Patterns for Returning Data from C Functions

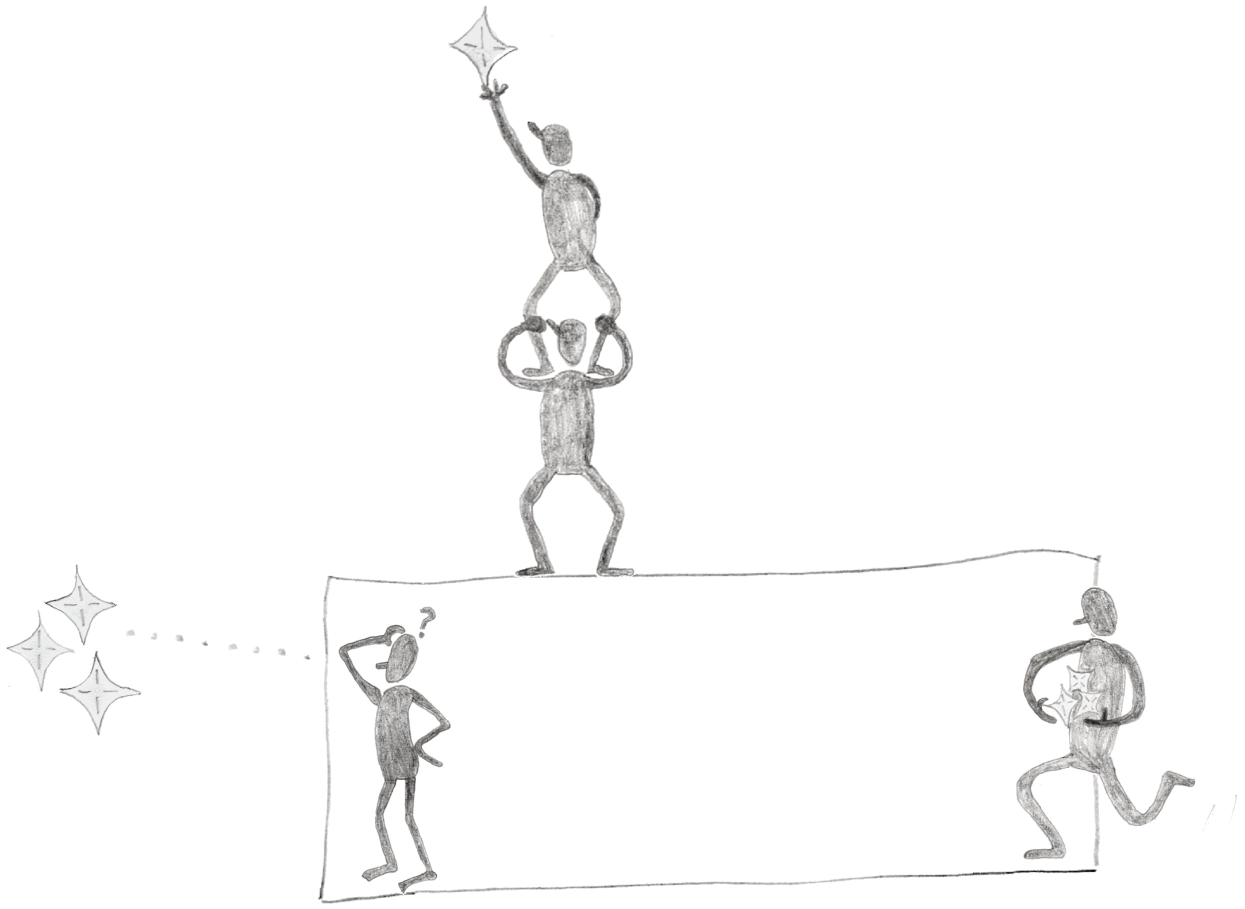
CHRISTOPHER PRESCHERN, B&R Industrial Automation GmbH

Even the simple task of returning data from functions becomes tricky, if the data to share becomes complex, or if the programmer has to cleanup the memory by himself or herself in case the programming language has no support in form of destructors or garbage collectors. To address this topic, this paper provides best practices in form of design patterns on the topic of returning data and provides C code examples.

Returning data from a function call is a task you are faced with when writing any kind of code which is longer than 10 lines and which you intend to be maintainable. Returning data is a simple task - you simply have to pass the data you want to share between two functions - in C you only have the possibility to directly return some value or to return data via emulated ”by-reference” parameters. There are not many choices and there is not much guidance to give - right? Wrong! Even the simple task of returning data from C functions is already tricky and there are many ways you can take of how to structure your program and of how to structure your function parameters.

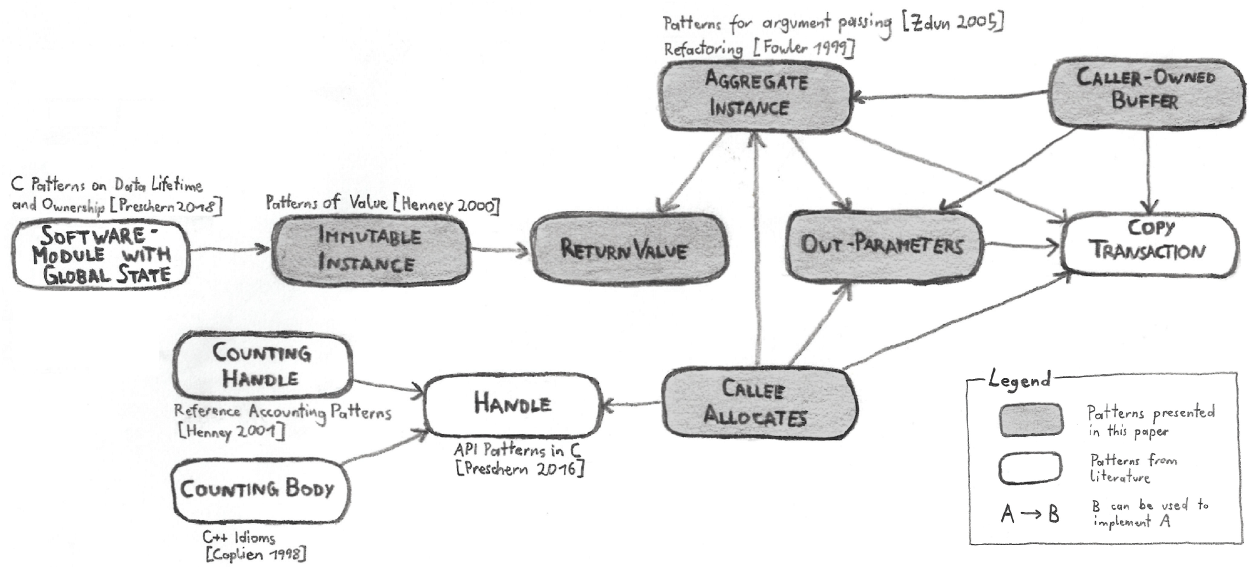
Especially in the C programming language, where you have to manage the memory allocation and deallocation on your own, passing complex data between functions becomes tricky, because there is no destructor or garbage collector, which helps you clean up the data. Now you are in the situation depicted in the figure on the following page and you have to ask yourself: Shall the data be put on the stack, or shall it be allocated? Who should allocate - the caller or the callee?

You might think that it is a bit late to write a paper tackling that issue and answering basic questions about passing data and data ownership. You might think that there must be many papers on that topic already out there. Wrong again! There is very little literature on C best practices on passing data between functions. That is the case, because on the one hand the topic is very specific, and on the other hand there is in general not much literature on C design patterns, because the topic of design patterns came up after object-oriented programming languages became state-of-the-art and therefore just few design patterns address procedural programming languages like C.



