Patterns for Returning Error Information in C

CHRISTOPHER PRESCHERN, B&R Industrial Automation GmbH

Error handling is part of any industrial strength code. In programming languages like C, which have no support for enhanced error handling mechanisms like exceptions, error handling is a non-trivial task and many decisions on how to implement it have to be made. This paper documents best practices for these decisions as well as their benefits and liabilities in form of design patterns on the topic of how to return error information in the C programming language.

1. INTRODUCTION

Error handling is a major concern for every program. For every program, the programmer has to decide how to react on errors arising in his own code, how to react on errors arising in 3rd party code, how to pass this error information along in the code, and how to present this error information to the user.

Most object-oriented programming languages come with the handy mechanism of exceptions to provide the programmer with an additional channel for transporting error information. However, programming languages like C do not natively provide such a mechanism. There are ways to emulate exception handling or even inheritance among exceptions in C as presented in [Schreiner 1993] or [Moen 1992]. However, for C programmers working on legacy C code or for C programmers who want to stick to the native C style they are used to, introducing such exception mechanisms is not the way to go. Instead such C programmers need guidance on how to use the mechanisms for error handling already natively present in the C programming language.

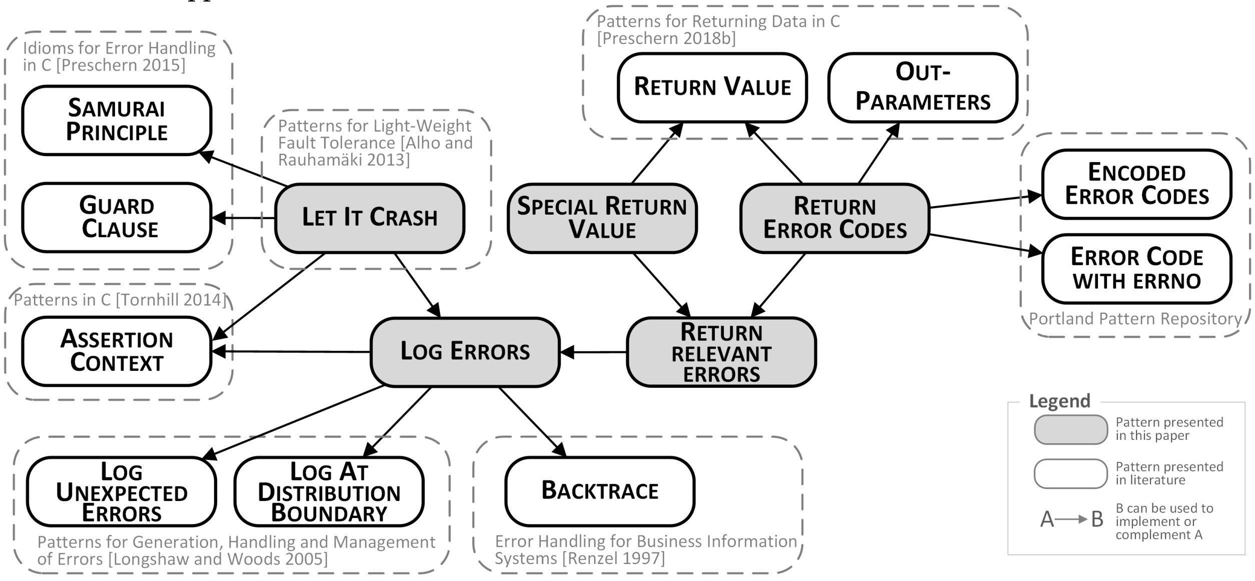
That is where this paper comes in. The paper is part of a series of papers on patterns for the C programming languages and it describes how to transport error information in the C programming language. Compared to a previous paper of this series [Preschern 2015], which focused on how error handling within the implementation of a single C function can be improved, this paper focuses on how error information can be transported between functions or between software-modules.

2. OVERVIEW OF PATTERNS PRESENTED IN THIS PAPER

The following patterns on error handling in C are described in this paper. The pattern RETURN ERROR CODES describes to provide the caller with numeric codes representing an occurring errors. RETURN RELEVANT ERRORS suggests to only transport error information to the caller, if the caller can react on these errors in the code and SPECIAL RETURN VALUE and LET IT CRASH show how to do that. LOG ERRORS suggests to provide an additional channel to transport error information which is not intended for the caller, but for the user or for debugging purposes.

EuroPLoP '19, July 3-7, 2019, Irsee, Germany
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6387-7/17/07...$15.00
https://doi.org/10.1145/XXXXXXX

Throughout this paper, these patterns are presented in detail and are applied to a running example in order to make them easier to grasp. The following figure provides an overview of the patterns presented in this paper and of related patterns from literature. A short description of all these patterns is shown in the appendix.


