers.

Running Example:

Now you have the following code for your Ethernet driver:

API

```
void sendByte(char data,
              char* destination_ip);

char receiveByte();

int getNumberOfSentBytes();

int getNumberOfReceivedBytes();
```

IMPLEMENTATION

```
static int sent_packets;
static int received_packets;

void sendByte(char data,
              char* destination_ip)
{
    sent_packets++;
    /* socket stuff */
}

char receiveByte()
{
    received_packets++;
    /* socket stuff */
}

int getNumberOfSentBytes()
{
    return sent_packets;
}

int getNumberOfReceivedBytes()
{
    return received_packets;
}
```

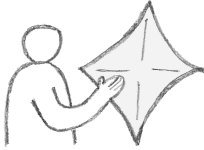
The API looks very similar an API of a STATELESS SOFTWARE-MODULE; however, behind this API now lies functionality to retain information between the function calls and the counters for sent and received bytes need that. As long as there is just one user (one thread) who uses this API, everything is just fine. However, if there are multiple threads, then with static variables you always run into the problem, that race conditions occur if you don't look after that - e.g. by implementing mutual exclusion for the access to the static variables.

Alright - now you want the Ethernet driver to be more efficient - you want to send more data. You could simply call your `sendByte` function very often to do that, however, in your Ethernet driver implementation that means that for each `sendByte` call, you establish a socket connection, send the data and close the socket connection again. Establishing and closing the socket connection takes most of the communication time.

That is quite inefficient and you'd rather want to open your socket connection once, then send all the data by calling your `sendByte` function several times, and then close the socket connection. But now your `sendByte` function requires some preparation and some teardown phase and this state cannot be stored in a SOFTWARE-MODULE WITH GLOBAL STATE, because as soon as you have more than one caller (one thread), you'd run into the problem that multiple callers want to simultaneously send data - maybe even to different destinations.

To achieve that, provide each of these callers with a CALLER-OWNED INSTANCE.

CALLER-OWNED INSTANCE

**Context:**

You want to provide functions with related functionality to a caller. The functions do operate on common data shared between the functions, they might require preparation of resources, like memory which has to be initialized previous to using your functionality, and they share caller-specific state information amongst each other.

Problem:

You want to provide multiple callers access to functionality with functions which depend on one another. The interaction of the caller with your functions builds up state information which has to be stored.

Maybe one function has to be called before another, because it influences some state stored in your software-module which is then needed by the other function. That can be achieved with a SOFTWARE-MODULE WITH GLOBAL STATE; however, that only works as long as there is just one single caller. In a multi-threaded environment with multiple callers, you cannot have one central software-module holding all caller-dependent state information.

Still, you want to hide implementation details from the caller and you want to make it as simple as possible for the caller to access your functionality. It has to be clearly defined whether the caller is responsible for allocating and cleaning up resources.

Solution:

For functions which share common resources or other kind of caller-dependent state information, construct a separate CALLER-OWNED INSTANCE for each caller and let the callers pass this instance, which holds the resource or state information, along to the functions. The functions operate on or based on these instances. Provide explicit functions to create and destroy these instances, so that the callers can determine their lifetime.

To implement such an instance which can be accessed from multiple functions, pass a struct pointer along with all functions that require sharing resources or state information. The functions can now use the struct members similar to private variables in object-oriented languages to store and read resource and state information.

The struct can be declared in the API to let the caller conveniently directly access its members. Alternatively, the struct can be declared in the implementation and only a pointer to the struct can be declared in the API (as suggested by HANDLE). The caller does not know the struct members (they are like private variables) and can only operate with functions on the struct.

As the instance has to be manipulated by multiple of your functions and as you do not know when the caller finished calling all functions he wants to call, the lifetime of the instance has to be determined by the caller. Therefore, provide explicit functions for creating and destroying it.