Problem addressed by this paper: Methods to retrieve data are unclear and not organized

This paper helps out from the situation depicted above and shows best practices in form of design patterns tailored for the C programming language describing how to share data between functions. These patterns help C programming beginners to understand techniques for returning data in C and they help advanced C programmers to better understand why these different techniques are applied. An overview of the presented patterns as well as related patterns from literature is shown in the following figure and the patterns are described as patlets in the appendix.


Patterns for argument passing [Zdun 2005]
Refactoring [Fowler 1999]
AGGREGATE INSTANCE
CALLER-OWNED BUFFER
C Patterns on Data Lifetime and Ownership [Preschern 2018]
Patterns of Value [Henney 2000]
SOFTWARE-MODULE WITH GLOBAL STATE
IMMUTABLE INSTANCE
RETURN VALUE
OUT-PARAMETERS
COPY TRANSACTION
COUNTING HANDLE
Reference Accounting Patterns [Henney 2001]
HANDLE
API Patterns in C [Preschern 2016]
CALLEE ALLOCATES
COUNTING BODY
C++ Idioms [Coplien 1998]
Legend
Patterns presented in this paper
Patterns from literature
A → B
B can be used to implement A

The paper presents the gray-colored patterns from the figure above and it shows the application of these patterns to the following running example.

Running example:

You want to implement the functionality to display diagnostic information of an Ethernet driver to the user. First, you simply add this functionality directly into the file with the Ethernet driver implementation and you directly access the variables which contain the required information.

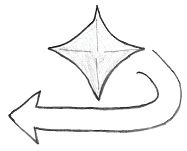
```
void ethShow()
{
  printf("%i packets received\n", driver.internal_data.rec);
  printf("%i packets sent\n", driver.internal_data.snd);
}
```

Later on, you realize that the functionality to display diagnostic information for your Ethernet driver will quite likely grow, so you decide to put it into a separate implementation file in order to keep your code clean. Now you need some simple way to transport the information from your Ethernet driver component to your diagnostics component.

One solution would be to use global variables to transport this information. However, if you use global variables, then the effort to split the implementation file was useless. You split the files, because you want to show that these code parts are not tightly coupled - with global variables you would bring that tight coupling back in.

A much better and very simple solution is the following: Let your Ethernet component have getter-functions which provide the desired information as RETURN VALUE.

Return Value

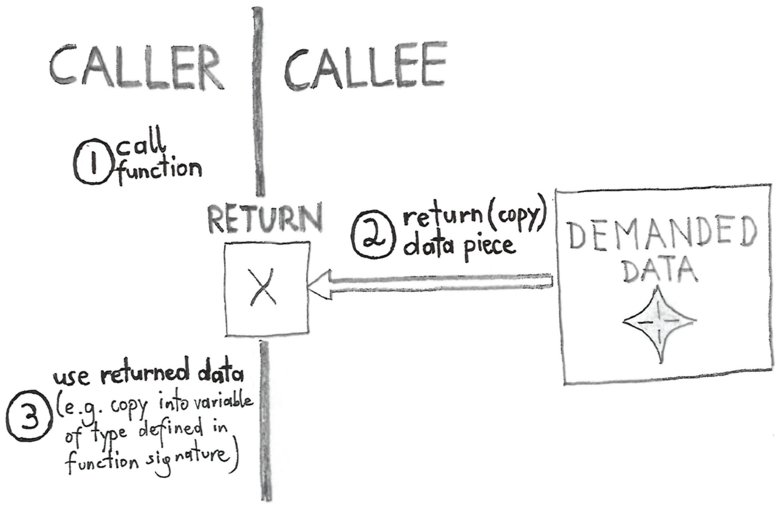


Problem:
You want to split your code into separate functions, as having everything in one function and in one implementation file is bad practice, because it gets difficult to read and to debug the code. However, your separate functions are not independent from one another. As it is usual in procedural programming, your functions deliver a result which is then needed by other functions. Your separate functions need to share some data.

You want to have a mechanism for sharing data which makes your code easy to understand. You want to make it explicit in your code which data is shared between functions and you want to make sure that functions don't communicate over side-channels not clearly visible in the code. Thus using global variables to return information to a caller is not a good solution for you, because global variables can be accessed and modified from any other part of the code and because it is not clear from the function signature, which exact global variable is used for returning data.

Also, global variables have the drawback that they could be used to store state information which could lead to different results for identical function calls. Aside from that, code using global variables for returning information would not be reentrant and it would not be safe to use in a multi-threaded environment.

Solution:
Simply use the one C mechanism intended to retrieve information about the result of a function call: The Return Value. The return-mechanism in C copies the function result and provides the caller access to this copy.


CALLER
CALLEE
① call function
RETURN
X
② return (copy) data piece
DEMANDED DATA
use returned data
③ (e.g. copy into variable of type defined in function signature)

Code Example

CALLER

```
int my_data = getData();
/* use my_data */
```

CALLEE

```
int getData()
{
  int requested_data;
  /* .... */
  return requested_data;
}
```

Consequences:
A RETURN VALUE allows the caller to retrieve a copy of the function result. No other code apart from the function implementation can modify this value and, as it is a copy, this value is solely used by the calling function. So compared to using global variables, it is more clearly defined which code influences the data retrieved from the function call.

Also, by not using global variables and using the copy of the function result instead, the function can be reentrant and it can safely be used in a multi-threaded environment.

However, for built-in C types, a function can return only a single object of the type specified in the function signature. It is not possible to define a function with several return types. You cannot, for example, have a function which returns an `int` and additionally another `int` and additionally a `double`. If you want to return more information than contained in just one simple, scalar C type, then you have to use an AGGREGATE INSTANCE or OUT-PARAMETERS.

Also, if you want to return data from an array, then the RETURN VALUE is not what you want, because it does not copy the content of the array, but only the pointer to the array and then the caller might end up with a pointer to data which ran out of scope. For returning arrays, you have to use other mechanisms like a CALLEE OWNED BUFFER or a CALLER ALLOCATED BUFFER.

Remember that whenever the simple RETURN VALUE mechanism is sufficient, then you should always take this most simple option to return data and you should not go for more powerful, but also more complex ways like OUT-PARAMETERS, AGGREGATE INSTANCE, CALLEE OWNED BUFFER or CALLER ALLOCATED BUFFER.

Known Uses and Related Patterns:
- Every C program uses this pattern. For example, every C program has a `main` function which already provides a return value to its caller (such as the operating system).

Running example:

Well... applying RETURN VALUE was quite obvious and simple. Now you have a new diagnostic component in an implementation file separate from the Ethernet driver and this component obtains the diagnostic information from the Ethernet driver in the following way:

```
void ethShow()
{
  int received_packets = ethernetDriverGetReceivedPackets();
  int sent_packets = ethernetDriverGetTotalSentPackets();
  printf("%i packets received\n", received_packets);
  printf("%i packets sent\n", sent_packets);
}
```

This code is very easy to read and if you want to add additional information, you can simply add additional functions to obtain this information. Now that is exactly what you want to do next: You want to show more information about the sent packets. You want to show the user how many packets were successfully sent and how many failed:

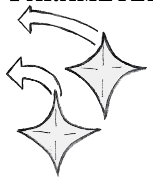
```
void ethShow()
{
  int received_packets = ethernetDriverGetReceivedPackets();
  int total_sent_packets = ethernetDriverGetTotalSentPackets();
  int successfully_sent_packets = ethernetDriverGetSuccesscullySentPackets();
  int failed_sent_packets = ethernetDriverGetFailedPackets();
  printf("%i packets received\n", received_packets);
  printf("%i packets sent\n", total_sent_packets);
  printf("%i packets successfully sent\n", successfully_sent_packets);
  printf("%i packets failed to send\n", failed_sent_packets);
}
```

With this code, after some time, you realize that sometimes, different from what you expected, `successfully_sent_packets` plus `failed_sent_packets` results in a number higher than `total_sent_packets`. That is, because your Ethernet driver runs in a separate thread and between your function calls to obtain the information, the Ethernet driver continues working and updates its packet information. So, if for example the Ethernet driver successfully sends a packet between your `ethernetDriverGetTotalSentPackets` call and your `ethernetDriverGetSuccesscullySentPackets` call, then the information which you show to the user is not consistent.