Idioms for Error Handling in C [Preschern 2015]
SAMURAI PRINCIPLE
GUARD CLAUSE
Patterns for Light-Weight Fault Tolerance [Alho and Rauhamäki 2013]
LET IT CRASH
Patterns for Returning Data in C [Preschern 2018b]
RETURN VALUE
OUT-PARAMETERS
SPECIAL RETURN VALUE
RETURN ERROR CODES
ENCODED ERROR CODES
ERROR CODE WITH ERRNO
Portland Pattern Repository
Patterns in C [Tornhill 2014]
ASSERTION CONTEXT
LOG ERRORS
RETURN RELEVANT ERRORS
LOG UNEXPECTED ERRORS
LOG AT DISTRIBUTION BOUNDARY
Patterns for Generation, Handling and Management of Errors [Longshaw and Woods 2005]
BACKTRACE
Error Handling for Business Information Systems [Renzel 1997]
Legend
Pattern presented in this paper
Pattern presented in literature
A → B B can be used to implement or complement A

3. RELATED WORK

Unsurprisingly, there already is quite some literature out there on the topic of error handling. However, When narrowing it down to the C programming language or even to patterns for the C programming language, there is just very few literature available and that is where this paper comes in.

A comprehensive overview of error handling in general is provided by [Aglassinger 1999] who describes error handling best practices including code examples for several programming languages including C.

The Portland Pattern Repository [Portland Pattern Repository] provides many patterns and discussions on error handling as well as other topics. Most of the error handling patterns target exception handling, but also some C idioms are presented. More C idioms are presented in the book by [Tornhill 2014], which also provides other patterns for the C programming language.

[Longshaw and Woods 2004] and [Longshaw and Woods 2005] present a collection of patterns for error logging and error handling, where most of the patterns target exception-based error handling. Patterns on exception-based error handling are also presented by [Wirfs-Brock 2005]. [Renzel 1997] presents error handling patterns tailored for object-oriented business information systems; however, most of the patterns can be applied for other domains as well.

This paper gathers the most common error handling techniques of the literature mentioned above and it provides additional insights by elaborating in more detail when which of the error handling techniques can be applied and which consequences arise. By showing error handling techniques across different functions and software-modules, this paper adds to a previous paper on error handling [Preschern 2015], which only focused on error handling techniques within a function implementation.

4. PATTERNS AND RUNNING EXAMPLE

Running example:

The running example in this grey box will show how to apply the presented patterns.

You want to implement a software-module which provides functionality to store string-values for keys identified via strings. In other words you want to implement some functionality similar to the Windows registry. To keep things simple, the following code will not contain hierarchical relationships between the keys and only functions to create registry elements will be discussed.

REGISTRY API

```
/* Handle for registry keys */
typedef struct Key* RegKey;

/* Create a new registry key
   identified via the
   provided 'key_name' */
RegKey createKey(char* key_name);

/* Store the provided 'value' to
   the provided 'key' */
void storeValue(RegKey key,
                char* value);

/* Make the key available
   for being read (by other
   functions which are not
   part of this code example) */
void releaseKey(RegKey key);
```

REGISTRY IMPLEMENTATION

```
#define STRING_SIZE 100
#define MAX_KEYS 40

struct Key
{
  char key_name[STRING_SIZE];
  char key_value[STRING_SIZE];
};

/* file-global array holding all registry keys */
static struct Key* key_list[MAX_KEYS];

RegKey createKey(char* key_name)
{
  RegKey newKey = calloc(1, sizeof(struct Key));
  strcpy(newKey->key_name, key_name);
  return newKey;
}

void storeValue(RegKey key, char* value)
{
  strcpy(key->key_value, value);
}

void releaseKey(RegKey key)
{
  int i;
  for(i=0; i<MAX_KEYS; i++)
    if(key_list[i] == NULL)
      key_list[i] = key;
}
```

With the code above, you are not sure how you should provide your caller with error information in case of internal errors or, for example, in case of unterminated string-input. Your caller does not really know whether the calls succeeded or whether something failed and ends up with the following code:

CALLER CODE

```
RegKey my_key = createKey("myKey");
storeValue(my_key, "A");
releaseKey(my_key);
```

The caller's code is very short and easy to read. However, the caller does not know whether any error occurred and the caller has no possibility to react on errors. To give the caller that possibility you next want to introduce error handling in your code and you want to provide your caller with error information. The first idea that comes to your mind is to let the caller know about any errors showing up in your software-module. To do that, you RETURN ERROR CODES.

RETURN ERROR CODES

Context:
You implement a software-module which performs some error handling and you want to transport error information to your caller.

Problem:
You want to have a mechanism to transport error information to the caller, so that the caller can react on them in his code. You want the mechanism to be simple to use and the caller should be able to clearly distinguish between different error situations which could occur.

In the old days of C, error information was transported by an ERROR CODE WITH ERRNO. The global `errno` variable had to be reset by the caller, then a function had to be called, and the function indicated occurring errors by setting the global `errno` variable, which the caller had to check after the function call.

However compared to using `errno`, you rather want a way to transport error information which makes it easier for the caller to check for errors. The caller should see from the function signature at best how the error information will be transported and which kind of error information to expect.