Code Example**CALLER**

```

struct INSTANCE* inst;
inst = createInstance();
operateOnInstance(inst);
/* access inst->x
   or inst->y */
destroyInstance(inst);

```

API

```

struct INSTANCE
{
    int x;
    int y;
};

struct INSTANCE* createInstance();
void operateOnInstance(struct INSTANCE* inst);
void destroyInstance(struct INSTANCE* inst);

```

IMPLEMENTATION

```

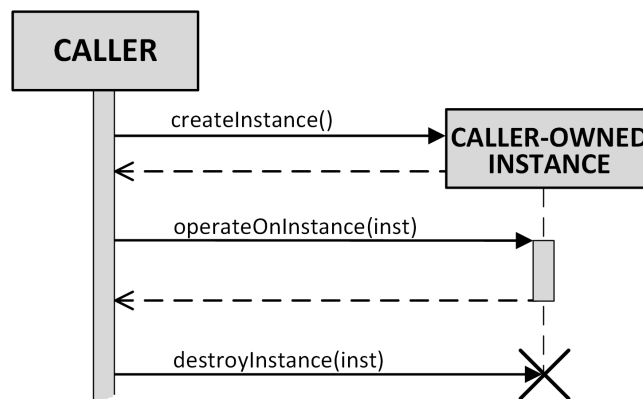
struct INSTANCE* createInstance()
{
    struct INSTANCE* inst;
    inst = malloc(sizeof(struct INSTANCE));
    return inst;
}

void operateOnInstance(struct INSTANCE* inst)
{
    /* work with inst->x and inst->y */
}

void destroyInstance(struct INSTANCE* inst)
{
    free(inst);
}

```

The function `operateOnInstance` works on resources created with the previous function call `createInstance`. The resource or state information between the two function calls is transported by the caller who has to provide the `INSTANCE` for each function call and who also has to clean up the resources (`destroyInstance`).



Consequences:

The functions in your API are more powerful now, because they can share state information and they can operate on shared data while still being available for multiple callers (multiple threads). Each created CALLER-OWNED INSTANCE has its own private variables and even if more than one such CALLER-OWNED INSTANCE is created - e.g. by multiple callers in a multi-threaded environment - it is no problem.

However, to achieve that, your API becomes more complicated. You have to make explicit `create()` and `destroy()` calls for managing the instance's lifetime, because C does not support constructors and destructors. This makes handling with instances much more difficult, because the caller obtains ownership and is responsible for correctly explicitly cleaning up the instance. As this has to be done manually with the `destroy()` call and not via a destructor like in object-oriented programming languages, this is a common pitfall for memory leaks. This issue is addressed by INIT AND CLEANUP SEPARATION which suggests that also the caller should have a dedicated cleanup function to make this task more explicit.

Also, compared to a STATELESS SOFTWARE-MODULE, calling each of the functions becomes a bit more complicated. Each function takes an additional parameter referencing the instance and the functions cannot be called in arbitrary order - the caller has to know which one has to be called first (however, that is made explicit through the function signatures).

Known Uses and References:

- An example for the use of a CALLER-OWNED INSTANCE is the doubly linked list provided with the `glibc` library. The caller creates a list with `g_list_alloc` and can then insert items into this list with `g_list_insert`. When finished working with the list, the caller is responsible to clean it up with `g_list_free`.
- This pattern is described as a way to build modular programs in C [Robert Strandh 1994]. The book states the importance of identifying abstract data types in the application, which can be manipulated or accessed with functions.
- The Windows API to create menus in the menu bar has a function to create a menu instance (`CreateMenu`), it has functions to operate on menus (like `InsertMenuItem`), and a function to destroy the menu instance (`DestroyMenu`). All these functions have one parameter to pass the `HANDLE` to the menu instance.
- Apache's software-module to handle HTTP requests provides a function to create request information (`ap_sub_req_lookup_uri`), to process it (`ap_run_sub_req`), and to destroy it (`ap_destroy_sub_req`). These functions take a struct pointer to the request instance in order to share request information.
- The source code of the Nethack game uses a struct instance to represent monsters and provides functions to create and destroy a monster. Also, the Nethack code provides functions to obtain information from monsters (`is_starting_pet`, `is_vampshifter`).
- CLIENT-DEPENDENT INSTANCE [Voelter et al. 2007] suggests for distributed object middlewares to provide remote objects whose lifetime is controlled by the clients. The server creates new instances for clients and the client can then work with these instances, pass this instance along, or destroy them.