A possible solution would be to make sure that the Ethernet driver is not working while you call the functions to obtain the packet information. You could, for example, use a Mutex or a Semaphore to make sure of that. However, for such as simple task like obtaining packet statistics, you'd expect that you are not the one who has to cope with that.

As a much easier alternative you can return multiple pieces of information from one function call by using OUT-PARAMETERS.

OUT-PARAMETERS



Problem:
You want to provide data that represents related pieces of information from your component to a caller and these pieces of information change during runtime.

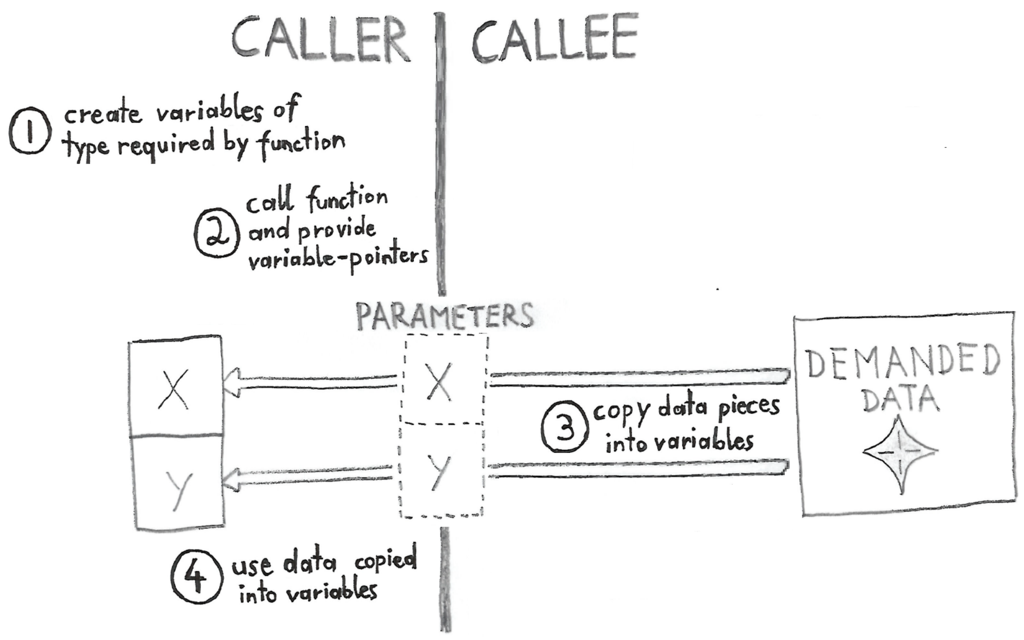
Using global variables to transport the data representing your pieces of information is not a good solution, because code using global variables for returning information would not be reentrant and it would not be safe to use in a multi-threaded environment. Aside from that, global variables can be accessed and modified from any other part of the code and when using global variables, it is not clear from the function signature, which exact global variables are used for returning the data. Thus global variables would make your code hard to understand and maintain. Also using the RETURN VALUES of multiple functions is not a good option, because the data you want to return is related, so splitting it across multiple function calls makes the code less readable.

As the pieces of data are related, the caller wants to retrieve a consistent snapshot of all this data. That becomes an issue when using multiple RETURN VALUES as soon as in a multithreaded environment the data can change at runtime. In that case, you would have to make sure that between the caller's multiple function calls, the data does not change. However, you cannot know whether the caller already finished reading all the data or whether there will be another piece of information which the caller wants to retrieve with another function call. Because of that, you cannot make sure that the data is not modified between the caller's function calls. When using multiple functions to provide related, consistent information, then you don't know the timespan during which the data must not change. Thus, with this approach, you cannot guarantee the caller to retrieve a consistent snapshot of the information.

Having multiple functions with RETURN VALUES also might not be a good solution, if a lot of preparation work is required for calculating the related pieces of data. If, for example you want to return the home- and the mobile-telephone-number for a specified person from a phone book and you'd have separate functions to retrieve the numbers, you'd have to separately search through the phone book for the entry of this person for each of the function calls. That requires unnecessary computation time and resources.

Solution:
Return all the data with one single function call. C does not support returning multiple types using the RETURN VALUE and C does not natively support by-reference arguments; however, by-reference arguments can be emulated by using pointers.

Have one single function with many pointer arguments. In the function implementation, de-reference the pointers and copy the data you want to return to the caller into the instance pointed to. Make sure in the function implementation, that the data does not change while copying. That can be achieved by mutual exclusion via Mutex or Semaphores (see COPY TRANSACTION).


CALLER
CALLEE
1 create variables of type required by function
2 call function and provide variable-pointers
PARAMETERS
X
Y
X
Y
DEMANDED DATA
3 copy data pieces into variables
4 use data copied into variables

Code Example

CALLER

```
int x,y;
getData(&x,&y);
/* use x,y */
```

CALLEE

```
void getData(int* x, int* y)
{
  *x = 42;
  *y = 78;
}
```

Consequences:
Now all data that represents related pieces of information are returned in one single function call and can be kept consistent with a COPY TRANSACTION. The function is reentrant and can safely be used in a multi-threaded environment.

For each additionally required data item, an additional pointer is passed to the function. That has the drawback, that if you want to return a lot of data, the function's parameter list becomes longer and longer. Having many parameters for one function is a bad code smell as it makes the code unreadable. That is why only rarely multiple OUT-PARAMETERS are used for a function and instead, to clean up the code, related pieces of information are returned with an AGGREGATE INSTANCE.

Also, for each piece of data, the caller has to pass a pointer to the function. That means that for each piece of data, an additional pointer has to be put onto the stack. If the caller's stack memory is very limited, that might become an issue.

OUT-PARAMETERS have the disadvantage, that when only looking at the function signature, they cannot clearly be identified as OUT-PARAMETERS. From the function signature, a caller can only guess whenever he or she sees a pointer, that might be an OUT-PARAMETER. However, such a pointer parameter could also be an input for the function.

Therefore, it has to be clearly described in the API documentation which parameters are for input and which are for output.

For simple, scalar C types the caller can simply pass the pointer to a variable as a function argument. For the function implementation all the information to interpret the pointer is specified, because of the specified pointer type. To return data with complex types, like arrays, either a CALLER-OWNED BUFFER has to be provided, or the CALLEE ALLOCATES and additional information about that data, like its size, has to be communicated.

Known Uses and Related Patterns:

- The Windows `RegQueryInfoKey` function returns information about a registry key via the function's OUT-PARAMETERS. The caller provides `unsigned long` pointers and the the functions writes, amongst other pieces of information, the number of sub-keys and the size of the key's value into the `unsigned long` variables being pointed to.
- Apple's Cocoa API for C programs uses an additional `NSError` parameter, which stores errors occurring during the function calls.
- The function `userAuthenticate` of the real-time operating system VxWorks uses RETURN VALUES to return information, whether a provided password is correct for a provided login name. Additionally the function takes an OUT-PARAMETER to return the user ID associated with the provided login name.