Also, the mechanism to transport error information should be safe to use in a multi-threaded environment and only the called function should have the possibility to influence the transported error information. In other words: It should be possible to use the mechanism and still have a reentrant function.

Solution:
Use the RETURN VALUE of a function to transport error information. Return a numeric identifier which represents a specific kind of error. The caller can check the function return value against the error identifiers and can react in his code accordingly. In case the function has to return other function results, provide them to the caller in form of OUT-PARAMETERS.

Define the numeric error identifiers in your API in form of an `enum` or by using `define`. If there are many error codes or if your software-module consists of more than one headerfile, you could have a separate headerfile which just contains the error codes and which is included by your other headerfiles.

Give the error identifiers a meaningful name and document their meaning in form of comments. Make sure to name your error codes in a consistent way across your APIs.

Code Example

CALLER

```
ErrorCode status = func();
if (status == MAJOR_ERROR)
{
  /* abort program */
}
else if (status == MINOR_ERROR)
{
  /* handle error */
}
else if (status == OK)
{
  /* continue normal execution*/
}
```

CALLEE API

```
typedef enum
{
  MINOR_ERROR,
  MAJOR_ERROR,
  OK
} ErrorCode;

ErrorCode func();
```

CALLEE IMPLEMENTATION

```
ErrorCode func()
{
  if (minorErrorOccurs())
     return MINOR_ERROR;
  else if (majorErrorOccurs())
    return MAJOR_ERROR;
  else
    return OK;
}
```

Consequences:
You now have a way to transport error information which makes it very easy for the caller to check for occurring errors. Compared to `errno`, the caller does not have to set and check the error information in steps additionally to the function call, but instead the caller can directly check against the return value of the function call.

Returning error codes can safely be used in multi-threaded environments. Callers can be sure that only the called function and no other side-channels influence the returned error.

The function signature makes it very clear how the error information is transported. This is made clear for the caller and also clear for the compiler or for static code analysis tools, which can check whether the caller checked the function return value and whether the caller checked against all errors which could occur.

As the function now provides different results in different error situations, these results have to be tested. Compared to a function without any error handling, more extensive testing has to be done.

C only provides one single return value which is now used to transport error information. Thus the return value cannot be used for transporting other function results anymore. That means that other function results now have to be transported as OUT-PARAMETERS, which have the drawback that an additional parameter is required for the function and that from the function signature, it is not clear that that additional parameter is not used as function input, but is used to transport results computed by the function.

Known Uses and Related Patterns:

- Microsoft uses HRESULT to return error information. An HRESULT is a unique error code. Making the error code unique has the advantage that the error information can be transported across many functions while still making it possible to find out where the error originated. However, makeing the error code unique brings in additional effort for assigning error numbers and for keeping track of who is allowed to use which error numbers. Another specialty of HRESULT is that it encodes specific information, like for example the severity of an error, into the error code by using some bits dedicated to transport this information.
- The code of the Apache Portable Runtime defines the type `apr_status_t` to return error information. Any function which returns error information via this way returns `APR_SUCCESS` on success or any other value to indicate errors. Other values are uniquely defined error codes specified via `#define` statements.
- The openssl code defines error codes in several header files (`dsaerr.h`, `kdferr.h`, ...). As an example, the error codes `KDF_R_MISSING_PARAMETER` or `KDF_R_MISSING_SALT` inform the caller in detail about missing or wrong input parameters. The error codes in each of the files are just defined for a specific set of functions which belong to that file and the error code values are not unique across the whole openssl code.
- Having an ERROR CODE is described in the Portland Pattern Repository as a pattern sketch, which also describes the idea of returning error information by explicitly using the function's return value.

Running example:

Now you provide your caller with information in case of errors in your code. In your code you check for things that could go wrong and you provide that information to the caller.

REGISTRY API

```
/* Error codes returned
   by this registry */
typedef enum
{
  OK,
  OUT_OF_MEMORY,
  INVALID_KEY,
  INVALID_STRING,
  STRING_TOO_LONG,
  CANNOT_ADD_KEY
}RegError;

/* Handle for registry keys */
typedef struct Key* RegKey;

/* Create a new registry key identi-
   fied via the provided 'key_name'.
   Returns OK in case no problem
   occurs, INVALID_KEY if
   the 'key' parameter is NULL,
   INVALID_STRING if 'key_name'
   is NULL, STRING_TOO_LONG if
   'key_name' is too long, or
   OUT_OF_MEMORY if no memory
   resources are available. */
RegError createKey(char* key_name,
                   RegKey* key);

/* Store the provided 'value' to
   the provided 'key'.
   Returns OK in case no problem
   occurs, INVALID_KEY if
   the 'key' parameter is NULL,
   INVALID_STRING if 'value' is
   NULL, or STRING_TOO_LONG
   if 'value' is too long. */
RegError storeValue(RegKey key,
                    char* value);

/* Make the key available
   for being read. Returns OK if
   no problem occurs,
   INVALID_KEY if 'key'
   is NULL, or CANNOT_ADD_KEY
   if the registry is full and no
   more keys can be released. */
RegError releaseKey(RegKey key);
```