Running Example:

Now you have the following code for your Ethernet driver:

API

```
typedef struct Sender* SENDER;
SENDER createSender(char*
                    destination_ip);
void sendByte(SENDER s, char data);
void destroySender(SENDER s);

char receiveByte();

int getNumberOfSentBytes();

int getNumberOfReceivedBytes();
```

IMPLEMENTATION

```
struct Sender
{
    char destination_ip[16];
    int socket;
}

SENDER createSender(char* destination_ip)
{
    SENDER s = malloc(sizeof(struct Sender));
    /* create socket to destination_ip
       and store it in SENDER s */
    return s;
}

void sendByte(SENDER s, char data)
{
    sent_packets++;
    /* send data via socket
       stored in SENDER s */
}

void destroySender(SENDER s)
{
    /* close socket stored in SENDER s */
    free(s);
}
```

A caller can first create a sender, then send all the data he wants, and then destroy the sender. Thus, the caller can make sure that the socket connection does not have to be established again for each `sendByte()` call. The caller has ownership of the created sender. He has full control over how long the sender lives and he is responsible for cleaning it up:

```
SENDER s = createSender("192.168.0.1");
char* dataToSend = "Hello World!";
char* pointer = dataToSend;
while(*pointer != '\0')
{
    sendByte(s, *pointer);
    pointer++;
}
destroySender(s);
```

Next, let's assume that you are not the only user of this API. There might be multiple threads using your API. As long as one thread creates a sender for sending to IP address X and another thread creates a sender for sending to Y, we are just fine and the Ethernet driver creates independent sockets for both threads.

However, let's say the two threads want to send data to the same recipient. Now the Ethernet driver is in trouble, because on one specific port, it can only open one socket per destination IP. A solution to that problem would be to not allow two different threads to send to the same destination. The second thread creating the sender could simply receive an error. However, it is also possible to allow both threads sending data using the same sender.

To achieve that, simply construct a SHARED INSTANCE.

SHARED INSTANCE



Context:

You want to provide functions with related functionality to a caller. The functions do operate on shared common data, they might require preparation of resources, like memory which has to be initialized previous to using your functionality. There are multiple contexts in which the functionality can be called and these contexts are shared between the callers.

Problem:

You want to provide multiple callers access to functionality with functions which depend on one another. The interaction of the caller with your functions builds up state information which has to be stored. Some of your callers want to share their state information.

Storing the state information in a SOFTWARE-MODULE WITH GLOBAL STATE is not an option, because there are multiple callers who want to build up different state information. Storing the state information per caller in a CALLER-OWNED INSTANCE is not an option, because either some of your callers want to access and operate on one and the same instance, or because you don't want to create new instances for every caller in order to keep resource costs low.

Still, you want to hide implementation details from the caller and you want to make it as simple as possible for the caller to access your functionality. It has to be clearly defined whether the caller is responsible for allocating and cleaning up resources.

Solution:

For functions which share common resources or other kind of state information, provide a SHARED INSTANCE to the caller and let the caller pass this instance along to the functions. The functions operate on or based on these instances. Keep ownership of the created instance in the software-module which creates it and don't transfer ownership to the callers. Instead, provide the callers with a reference to the instance and one and the same reference can also be provided to multiple callers, if required.

Just like with the CALLER-OWNED INSTANCE, provide a struct pointer or a HANDLE which the caller then has to pass along the function calls. However, when creating the instance, the caller now additionally has to provide an identifier (e.g. unique name) to specify the kind of instance which he wants. With this identifier you can know, whether such an instance already exists. In case it exists, you don't create a new instance, but instead return the struct pointer or HANDLE to the instance which you already created and returned to other callers before. In any case, it makes no difference to the caller whether the retrieved instance is new or was already created before by a different caller.

To know whether an instance already exists, you have to hold a list of already created instances in your software-module. That can be done by implementing a SOFTWARE-MODULE WITH GLOBAL STATE to hold the list. Additionally to whether an instance was already created or not, you can store the information of who currently accesses which instances or at least how many callers currently access an instance. This additional information is required, because when everybody is finished accessing it, it is your duty to clean the instance up, because you are the one who has ownership of it.

You also have to check, whether your functions can be called simultaneously by different callers on one and the same instance. In some easier cases, there might be no things whose access has to be mutually excluded by different callers, because information is only read and in such cases an IMMUTABLE OBJECT, which does not allow the caller changing the instance, could be implemented. However, in other cases, in your functions you have to implement mutual exclusion for resources shared through the instance.