Running example:

By applying OUT-PARAMETERS you'll get the following code:

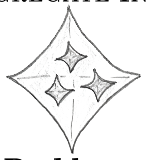
```
void ethShow()
{
  int received_packets = ethernetDriverGetReceivedPackets();
  int total_sent_packets, success_sent_packets, failed_sent_packets;
  ethernetDriverGetStatus(&total_sent_packets, &success_sent_packets, &failed_sent_packets);
  printf("%i packets received\n", received_packets);
  printf("%i packets sent\n", total_sent_packets);
  printf("%i packets successfully sent\n", success_sent_packets);
  printf("%i packets failed to send\n", failed_sent_packets);
}
```

To retrieve information about sent packets, you only have one single function call to the Ethernet driver and the Ethernet driver can make sure that the data delivered within this call is consistent.

You consider also retrieving the `received_packets` in one and the same function call with the sent packets; however, you realize that the one function call becomes more and more complicated. Having the one function call with three OUT-PARAMETERS is already complicated to write and to read. When calling the functions, the parameters could easily be mixed up. Adding a fourth parameter wouldn't make the code better.

To make the code more readable, an AGGREGATE INSTANCE can be used.

AGGREGATE INSTANCE



Problem:
You want to provide data that represents related pieces of information from your component to a caller and these pieces of information change during runtime.

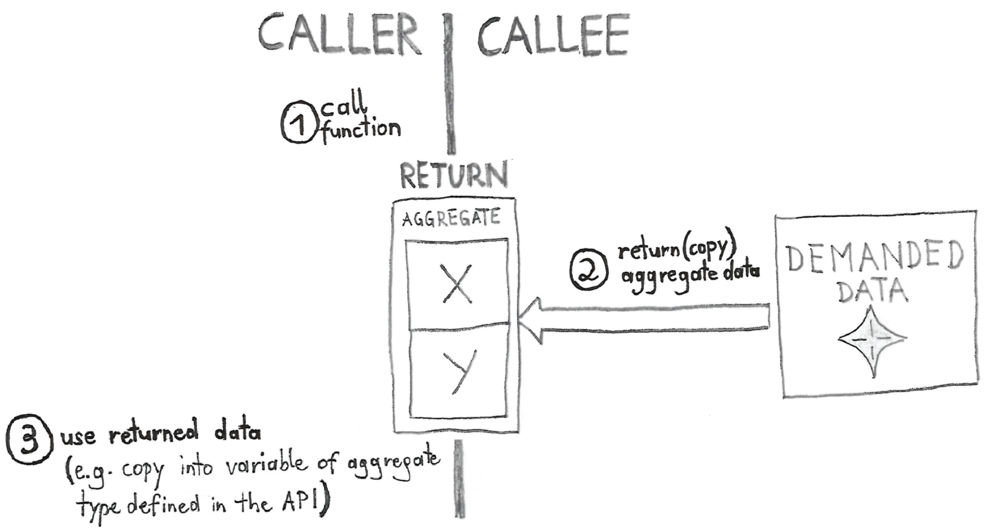
Using global variables to transport the data representing your pieces of information is not a good solution, because code using global variables for returning information would not be reentrant and it would not be safe to use in a multi-threaded environment. Aside from that, global variables can be accessed and modified from any other part of the code and when using global variables, it is not clear from the function signature, which exact global variables are used for returning the data. Thus global variables would make your code hard to understand and maintain. Also using the RETURN VALUES of multiple functions is not a good option, because the data you want to return is related, so splitting it across multiple function calls makes the code less readable. Having one single function with many OUT-PARAMETERS also is not a good idea, because if you have many such OUT-PARAMETERS, it gets easy to mix them up and your code becomes unreadable.

As the pieces of data are related, the caller wants to retrieve a consistent snapshot of all this data. That becomes an issue when using multiple RETURN VALUES as soon as in a multithreaded environment the data can change at runtime. In that case, you would have to make sure that between the caller's multiple function calls, the data does not change. However, you cannot know whether the caller already finished reading all the data or whether there will be another piece of information which the caller wants to retrieve with another function call. Because of that, you cannot make sure that the data is not modified between the caller's function calls. When using multiple functions to provide related information, then you don't know the timespan during which the data must not change. Thus, with this approach, you cannot guarantee the caller to retrieve a consistent snapshot of the information.

Having multiple functions with RETURN VALUES also might not be a good solution, if a lot of preparation work is required for calculating the related pieces of data. If, for example you want to return the home- and the mobile-telephone-number for a specified person from a phone book and you'd have separate functions to retrieve the numbers, you'd have to separately search through the phone book for the entry of this person for each of the function calls. That requires unnecessary computation time and resources.

Solution:
Put all data which is related together into a newly defined type. Define this AGGREGATE INSTANCE to contain all the related data which you want to share. Define it in the interface of your component to let the caller directly access all the data stored in the instance.

To implement that, define a struct in your header file and define all types to be returned from the called function as members of this struct. In the function implementation, copy the data to be returned into the struct members. Make sure in the function implementation, that the data does not change while copying. That can be achieved by mutual exclusion via Mutex or Semaphores (see COPY TRANSACTION).


CALLER
CALLEE
1 call function
RETURN
AGGREGATE
x
y
2 return (copy) aggregate data
DEMANDED DATA
3 use returned data (e.g. copy into variable of aggregate type defined in the API)

To actually return the struct to the caller, there are two main options:

- Pass the whole struct as RETURN VALUE. C allows not only built-in types to be passed as RETURN VALUE of functions, but also user-defined types such as a struct can be passed.
- Pass a pointer to the struct using an OUT-PARAMETER. However, when only passing pointers, the issue of who provides and owns the memory being pointed to comes up. That issue is addressed in CALLER-OWNED BUFFER and CALLEE ALLOCATES. Alternatively to passing a pointer and letting the caller directly access the AGGREGATE INSTANCE, you could consider hiding the struct from the caller by using a HANDLE.

Code Example

CALLER

```
struct AggregateInstance my_instance;
my_instance = getData();
/* use my_instance.x
   use my_instance.y, ... */
```

CALLEE

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

struct AggregateInstance getData()
{
  struct AggregateInstance inst;
  /* fill inst.x and inst.y */
  return inst;
}
```

When returning, the `getData` function copies the content of `inst` for the caller. So even though `inst` runs out of scope, the caller can access the copied content.

Consequences:
Now the caller can retrieve multiple data that represents related pieces of information via the AGGREGATE INSTANCE with one single function call. The function is reentrant and can safely be used in a multi-threaded environment.

That provides the caller with a consistent snapshot of the related pieces of information and it makes the caller's code clean, because he or she does not have to call multiple functions or one function with many OUT-PARAMETERS.

When passing data between functions without pointers by using RETURN VALUES, all this data is put on the stack. When passing one struct along 10 nested functions, then this struct is on the stack 10 times. In some cases that is no problem, but in some other cases it is - especially if the struct is too large and you don't want to waste stack memory by copying the whole struct onto the stack every time. Because of that, quite often instead of directly passing or returning a struct, a pointer to that struct is passed or returned.

When passing pointers to the struct, or if the struct contains pointers, you have to keep in mind that C does not perform the work of doing a deep copy for you. C only copies the pointer values and does not copy the instances they point to. That might not be what you want, so you have to keep in mind that as soon as pointers come into play, you have to deal with providing and cleaning up the memory being pointed to. This issue is addressed in CALLER-OWNED BUFFER and CALLEE ALLOCATES.

Known Uses and Related Patterns:

- [Zdun 2005] describes this pattern including C++ examples as CONTEXT OBJECT and [Fowler 1999] describes it as PARAMETER OBJECT.
- The code of the game Nethack stores monster-attributes in AGGREGATE INSTANCES and provides a function for retrieving this information [Smith 2014].
- The implementation of the text editor `sam` copies structs when passing them to functions and when returning them from functions in order to keep the code simpler.

Running example:

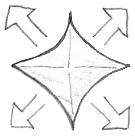
With the AGGREGATE INSTANCE, you'll get the following code:

```
void ethShow()
{
  struct EthernetDriverInfo eth_stat = ethernetDriverGetStatistics();
  printf("%i packets received\n", eth_stat.received_packets);
  printf("%i packets sent\n", eth_stat.total_sent_packets);
  printf("%i packets successfully sent\n", eth_stat.successfully_sent_packets);
  printf("%i packets failed to send\n", eth_stat.failed_sent_packets);
}
```

Now you have one single call to the Ethernet driver and the Ethernet driver can make sure that the data delivered within this call is consistent. Also, your code looks cleaned up, because the data which belongs together now is collected in a single struct.

Next, you want to show some more information about the Ethernet driver to your user. You want to show the user to which Ethernet interface the packet statistics information belongs to and thus you want to show the driver name including a textual description of the driver. Both is contained in a string stored in the Ethernet driver component. The string is quite long and you don't exactly know how long the string is. Luckily the string does not change during runtime, so you can access an IMMUTABLE INSTANCE.

Immutable Instance



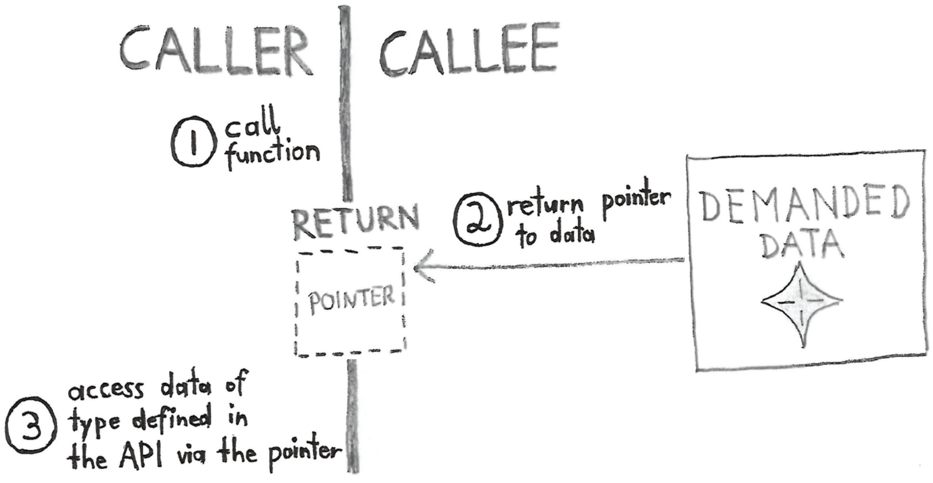
Problem:
You want to provide information held in large pieces of data (arrays, long strings, ...) from your component to a caller. Luckily you are in the situation, that the data you want to provide to the caller is fixed at compile time or at boot time and does not change at runtime.

In that case, copying the data for each and every caller would be a waste of memory, so providing all the data by returning an Aggregate Instance or by copying all the data into Out-Parameters is not an option due to stack memory limitations.

Solution:
Create an instance (e.g a struct with the data to share) in static memory. Write the data to be contained in the instance at compile-time or at boot-time and do not change it at runtime anymore (see Stateful Software-Module for initialization variants).

Return a pointer to this instance for callers who want to access this data and make sure the callers cannot modify this data. To achieve that, make the data being pointed to `const`. Callers obtain a reference, but they don't get ownership of the memory.

Even if multiple callers (and multiple threads) access the instance at the same time, they don't have to care about each other, because the instance does not change and is thus always in a consistent state and contains the required information.



Code Example

CALLER

```c
const struct ImmutableInstance*
             my_instance;

my_instance = getData();
/* use my_instance->x,
   use my_instance->y, ... */
```

CALLEE API

```c
struct ImmutableInstance
{
  int x;
  int y;
};
const struct ImmutableInstance* getData();
```

CALLEE IMPLEMENTATION

```c
static struct ImmutableInstance
                inst = {12, 42};
const struct ImmutableInstance* getData()
{
   return &inst;
}
```

Consequences:
The caller can call one simple function to get access to even complex or large data and does not have to care about where this data is stored. The caller does not have to provide buffers where this data can be stored in, does not have to cleanup memory, and does not have to care about the lifetime of the data - it simply always exists.

The caller can read all data via the retrieved pointer. The simple function for retrieving the pointer is reentrant and can safely be used in multi-threaded environments. Also the data can safely be accessed in multi-threaded environments, because it does not change at runtime and multiple threads which only read the data are no problem.

However, the data cannot be changed at runtime without taking further measures. If it is required that the caller can change the data, then a COUNTING BODY or COUNTING HANDLE in combination with COPY ON WRITE can be implemented. If the data in general can change at runtime, then an IMMUTABLE INSTANCE isn't an option and instead, for sharing complex and large data, a CALLER-OWNED BUFFER has to be used or the CALLEE ALLOCATES.

Known Uses and Related Patterns:
- [Henney 2000] describes the similar IMMUTABLE OBJECT pattern in detail and provides C++ code examples.
- The code of the game Nethack stores immutable monster-attributes in an IMMUTABLE INSTANCE and provides a function for retrieving this information [Smith 2014].

Running example:

Usually, returning a pointer to access data stored within a component is tricky, because if multiple callers access (and maybe write) this data, then a plain pointer isn't the solution for you, because you never know whether the pointer you have is still valid and whether the data contained in this pointer is consistent. However, in this case we are lucky, because we have an IMMUTABLE INSTANCE. The driver name and description are both information which is determined at compile-time and which does not change afterwards. Thus, we can simply retrieve a constant character pointer to this data.

```
void ethShow()
{
  struct EthernetDriverStat eth_stat = ethernetDriverGetStatistics();
  printf("%i packets received\n", eth_stat.received_packets);
  printf("%i packets sent\n", eth_stat.total_sent_packets);
  printf("%i packets successfully sent\n", eth_stat.successfully_sent_packets);
  printf("%i packets failed to send\n", eth_stat.failed_sent_packets);