REGISTRY IMPLEMENTATION

```
#define STRING_SIZE 100
#define MAX_KEYS 40

struct Key
{
  char key_name[STRING_SIZE];
  char key_value[STRING_SIZE];
};

/* file-global array holding all registry keys */
static struct Key* key_list[MAX_KEYS];

RegError createKey(char* key_name, RegKey* key)
{
  if(key == NULL)
    return INVALID_KEY;

  if(key_name == NULL)
    return INVALID_STRING;

  if(STRING_SIZE <= strlen(key_name))
    return STRING_TOO_LONG;

  RegKey newKey = calloc(1, sizeof(struct Key));
  if(newKey == NULL)
    return OUT_OF_MEMORY;

  strcpy(newKey->key_name, key_name);
  *key = newKey;
  return OK;
}

RegError storeValue(RegKey key, char* value)
{
  if(key == NULL)
    return INVALID_KEY;

  if(value == NULL)
    return INVALID_STRING;

  if(STRING_SIZE <= strlen(value))
    return STRING_TOO_LONG;

  strcpy(key->key_value, value);
  return OK;
}
```

```
RegError releaseKey(RegKey key)
{
  int i;
  if(key == NULL)
    return INVALID_KEY;

  for(i=0; i<MAX_KEYS; i++)
    if(key_list[i] == NULL)
    {
      key_list[i] = key;
      return OK;
    }

  return CANNOT_ADD_KEY;
}
```

Now the caller can react on the provided error information and can, for example, provide the user of the application with detailed information about what went wrong.

CALLER CODE

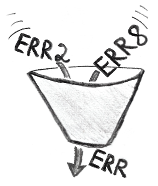
```
RegError err;
RegKey my_key;

err = createKey("myKey", &my_key);
if(err == INVALID_KEY || err == INVALID_STRING)
  printf("Internal application error\n");
if(err == STRING_TOO_LONG)
  printf("Provided registry key name too long\n");
if(err == OUT_OF_MEMORY)
  printf("Insufficient resources to create key\n");

err = storeValue(my_key, "A");
if(err == INVALID_KEY || err == INVALID_STRING)
  printf("Internal application error\n");
if(err == STRING_TOO_LONG)
  printf("Provided registry value to long to be stored to this key\n");

err = releaseKey(my_key);
if(err == INVALID_KEY)
  printf("Internal application error\n");
if(err == CANNOT_ADD_KEY)
  printf("Key cannot be relased, because the registry is full\n");
```

Now the caller can react on errors; however, note that the code for the registry software-module as well as the code for the caller more than doubled in size. Error handling did not come for free. A lot of effort was put into implementing error handling. That can also be seen in the registry API. The comments for the functions became a lot longer, because they have to describe which error situations can occur. Also the caller has to put a lot of effort into thinking about what to do if a specific error occurs.

With providing such detailed error information to the caller, you burden the caller with reacting on these errors and with thinking about which errors are relevant to handle and which are irrelevant to the caller. Thus, special care as to be taken to on the one hand, provide the caller with the necessary error information, but to also on the other hand not flood the caller with information he does not need.

Next you want to make these considerations in your code and you only want to provide error information which is actually useful to the caller. Thus, you only RETURN RELEVANT ERRORS.

RETURN RELEVANT ERRORS



Context:
You implement a software-module which performs some error handling and you want to transport error information to your caller.

Problem:
On the one hand, the caller should have the possibility to react on errors; however, on the other hand the more error information you return, the more your code and the code of your caller has to deal with error handling, which makes the code longer. Longer code is harder to read and maintain and longer code brings in the risk of additional bugs.

In order to transport error information to your caller, detecting the error and returning the information is not the only task you have to do. Additionally you have to document in your API which errors are returned. If you don't do that, then your caller would not know that he has to expect and handle these errors. Documenting error behavior is work that has to be done. The more errors there are, the more documentation work has to be done.

Returning very detailed, implementation-specific error information and adding additional error information later on in your code if the implementation changes implies that with such an implementation change you have to semantically change your interface which documents the returned error information. Such changes might not be desirable for your existing callers, because they would have to adapt their code to additionally react on the newly introduced error information.

Providing more detailed error information is also not always a good thing for the caller ether. Each error information transported to the caller means additional work for the caller. The caller has to decide whether the error information is relevant to him and how to handle it.

Solution:
Only transport error information to the caller, if that information is relevant to the caller. Error information is only relevant to the caller, if the caller can react on that information in his code. If the caller cannot react on the error information in his code, then it would be unnecessary to provide the caller to opportunity (or the burden) to react on the error.

If the only reason for transporting the error to the caller is that the caller can then also transport this error information to his callers, who then log this error information or display it to the user, then why would not the software-module where the error occurred directly LOG ERRORS like these?