Code Example

CALLER1

```
struct INSTANCE* inst;
inst = openInstance(42);
/* operate on the same
   instance as CALLER2 */
operateOnInstance(inst);
closeInstance(inst);
```

CALLER2

```
struct INSTANCE* inst;
inst = openInstance(42);
/* operate on the same
   instance as CALLER1 */
operateOnInstance(inst);
closeInstance(inst);
```

API

```
struct INSTANCE
{
    int x;
    int y;
};

struct INSTANCE* openInstance(int id);
void operateOnInstance(struct INSTANCE* inst);
void closeInstance(struct INSTANCE* inst);
```

IMPLEMENTATION

```
struct INSTANCELIST
{
    struct INSTANCE* inst;
    int count;
};

static struct INSTANCELIST list[MAX_INSTANCES];

struct INSTANCE* openInstance(int id)
{
    if(list[id].count == 0)
    {
        list[id].inst =
            malloc(sizeof(struct INSTANCE));
    }
    list[id].count++;
    return list[id].inst;
}

void operateOnInstance(struct INSTANCE* inst)
{
    /* work with inst->x and inst->y */
}
```

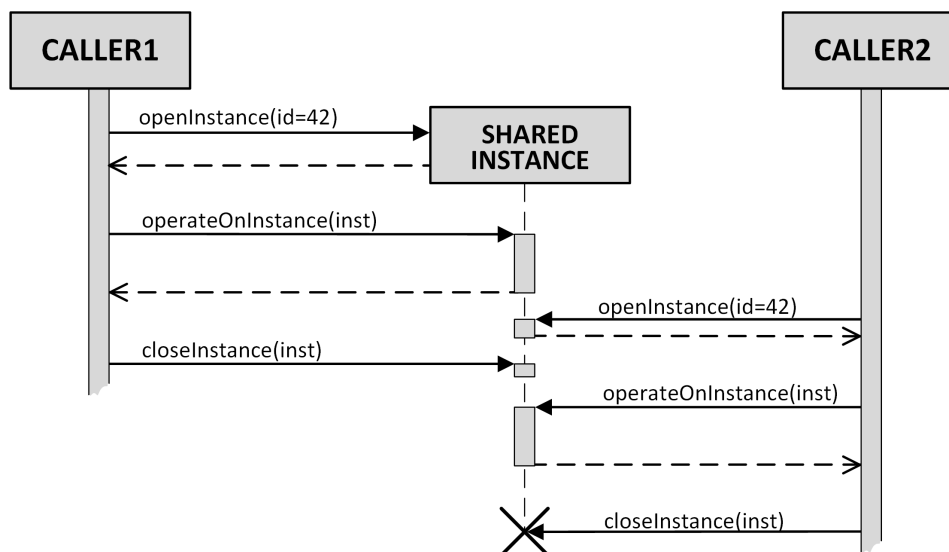
```

static int getInstanceId(struct INSTANCE* inst)
{
    int i;
    for(i=0; i<MAX_INSTANCES; i++)
    {
        if(inst == list[i].inst)
        {
            break;
        }
    }
    return i;
}

void closeInstance(struct INSTANCE* inst)
{
    int id = getInstanceId(inst);
    list[id].count--;
    if(list[id].count == 0)
    {
        free(inst);
    }
}

```

The caller retrieves an `INSTANCE` by calling `openInstance`. The `INSTANCE` might be created by this function call, or it might already have been created by a previous function call and might also be used by another caller. The caller can then pass the `INSTANCE` along to the `operateOnInstance` function calls, to provide this function with the required resource or state information from the `INSTANCE`. When finished, the caller has to call `closeInstance`, so that the resources can be cleaned up, if no other caller operates on the `INSTANCE` anymore.



Consequences:

Multiple callers now have simultaneous access to one single instance. This quite often implies that you have to cope with mutual exclusion within your implementation in order to not burden the user with such issues. This implies that the duration for a function call varies, because the caller never knows whether another caller currently uses the same resources and whether another caller blocks them.

Your software-module and not the caller has ownership of the instance and your software-module is responsible for cleaning up resources. Still the caller is responsible for releasing the resources so that your software-module knows when to clean everything up - like for the CALLER-OWNED INSTANCE, this a pitfall for memory leaks.