  const struct EthernetDriverInfo* eth_info = ethernetDriverGetInfo();
  printf("Driver name: %s\n", eth_info->name);
  printf("Driver description: %s\n", eth_info->description);
}
```

As a next step, additionally to the name and description of the Ethernet interface, you also want to show the user the currently configured IP address and subnet mask. The addresses are stored as a string in the Ethernet driver. Both addresses are information which might change during runtime, so you cannot simply return a pointer to an IMMUTABLE INSTANCE.

While it would be possible to have the Ethernet driver pack these strings into an AGGREGATE INSTANCE and simply return this instance (arrays in a struct are copied when returning the struct), such a solution is rather uncommon for large amounts of data, because it consumes a lot of stack memory. Usually, instead, pointers are used.

Using pointers is the exact solution you are looking for - use a CALLER-OWNED BUFFER.

CALLER-OWNED BUFFER

Problem:
You want to provide complex or large data of known size to the caller. The data changes at runtime (maybe because you provide the callers with functions to write the data), so you cannot simply provide the caller with a pointer to static data (as it is the case with an IMMUTABLE INSTANCE). You cannot do that, because if you simply provide the callers with such a pointer, you'd run into the problem, that the data one caller reads might be inconsistent (partially overwritten), because, in a multi-threaded environment, another caller might simultaneously write that data.

Simply copying all the data into an AGGREGATE INSTANCE and passing it via the RETURN VALUE to the caller is not an option, because, as the data is large, it cannot be passed via the stack which only has very limited memory.

When instead only returning a pointer to the AGGREGATE INSTANCE, there would be no problem with stack memory limitations anymore, but you have to keep in mind that C does not do the work of performing a deep copy for you. C only returns the pointer. You have to make sure that the data (stored in an AGGREGATE INSTANCE or in an array) being pointed to is still valid after the function call. For example, you cannot store the data in auto-variables within your function and provide a pointer to these variables, because after the function call, the variables run out of scope.

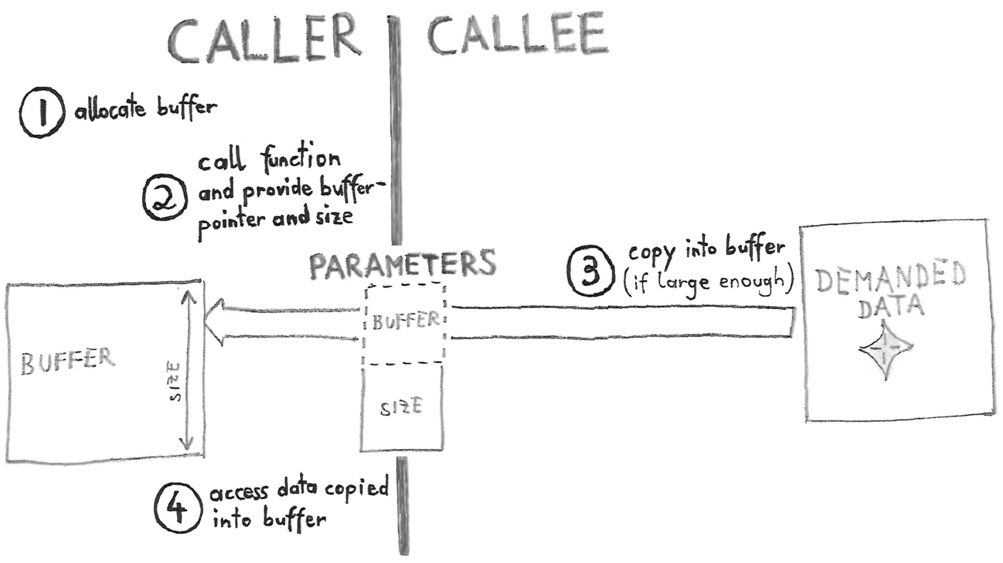
Now the question where the data should be stored arises. It has to be clarified whether the caller or the callee should provide the required memory and it has to be clarified whether the caller or the callee is then responsible for managing and cleaning up the memory.

Solution:
Let the caller provide a buffer and its size to the function which returns the complex, large data. In the function implementation, copy the required data into the buffer if the buffer size is large enough. Make sure that the data does not change while copying. That can be achieved by mutual exclusion via Mutex or Semaphores (see COPY TRANSACTION).

The caller then has a snapshot of the data in the buffer, is the sole owner of this snapshot, and thus can consistently access this snapshot even if the original data changes in the meantime.

The caller can provide the buffer and the size each as a separate function parameter, or the caller can pack the buffer and the size into an AGGREGATE INSTANCE and pass a pointer to the AGGREGATE INSTANCE to the function.

As the caller has to provide the buffer and the size to the function, the caller has to know the size beforehand. To let the caller know of which size the buffer has to be, the information of the required size has to be present in the API. That can be implemented by defining the size as a macro or by defining a struct containing a buffer of the required size in the API.


CALLER
CALLEE
1 allocate buffer
2 call function and provide buffer-pointer and size
PARAMETERS
BUFFER
SIZE
3 copy into buffer (if large enough)
DEMANDED DATA
BUFFER
SIZE
4 access data copied into buffer

Code Example

CALLER

```
struct Buffer buffer;
getData(&buffer);
/* use buffer.data */
```

CALLEE API

```
struct Buffer
{
  char data[256];
};
void getData(struct Buffer* buffer);
```

CALLEE IMPLEMENTATION

```
void getData(struct Buffer* buffer)
{
  memcpy(buffer->data, some_data, 256);
}
```

Consequences:
The complex, large data can be consistently provided to the caller with one single function call. The function is reentrant and can safely be used in a multi-threaded environment. Also the caller can safely access the data in multi-threaded environments, because the caller is the sole owner of the buffer.

The caller can decide the memory type for the buffer. The caller can put the buffer on the stack and benefit from the advantage that stack memory will be cleaned up after the variable runs out of scope. Alternatively the caller can put the memory on the heap to determine the lifetime of the variable or to not waste stack memory. Also, the calling function might only have a reference to a buffer obtained by its calling function. In this case this buffer can simply be passed on and there is no need to have multiple buffers.

The time-intensive operation of allocating and freeing memory is not performed during the function call. The caller can determine when these operations take place and thus the function call becomes quicker and more deterministic.

From the API it is absolutely clear that the caller has ownership of the buffer. The caller has to provide the buffer and the caller has to clean it up afterwards. If the caller allocated the buffer, he or she is the one responsible for freeing it afterwards.

The caller has to know the size of the buffer beforehand and knowing this size, the function can safely operate in the buffer - e.g. the function can even handle unterminated strings transported via the buffer. However, in some cases the caller might not know the exact size required and it would be better if instead the CALLEE ALLOCATES.

Known Uses and Related Patterns:

- The Nethack code uses this pattern to provide the information about a savegame to the component which then actually stores the game on the disk [Smith 2014].
- The B&R Automation Runtime operating system uses this pattern for a function to retrieve the IP address.
- The C stdlib function `gets` stores the input into a provided buffer.

Running example:

You now provide a CALLER-OWNED BUFFER to the Ethernet driver function and the function copies its data into this buffer. You have to know beforehand how large the buffer has to be. In case of obtaining the IP address string that is no problem, because the string has a fixed size. So you can simply put the buffer for the IP address on the stack and provide this stack variable to the Ethernet driver. Alternatively it would have been possible to allocate the buffer on the heap, but in this case that is not required, because the size of the IP address is known and because the size of the data is small enough to fit on the stack.