There are several way how to only return relevant error information. One way is to simply not return any error information at all. For example, when having some function `cleanupMemory(void* handle)` which cleans up memory for the provided HANDLE, then there is no need to transport information whether the cleanup succeeded, because the caller can not react in the code and such a cleanup error (retrying to call a cleanup function is in most cases no solution). Thus the function can simply not return any error information. To make sure that errors within the function do not go unnoticed, it might even be an option to LET IT CRASH in case an error occurs.

If you already RETURN ERROR CODES, then only the error information which is relevant to the caller should be transported. Other errors which occur can be summarized as one internal error code and if the detailed error information is needed for debugging purposes, you could LOG ERRORS. If you realize that there are not many error situations after only returning relevant errors, then instead of error codes, it might a better solution to simply have SPECIAL RETURN VALUES to transport the error information.

Code Example

CALLER

```
ErrorCode status = func();
if (status == MAJOR_ERROR ||
    status == UNKNOWN_ERROR)
{
  /* abort program */
}
else if (status == MINOR_ERROR)
{
  /* handle error */
}
else if (status == OK)
{
  /* continue normal execution*/
}
```

CALLEE API

```
typedef enum
{
  MINOR_ERROR,
  MAJOR_ERROR,
  UNKNOWN_ERROR,
  OK
} ErrorCode;

ErrorCode func();
```

CALLEE IMPLEMENTATION

```
ErrorCode func()
{
  if (minorErrorOccurs())
    return MINOR_ERROR;
  else if (majorErrorOccurs())
    return MAJOR_ERROR;
  else if(internalError1Occurs() ||
          internalError2Occurs())
    return UNKNOWN_ERROR;
  else
    return OK;
}
```

In the code above, the same error information is returned in case `internalError1Occurs` and in case `internalError2Occurs`, because it is irrelevant for the caller which of the two implementation-specific errors occurs. The caller would react on both errors in the same way (in the example above: aborting the program).

Consequences:
Not returning detailed information about which kind of internal errors occurred is a relief for the caller. The caller is not burdened with thinking about how to handle all possible internal errors which occur and it is more likely that the caller does actually react on all different kind of errors which are transported to him, because all of the transported errors

make sense to him. Also testers can be happy, because now that less error information is returned by the functions, less error situations have to be tested.

In case the caller uses very strict compilers or static code analysis tools which verify whether the caller does check for all possible return values, the caller does not have to simply explicitly handle errors he is not interested in (e.g. a switch statement will many fallthroughs and one error handling for all internal errors). Instead, the caller only handles one internal error code or in case that you LET IT CRASH, the caller does not have to handle any of such errors.

Not returning the detailed error information makes it impossible to the caller to show this error information to the user or to save this error information for debugging purposes for the developer. However, for such debugging information it would be better to LOG ERRORS directly in the software-module where they occur and not burden the caller with doing that.

If you don't transport all information about errors occurring in your function, but instead you only transport information which you think is relevant to the caller, then there is the chance that you get it wrong. You might forget some information which is necessary for the caller and maybe that leads to a change request for adding this information. If you RETURN ERROR CODES, additional error codes can easily be added to your function, but when using SPECIAL RETURN VALUES, it might not be so easy to add error information later on.

Known Uses and Related Patterns:

- The function `FlushWinFile` of the game Nethack flushes a file to the disk calling the Macintosh function `FSWrite` which does return error codes. However, the Nethack wrapper explicitly ignores the error code and `FlushWinFile` is of return type `void`, because the code using that function cannot react accordingly in case an error occurs. Thus, the error information is not passed along.
- The openssl function `EVP_CIPHER_do_all` initializes several cipher suites by calling the internal function `OPENSSL_init_crypto`, which does return error information. However, this error information is ignored by the `EVP_CIPHER_do_all` function which is of return type `void`.
- For security-relevant code it is very common to only return relevant information in case of errors. For example, if a function to authenticate a user returns detailed information whether authentication did not work, because the username is invalid, or because the password is invalid, then the caller could use this function for checking which usernames are already taken. To avoid opening such information side-channels, it is common to only transport the binary information whether authentication worked or not. For example, the function `rbacAuthenticateUserPassword` used to authenticate users in the B&R Automation Runtime operating system has the return type `bool` and returns `true` in case the authentication worked or returns `false` in case it did not work. No detailed information about why the authentication did not work is returned.