As the software-module has ownership of the instances, the software-module can also clean up the instances without requiring the callers to initiate that. For example, if the software-module receives a shutdown signal from the operating system, it has the possibility to clean up all instances, because it has ownership of all the instances.

Known Uses and References:

- An example for the use of a SHARED INSTANCE are the `stdio.h` file-functions. A file can be opened by multiple callers via the function `fopen`. The caller retrieves a handle to the file and can the read from or write to the file (`fread`, `fprintf`). The file is a shared resource - e.g. there is one global cursor position in the file for all callers. When a caller finished operating on the file, it has to be closed with `fclose`.
- This pattern and its implementation details for object-oriented programming languages is presented as COUNTING HANDLE by [Henney 2001]. The COUNTING HANDLE describes how a shared object on the heap can be accessed and how its lifetime can transparently be handled.
- The Windows registry can be accessed simultaneously by multiple threads with the function `RegCreateKey` (which opens the key, if it already exists). The function delivers a HANDLE which can be used by other functions to operate on the registry key. When the registry operations are finished the `RegCloseKey` function has to be called by everybody who opened the key.
- The Windows functionality to access Mutex objects (`CreateMutex`) can be used to access a shared resource (the Mutex) from multiple threads. With the Mutex, inter-process synchronization can be implemented. When finished working with the Mutex, each caller has to close it by using the function `CloseHandle`.
- The B&R Automation Runtime operating system allows multiple callers to access device drivers simultaneously. A caller uses the function `DmDeviceOpen` to select one of the available devices. The device driver framework checks whether the selected driver is available and then provides a HANDLE to the caller, also if other callers currently already operate on this driver. The callers can then simultaneously interact with the driver (send or read data, interact via IO-controls,) and after this interaction they tell the device driver framework that they are finished by calling `DmDeviceClose`.

Running Example:

The driver now implements the following functions:

API

```
typedef struct Sender* SENDER;
SENDER openSender(char*
                destination_ip);
void sendByte(SENDER s, char data);
void closeSender(SENDER s);

char receiveByte();

int getNumberOfSentBytes();
int getNumberOfReceivedBytes();
```

IMPLEMENTATION

```
SENDER openSender(char* destination_ip)
{
    SENDER s;
    if(isInSenderList(destination_ip))
    {
        s = getSenderFromList(destination_ip);
    }
    else
    {
        s = createSender(destination_ip);
    }
    increaseNumberOfCallers(s);
    return s;
}

void sendByte(SENDER s, char data)
{
    lock(); /* mutual exclusion for threads */
    sent_packets++;
    unlock();
    /* send data via socket
       stored in SENDER s */
}

void closeSender(SENDER s)
{
    decreaseNumberOfCallers(s);
    if(numberOfCallers(s) == 0)
    {
        /* close socket stored in SENDER s */
        free(s);
    }
}
```





Running Example:

The API of the running example did not change a lot - instead of having create/destroy-functions, your driver now provides open/close-functions. By calling such a function, the caller retrieves the HANDLE for the sender and indicates the driver, that he is now operating a sender; however, the driver does not necessarily create this sender at that point in time. That might already have happened by an earlier call to the driver (maybe performed by a different thread). Also a close-call might not actually destroy the sender. The ownership of this sender remains in the driver implementation which can decide when to destroy the senders (e.g. when all callers close the sender, or if some termination signal is received).

The fact that you now have a SHARED INSTANCE instead of a CALLER-OWNED INSTANCE is mostly transparent to the caller. However, the driver implementation changed - it has to remember whether a specific sender was already created and has to provide this shared instance instead of creating a new one. When opening a sender, the caller does not know, whether this sender will be newly created or whether he retrieves an existing sender - depending on that, the duration of the function call might vary.

The presented running driver example showed different kinds of ownership and data lifetime in one single example. We saw, how a simple Ethernet driver evolved by adding functionality. First, a STATELESS SOFTWARE-MODULE was sufficient, because the driver did not require any state information. Next, such state information was required and it was realized by having a SOFTWARE-MODULE WITH GLOBAL STATE in the driver. Then, the need for more performant send-functions and for multiple callers for these send-functions came up and was first implemented by the CALLER-OWNED INSTANCE and in a next step by the SHARED INSTANCE.