```c
void ethShow()
{
  struct EthernetDriverStat eth_stat = ethernetDriverGetStatistics();
  printf("%i packets received\n", eth_stat.received_packets);
  printf("%i packets sent\n", eth_stat.total_sent_packets);
  printf("%i packets successfully sent\n", eth_stat.successfully_sent_packets);
  printf("%i packets failed to send\n", eth_stat.failed_sent_packets);

  const struct EthernetDriverInfo* eth_info = ethernetDriverGetInfo();
  printf("Driver name: %s\n", eth_info->name);
  printf("Driver description: %s\n", eth_info->description);

  struct IpAddress ip;
  ethernetDriverGetIp(&ip);
  printf("IP address: %s\n", ip.address);
}
```

Next, you want to extend your diagnostic component to also print a dump of the last received packet. This now is a piece of information which is too large to put it on the stack and because Ethernet packets have variable size, you cannot know beforehand how large the buffer for the packet has to be, so CALLER-OWNED BUFFER isn't an option for you.

You could, of course simply have functions `EthernetDriverGetPacketSize()` and `EthernetDriverGetPacket(buffer)`, but here again you'd have the problem that you'd have to call two functions and between the two function calls the Ethernet driver could receive another packet which would make your data inconsistent. Also this solution is not very elegant, because you'd have to call two different functions to achieve one purpose.

Instead, it is much easier, if the CALLEE ALLOCATES.

Callee Allocates

Problem:
You want to provide complex or large data to the caller. The data changes at runtime (maybe because you provide the callers with functions to write the data), so you cannot simply provide the caller with a pointer to static data (as it is the case with an Immutable Instance). You cannot do that, because if you simply provide the callers with such a pointer, you'd run into the problem, that the data one caller reads might be inconsistent (partially overwritten), because, in a multi-threaded environment, another caller might simultaneously write that data.

Simply copying all the data into an Aggregate Instance and passing it via the Return Value to the caller is not an option, because, with the Return Value you can only pass data of known size and because, as the data is large, it cannot be passed via the stack which only has very limited memory.

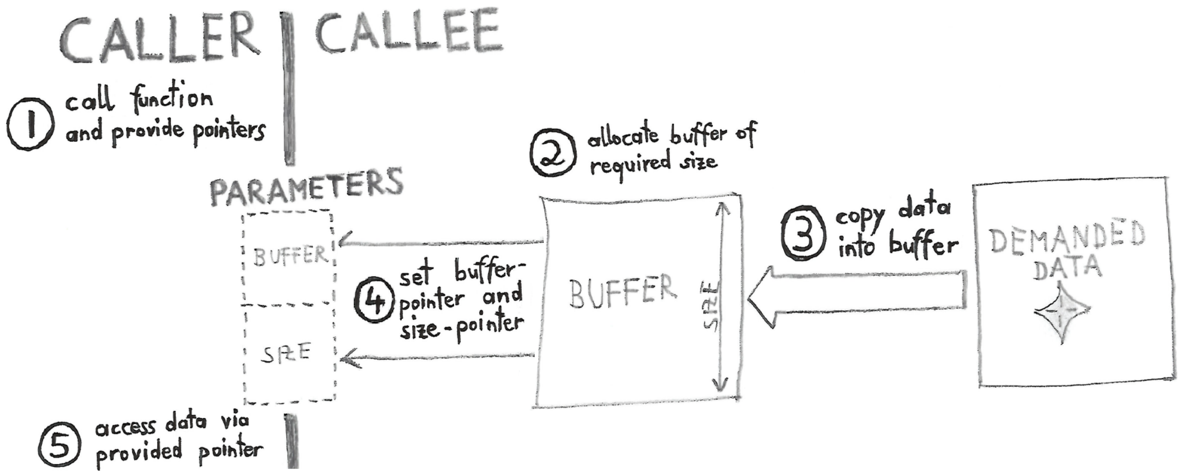
When instead only returning a pointer to the Aggregate Instance, there would be no problem with stack memory limitations anymore, but you have to keep in mind that C does not do the work of performing a deep copy for you. C only returns the pointer. You have to make sure that the data (stored in an Aggregate Instance or in an array) being pointed to is still valid after the function call. For example, you cannot store the data in auto-variables within your function and provide a pointer to these variables, because after the function call, the variables run out of scope and are being cleaned up.

Now the problem arises, where the data should be stored. It has to be clarified whether the caller or the callee should provide the required memory and it has to be clarified whether the caller or the callee is then responsible for managing and cleaning up the memory.

The amount of data you want to provide is not fixed at compile time - e.g. you want to return a string of beforehand unknown size. That makes using a Caller-Owned Buffer impractical, because the caller does not know the size of the buffer beforehand. The caller could beforehand ask for the required buffer size (e.g. with a `getRequiredBufferSize()` function), however that also is impractical, because in order to retrieve one piece of data, the caller would have to make multiple function calls. Also, maybe the data you want to provide could change between those function calls and then the caller again would provide a buffer of the wrong size.

Solution:
Allocate a buffer with the required size inside the function which provides the complex, large data. Copy the required data into the buffer. Make sure that the data does not change while copying. That can be achieved by mutual exclusion via Mutex or Semaphores (see Copy Transaction).

Provide a pointer to the buffer and its size to the caller as Out-Parameters. After the function call, the caller can operate on the buffer and knows its size. The caller has ownership of the buffer, determines its lifetime, and thus is responsible for cleaning it up.


CALLER
CALLEE
1 call function and provide pointers
PARAMETERS
BUFFER
SIZE
2 allocate buffer of required size
BUFFER
SIZE
3 copy data into buffer
DEMANDED DATA
4 set buffer-pointer and size-pointer
5 access data via provided pointer

Code Example

CALLER

```
    char* buffer;
    int size;
    getData(&buffer, &size);
    /* use buffer */
    free(buffer);
  }
```

CALLEE IMPLEMENTATION

```
    void getData(char** buffer, int* size)
    {
      *size = data_size;
      *buffer = malloc(data_size);
      /* write data to buffer */
    }
```

Alternatively, the pointer to the buffer and the size can be put into an AGGREGATE INSTANCE provided as RETURN VALUE. To make it more clear for the caller that there is a pointer in the AGGREGATE INSTANCE which has to be freed, the API can provide an additional function for cleaning it up. When also providing a function to clean up, the API already looks very similar to an API with a HANDLE, which would bring the additional benefit of flexibility while maintaining API compatibility.

No matter whether the called function provides the buffer via an AGGREGATE INSTANCE or via OUT-PARAMETERS, it has to be made clear to the caller, that the caller owns the buffer is responsible for freeing it. That has to be well documented in the API.

Consequences:
The caller can retrieve the buffer of beforehand unknown size with one single function call. The function is reentrant, can safely be used in multi-threaded environments, and provides the caller with consistent information about the buffer and its size. Knowing the size, the caller can safely operate on the data - e.g. the caller can even handle unterminated strings transported via such buffers.

The caller has ownership of the buffer, determines its lifetime, and is responsible for freeing it. From looking at the interface, it has to be made very clear that the caller has to do that. One possibility to make that clear, is to document it in the API. Another approach is to have an explicit cleanup-function to make it more obvious that something has to be cleaned up. Such a cleanup function has the additional advantage that the same component which allocates the memory also frees it. That is important, if the two involved components are compiled with different compilers or if they run on different platforms - in such cases the functions for allocating and freeing memory could differ between the components which makes it mandatory that the same component which allocates also frees.

The caller cannot determine which kind of memory should be used for the buffer - that would have been possible with a CALLER-OWNED BUFFER. Now the caller must use the provided kind of memory, which is allocated inside the function call.

Allocating takes time, which means that compared to CALLER-OWNED BUFFER, the function call becomes slower and less deterministic.

Known Uses and Related Patterns:

- The `strdup` function allocates the duplicated string and returns it.
- The `getifaddrs` Linux function provides information about configured IP addresses. The data holding this information is stored in a buffer allocated by the function.
- The Nethack code uses this pattern to retrieve buffers [Smith 2014].

Running example:

Your diagnostic component retrieves the packet data in a buffer which the CALLEE ALLOCATES:

```
void ethShow()
{
  struct EthernetDriverStat eth_stat = ethernetDriverGetStatistics();
  printf("%i packets received\n", eth_stat.received_packets);
  printf("%i packets sent\n", eth_stat.total_sent_packets);
  printf("%i packets successfully sent\n", eth_stat.successfully_sent_packets);
  printf("%i packets failed to send\n", eth_stat.failed_sent_packets);

  const struct EthernetDriverInfo* eth_info = ethernetDriverGetInfo();
  printf("Driver name: %s\n", eth_info->name);
  printf("Driver description: %s\n", eth_info->description);

  struct IpAddress ip;
  ethernetDriverGetIp(&ip);
  printf("IP address: %s\n", ip.address);

  struct Packet* packet = ethernetDriverGetPacket();
  printf("Packet Dump:");
  fwrite(packet->data, 1, packet->size, stdout);
  freePacket(packet);
}
```

With this final version of the diagnostic component we can see all the presented ways of how to retrieve information from another function. Mixing all these ways in one piece of code might not be what you actually want to do, because it gets a bit confusing to have one piece of data on the stack and to have another piece of data on the heap.

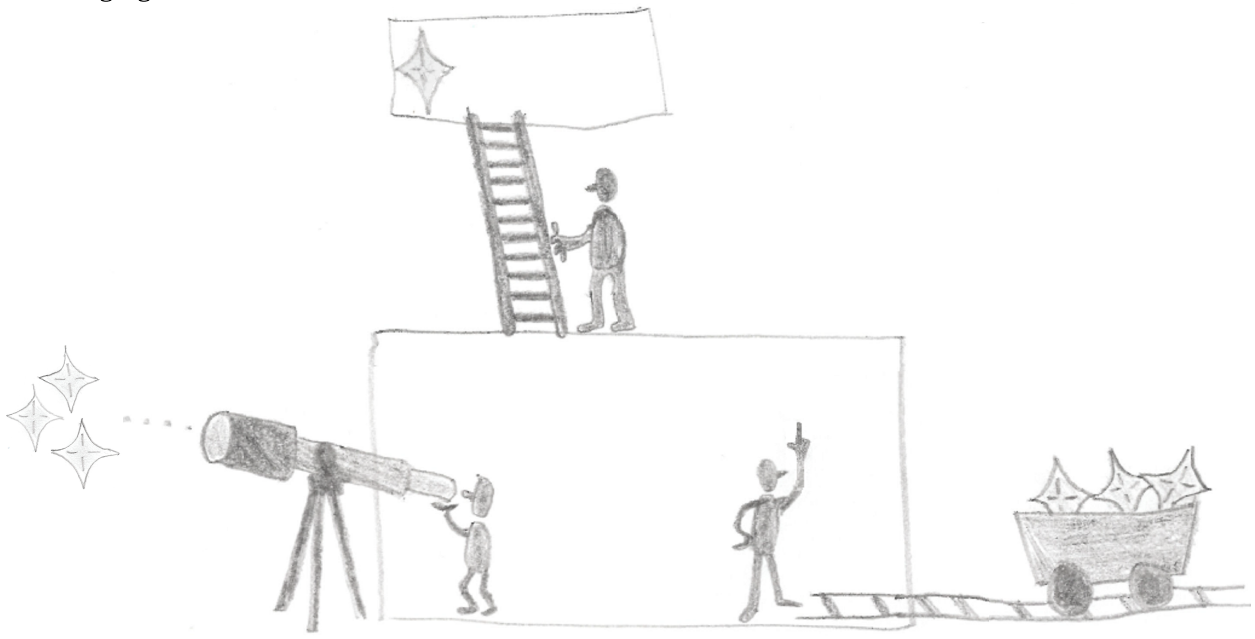
As soon as you allocate buffers, you don't want to mix different approaches, so using CALLER-OWNED BUFFER and CALLER ALLOCATES in one single function might not be what you want to do. Instead, pick the one approach that suits all your needs and stick to that within one function or within one component. That makes your code more uniform and easier to understand.

However, if you just have to obtain a single piece of data from another component and if you have the choice to use the easier alternatives to retrieve data (the patterns presented earlier in this paper), then always do that to keep your code simple. For example, if you have the possibility to put buffers on the stack, then do that, because it saves you the effort to free the buffer.

CONCLUSION

This paper showed different ways of how to return data from functions and on how to handle buffers in C. The simplest way is to use RETURN VALUE to return a single piece of data. However, if multiple pieces of related data have to be returned, then instead OUT-PARAMETERS or, even better, AGGREGATE INSTANCE have to be used. In case the data to be returned does not change at runtime, IMMUTABLE INSTANCE can be used. When returning data in a buffer, CALLER-OWNED BUFFER can be used if the size of the buffer is known beforehand and CALLEE ALLOCATES can be used if the size is unknown beforehand.

With the patterns from this paper, a C programmer has some basic guidance on how to transport data between functions and on how to cope with returning, allocating, and freeing buffers. A programmer using C now has tools to retrieve data from other functions in a structured way as depicted in the following figure.



Situation when having the patterns: Well-defined methods and tools to transport data between functions

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.

ACKNOWLEDGMENTS

I want to thank my shepherd James Coplien, who taught me a lot about pattern basics and who made sure that I get my C terminology right. I also want to thank my work colleague Thomas Havlovec for providing feedback on the code examples.

APPENDIX - PATLETS

Pattern Name	Pattern Solution Sketch
RETURN VALUE	Simply use the one C mechanism intended to retrieve information about the result of a function call: The Return Value. The return-mechanism in C copies the function result and provides the caller access to this copy.
OUT-PARAMETERS	Return all the data with one single function call. C does not support returning multiple types using the RETURN VALUE and C does not natively support by-reference arguments, but by-reference arguments can be emulated by passing pointers as function parameters and by writing data to the memory pointed to.
AGGREGATE INSTANCE	Define an instance which contains all the related information which you want to share. Define this instance in the interface of the component which provides this information and let the user directly access information from its members [Zdun 2005].
IMMUTABLE INSTANCE	Create an instance in the static memory. Fill data into this instance at compile-time or at boot-time. Return a pointer this instance to callers who want to access this data and make sure that the callers cannot modify the data [Henney 2000].
CALLER-OWNED BUFFER	Let the caller provide a buffer and its size to the function for returning complex, large data. In the function implementation, copy the required data into the buffer if the buffer size is large enough.
CALLEE ALLOCATES	Allocate a buffer with the required size inside the function for returning complex, large data. Copy the required data into the buffer and return a pointer to this buffer to the caller.
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 [Preschern 2016] [Coplien 1998].
COUNTING HANDLE	Have a handle object through which the user works on a shared object. The handle encapsulates the responsibility for tracking references to the shared object and for its deletion [Henney 2001].
COUNTING BODY	Manage logical sharing and proper resource deallocation of objects that use dynamically allocated resources [Coplien 1998].
COPY ON WRITE	Treat the body as an immutable object for all query operations on the handle. Any operations that require change to the representation must ensure that it is not shared, making their own copy of it if necessary [Henney 2001].
STATEFUL SOFTWARE-MODULE	Put related functions together into one API and put all function implementations in one single C file. Use global variables to communicate state information. [Preschern 2018].
COPY TRANSACTION	*This pattern is not yet written and might be subject to future work.* The pattern tackles the problem that data has to be copied in a multi-threaded environment, while others want to access the data (read and write).

REFERENCES

James Coplien. 1998. C++ Idioms. In *Proceedings of the 3rd European Conference on Pattern Languages of Programming (EuroPLoP)*.

Martin Fowler. 1999. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley.

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)*.

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. C Patterns on Objects and their Lifetime. In *Proceedings of the 23rd European Conference on Pattern Languages of Programming (EuroPLoP)*.

Alex Smith. 2014. Memory management in C programs. (2014). http://nethack4.org/blog/memory.html

Uwe Zdun. 2005. Patterns of Argument Passing. In *Proceedings of the 4th Nordic Conference of Pattern Language of Programs (VikingPLoP)*.