Running Example:

When you only RETURN RELEVANT ERRORS, your registry code looks like the following. To keep things simple, only the `createKey` is shown:

REGISTRY API

```
/* Create a new registry key
   identified via the provided
   'key_name'. Stores a handle to
   the key in the provided 'key'
   parameter.
   Returns OK on success,
   STRING_TOO_LONG if the
   provided 'key_name' is too long,
   OUT_OF_MEMORY in case of
   insufficient memory, or
   INVALID_PARAMETER if any
   of the provided parameters
   is not valid. */
RegError createKey(char* key_name,
                   RegKey* key);
```

REGISTRY IMPLEMENTATION

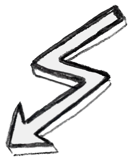
```
RegError createKey(char* key_name, RegKey* key)
{
  if(key == NULL || key_name == NULL)
    return INVALID_PARAMETER;

  if(STRING_SIZE <= strlen(key_name))
    return STRING_TOO_LONG;

  RegKey newKey = calloc(1, sizeof(struct Key));
  if(newKey == NULL)
    return OUT_OF_MEMORY;

  strcpy(newKey->key_name, key_name);
  *key = newKey;
  return OK;
}
```

Instead of returning `INVALID_KEY` or `INVALID_STRING`, now the function returns `INVALID_PARAMETER` for all these error cases. Now the caller cannot handle the two error situations differently which also means the caller does not have to think about how to handle the two error situations differently. The caller code becomes more simple, because now there is one error situation less to be handled.

That is good, because what would the caller do in case the function returns `INVALID_KEY` or `INVALID_STRING`? It wouldn't make any sense for the caller to try calling the function again. In both cases the caller could just accept that calling the function did not work and the caller could report that to the user or he could abort the program. As there would be no reason for the caller to react differently on the two errors, you now took the caller the burden to think about two different error situations. Now the caller only has to think about one error situation and has to react accordingly.

Next, you question yourself whether it is even necessary that the caller has to decide how to react on such an `INTERNAL_ERROR` and you realize that you can take the decision from the caller and LET IT CRASH.

Let It Crash



Context:
You have a function in which major errors can occur. Perhaps currently you already Return Error Codes and perhaps you only Return Relevant Errors to your caller.

Problem:
When transporting error information via the Return Value to your caller, you cannot be sure that the caller actually handles this error. In C it is not mandatory to check return values of the called functions and your caller can simply ignore the return value of a function. In case the error which occurs in your function is major, sometimes you don't want your caller to decide whether the error should be handled. Instead you'd want to make sure that definitely an action is taken in case the error occurs.

You don't want to have additional parameters for your function only to transport error information. When Returning Error Codes you have the drawback that you have to use the Return Value of the function to transport error information and you have to add additional Out-Parameters to transport the actual function results. In some cases, instead, you could transport error information via Special Return Values and you therefore wouldn't need Out-Parameters; however, that is not always possible.

The errors in your code might only occur very rarely. Handling such rarely occurring situations makes the caller code less readable, because it distracts from the actual purpose of the caller code. The caller might have to write many lines of code to handle a very rarely occurring situation.

If the caller handles some error situation, quite often the program will still crash or some error will still occur. The error might simply show up somewhere else - maybe somewhere in the caller's caller code who does not handle error situations properly. In such a case, handling the error disguises the error and that makes it much harder to debug the error in order to find out the root cause.

Solution:
Don't use any additional parameters or the return value of your function to transport error information. In case an error in occurs, simply let the program crash. At best, abort the program in a structured way, e.g. by using the `assert` statement. However, also aborting the program in a less structured way, e.g. by not checking for NULL pointers and by accessing the pointer, would be OK. Simply make sure that the program crashes at the point where the error occurs.

Of course, this approach is not appropriate for any kind of errors and not appropriate for any kind of application domains. You wouldn't want to let the program crash in case of some unexpected user input. However, in case of a programming error, it can be appropriate to let the program crash in order to make it as simple as possible for the programmers to find the error. To make it even easier to find the error, you can Log Errors.

Quite often a function has GUARD CLAUSES, which check for valid pre-conditions (valid parameters) at the beginning of the function. Some of these parameter validity checks are very good candidates to LET IT CRASH.

The caller has to be well aware of the behavior of your function, so you have to document in the functions API in which cases the function aborts the program. For example, the function documentation has to state whether the program crashes in case the function is provided a NULL pointer as parameter.

Code Example

CALLER

```
func();
/* continue - no error
   handling required */
```

CALLEE

```
void func()
{
  if(severeErrorOccurs())
  {
     assert(false);
  }
}
```

Consequences:
Errors can be directly handled in the callee code. The caller does not have to cope with handling the error and thus the caller code becomes much simpler. Also, it is not possible that the error goes unnoticed, because it is not up to the caller to check some return value for error information. Instead the callee code makes sure that the error is handled.

The caller is not burdened with the decision on how to handle the error; however, that also means that the caller has no saying on how the program reacts in case such an error occurs. In case of such an error, the program simply crashes. In some cases that is good, because sometimes a crash is better that unpredictable behavior, but in some other cases that might not be appropriate. For example, for safety-critical applications it might be necessary to implement fault-tolerance mechanisms and simply aborting the program is not an option at all.

If the error occurs and the program directly crashes at that point, then the programmer is provided with very good debug information. When you LOG ERRORS or when you are using the `assert` statement with an ASSERTION CONTEXT, then you'll get very detailed information regarding where in the code an error occurred.