The following table shows an overview of the consequences of applying the patterns presented throughout this paper:

	STATELESS SOFTWARE-MODULE 	SOFTWARE-MODULE WITH GLOBAL STATE 	CALLER-OWNED INSTANCE 	SHARED INSTANCE 
Resource sharing between functions	Not possible	Single set of resources	Set of resources per instance (= per caller)	Set of resources per instance (shared by multiple callers)
Resource ownership	Nothing to own	The software-module owns the static data	The caller owns the instance	The software-module owns instances and provides references
Resource lifetime	No resources live longer than a function call	Static data lives forever in the software-module	Instances live until callers destroy them	Instances live until the software-module destroys them
Resource initialization	Nothing to initialize	At compile time or at startup	By the caller when creating an instance	By the software-module when the first caller opens an instance

CONCLUSION

This paper showed different ways on how to structure your C programs and on how long different instances in your program live. With this paper, a C programmer has some basic guidance about the options he has when he has to organize his programs into software-modules and about the options regarding ownership and lifetime he has when constructing instances.

Especially the running example shows in which cases which kind of ownership and lifetime is a good choice. The presented patterns throughout this running example provide some basic programming vocabulary and with this vocabulary, for future work, larger patterns using these basic programming vocabulary will be documented.

This paper is part of a series of papers on C programming [Preschern 2015][Preschern 2016][Preschern 2017] [Preschern 2018]. This series of papers is the start of a collection of hands-on best practices for the C programming language and with these patterns as part of your programming vocabulary, you can more easily construct complex software and you have some guidance on how to design your C programs - similar to the guidance you have for object-oriented programming languages with the Gang of Four design patterns.

ACKNOWLEDGMENTS

I want to thank my shepherd Uwe Zdun, in particular for pointing me towards relevant literature, for asking the right questions in order to make me get the terminology right, and for helping me to go into more detail in my patterns. I also want to thank my work colleague Thomas Havlovec for providing me with feedback on the paper.

REFERENCES

- James Coplien. 1998. C++ Idioms. In *Proceedings of the 3rd European Conference on Pattern Languages of Programming*.
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- Michael Goedicke, Gustaf Neumann, and Uwe Zdun. 2000. Object System Layer. In *Proceedings of 5th European Conference on Pattern Languages of Programs (EuroPLoP)*.
- Kevlin Henney. 2000. Patterns in Java: Patterns of Value. In <http://www.curbralan.com>.
- Kevlin Henney. 2001. C++ Patterns: Reference Accounting. In *Proceedings of the 6th European Conference on Pattern Languages of Programming (EuroPLoP)*.
- Brian W. Kernighan and Dennis M. Ritchie. 1988. *C Programming Language, 2nd Edition*. Prentice Hall.
- Michael Kirchner and Prashant Jain. 2004. *Pattern-Oriented Software Architecture: Volume 3: Patterns for Resource Management*. Wiley.
- Robert C. Martin. 2018. *Clean Architecture*. Prentice Hall.
- Christopher Preschern. 2015. Idioms for Error Handling in C. In *Proceedings of the 20th European Conference on Pattern Languages of Programming (EuroPLoP)*.
- Christopher Preschern. 2016. API Patterns in C. In *Proceedings of the 21st European Conference on Pattern Languages of Programming (EuroPLoP)*.
- Christopher Preschern. 2017. Patterns for C Iterator Interfaces. In *Proceedings of the 22nd European Conference on Pattern Languages of Programming (EuroPLoP)*.
- Christopher Preschern. 2018. Patterns for Returning Information from C Functions. In *Proceedings of the 23rd European Conference on Pattern Languages of Programming (EuroPLoP)*.
- Robert Strandh. 1994. Modular C. <http://metamodulaire.com/Computing/modular-c.pdf>. (1994).
- Axel-Tobias Schreiner. 2011. *Object Oriented Programming with ANSI C*. Lulu.
- Markus Voelter, Michael Kirchner, and Uwe Zdun. 2007. *Remoting Patterns*. Wiley.
- Uwe Zdun. 2005. Patterns of Argument Passing. In *Proceedings of the 4th Nordic Conference of Pattern Language of Programs (VikingPLoP)*.