Known Uses and Related Patterns:

- The SAMURAI PRINCIPLE [Preschern 2015] suggests to use the `assert` statement in C on order to make erorr handling within the implementation of a function easier.
- A similar pattern with the name LET IT CRASH is presented by [Alho and Rauhamäki 2013]. The pattern targets distributed control systems and suggests to let single fail-safe processes crash and to let them restart quickly.
- [Candea and Fox 2003] describe that Internet applications should fail fast and recovers quickly in order to provide more reliable services.
- The C stdlib function `strcpy` does not check for valid user input. If you provide the function with a NULL pointer, it crashes.

Running Example:

Now that you intentionally LET IT CRASH in case of some internal errors, your `createKey` function looks like the following:

REGISTRY API

```
/* Create a new registry key
   identified via the provided
   'key_name' (must not be NULL,
   max. STRING_SIZE characters).
   Stores a handle to the key in the
   provided 'key' parameter (must
   not be NULL).
   Returns OK on success, or
   OUT_OF_MEMORY in case of
   insufficient memory. */
RegError createKey(char* key_name,
                   RegKey* key);
```

REGISTRY IMPLEMENTATION

```
RegError createKey(char* key_name, RegKey* key)
{
  assert(key != NULL && key_name != NULL);
  assert(STRING_SIZE > strlen(key_name));

  RegKey newKey = calloc(1, sizeof(struct Key));
  if(newKey == NULL)
    return OUT_OF_MEMORY;

  strcpy(newKey->key_name, key_name);
  *key = newKey;
  return OK;
}
```

Instead of returning an `INVALID_PARAMETER` or `STRING_TOO_LONG`, now the function aborts the program in case one of the provided parameters is not what you expect them to be. Aborting in case of too long strings at first seems a bit drastic. However, similar to NULL pointers, a too long string is invalid input for your function. In case your registry does not get its string input from a user via a GUI, but instead gets a fixed input from the caller's code, then also for too long strings this code only aborts in case of programming errors which is perfectly fine behavior.

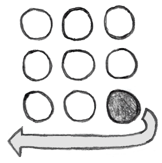
The caller does not have to think about how to handle these errors. The function implementation takes away that burden from the caller and therefore the caller's code becomes more simple. However, also the caller cannot influence the error handling behavior of the function and the caller cannot gracefully handle such errors of the function. For some applications that is ok; however, for some other applications failure might not be an option and such errors would have to be handled in a fault tolerant way.

Anyways, for your registry you know that the caller can only make a useful application in case he calls the registry function with valid parameters - calling it with invalid parameters is a programming error and you decide that such errors rather have to be fixed and not masqueraded. Thus the above code aborts the program in case of invalid parameters and it does that by explicitly using `assert`.

An alternative approach, which makes your code shorter, would be to build on the fact that accessing a NULL pointer also aborts the program and that the function `strcpy` also does not check for NULL pointers. So an alternative would be to skip the first assert statement in in the code. The code would still have the same behavior and it would be shorter; however, it would have the disadvantage that it would not explicitly show anymore that you do want it to fail in case of NULL pointers. Somebody reading such code could assume that the programmer simply forgot about handling these error situations.

Sticking with the code in the code sample above, next, you realize that the `createKey` function only returns two different error codes: OUT_OF_MEMORY and OK. Your code can be made much more beautiful by simply transporting this kind of error information with SPECIAL RETURN VALUES.

SPECIAL RETURN VALUES



Context:
You have a function which computes some result and you want to transport error information to your caller if an error occurs when executing the function. You only want to transport RELEVANT ERROR INFORMATION.

Problem:
It is not an option to you to explicitly RETURN ERROR CODES to transport error information, because that implies that you cannot use the RETURN VALUE of the function to return other data and you'd have to transport that data via OUT-PARAMETERS which would make calling your function more difficult. However, you want it to be very simple to call your function.

Returning no error information at all is also not an option to you. You want to provide your caller with some error information and you want your caller to be able to react on these errors in his code. There is not a lot of error information which you want to transport to your caller. It might just be the binary information whether the function call worked or whether it did not work. To RETURN ERROR CODES for such simple information would be an overkill.

You cannot LET IT CRASH, because the errors occurring in your function are not severe or because you want to make it possible to the caller to decide how the errors should be handled, because maybe the caller can handle the errors gracefully.

Solution:
Use the RETURN VALUE of your function to transport the data computed by the function. Reserve one ore more special values to be returned in case an error occurs.

If, for example, your function returns a pointer, then you could use as reserved special value the NULL pointer to indicate that some error occurred. The NULL pointer is by definition no valid pointer, so you can be sure that this special value is not confused with a valid pointer calculated by your function as a result.

You have to make sure to document in the API which returned special value has which meaning. In some cases a common convention settles which special values indicate errors. For example, UNIX functions usually indicate an error by returning negative integer values. Still, even in such cases the meaning of the specific return values have to be documented.

You have to make sure that the special value which indicates error information really is a value which cannot occur in case of no error. For example, if a function returns a temperature value in degree Celsius as an integer value, then it would not be a good idea to stay with the UNIX convention where any negative value indicates an error. Instead, it would be better to use, for example, the value -300 to indicate an error, because it is physically impossible that a temperature takes a value below -273 degree Celsius.

Code Example

CALLER

```
void* pointer = func();
if (pointer != NULL)
{
  /* continue */
}
else
{
  /* handle error */
}
```

CALLEE

```
void* func()
{
  if(somethingGoesWrong())
  {
     return NULL;
  }
  else
  {
    return some_pointer;
  }
}
```

Consequences:
The function can now return error information via the RETURN VALUE even though the RETURN VALUE is used to transport the computation result of the function. No additional OUT-PARAMETERS have to be used only for having a way to transport error information.

Sometimes you don't have many special values to encode error information. For example, for pointers there is only the NULL pointer to indicate error information. That leads to the situation that it is only possible to indicate to the caller whether everything worked well or whether anything went wrong. This has the drawback that you cannot transport detailed error information. However, this also has the benefit that you are not tempted to transport unnecessary error information. In many cases it is sufficient to only transport the information that anything went wrong and the caller cannot react on more detailed information anyway.

If, at a later point in time, you realize that you have to transport more detailed error information, then perhaps that is not possible anymore because you have no more unused special values left. You'd have to change the whole function signature and instead RETURN ERROR CODES to transport that additional error information. Changing the function signature might not always be an option, because your API might have to stay compatible for existing callers. In such cases, it might be better to RETURN ERROR CODES right at the beginning in order to be able to react on such changes while maintaining compatibility.

Sometimes programmers assume that it is clear which returned values indicate errors. For example, to some programmers it might be clear that a NULL pointer indicates an error. For some other programmers it might be clear that -1 indicates an error. This brings in the dangerous situation that the programmers assume that it is clear to everybody which values indicate errors. However, these are just assumptions. In any case it should be well-documented in the API which values indicate errors, but sometimes programmers forget to do that wrongly assuming that that is absolutely clear.

Known Uses and Related Patterns:

- The C stdlib function `fopen` returns a `FILE` handle in case no error occurs. In case of an error, like for example, if you don't have the permission to open the file, the function returns NULL. Additionally, the function sets the `errno` variable to transport more specific information about the error.
- The `getobj` function of the game Nethack returns the pointer to some object in case no error occurs and returns NULL in case an error occurs. To indicate the special case that there is no object to return, the function returns the pointer to a global object called `zeroobj` which is a object of the return type defined for the function and which is also known to the caller. The caller can then check whether the returned pointer is the same as the pointer to the global object and can thus distinguish between a pointer to any valid object and a pointer to the `zeroobj` which carries some special meaning.
- The C stdlib function `getchar` reads a character from `stdin`. The function has return type `int` which allows transporting much more information than simple characters. In case no more characters are available, the function returns `EOF` which is usually defined as -1. As characters cannot take negative integer representations, `EOF` can clearly be distinguished from regular function results and can thus be used to indicate the special situation when no more characters are available.
- Most UNIX or POSIX function use negative numbers to transport error information. For example, the POSIX function `write` returns the number of written bytes or -1 on error.

Running Example:

With SPECIAL RETURN VALUES, your code looks like the following. To keep it simply, only the `createKey` function is shown.

REGISTRY API

```
/* Create a new registry key
   identified via the provided
   'key_name' (must not be NULL,
   max. STRING_SIZE characters).
   Returns a handle to the key or
   NULL on error. */
RegKey createKey(char* key_name);
```

REGISTRY IMPLEMENTATION

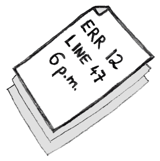
```
RegKey createKey(char* key_name)
{
  assert(key_name != NULL);
  assert(STRING_SIZE > strlen(key_name));

  RegKey newKey = calloc(1, sizeof(struct Key));
  if(newKey == NULL)
    return NULL;

  strcpy(newKey->key_name, key_name);
  return newKey;
}
```

The `createKey` function became much more simple. It does not RETURN ERROR CODES anymore, but instead it directly returns the handle and no OUT-PARAMETER is needed to transport this information. Because of that, also the API documentation for the function became much more simple, because there is no need to describe the additional parameter and there is no need to lengthly describe how the function result will be transported to the caller.

Things also became much more simple for your caller. The caller does not have to provide a handle as an OUT-PARAMETER anymore, but instead the caller directly retrieves this handle via the RETURN VALUE, which makes the caller's code a lot more readable and thus easier to maintain.

However, now you have the problem, that compared to the detailed error information which you can transport if you RETURN ERROR CODES, now the only error information which comes out of the function is whether it worked or whether it did not work. The internal details about the error are thrown away and if you need these details later on, for example as debug information, there is no way to get it. To address that issue, you can LOG ERRORS.

LOG ERRORS



Context:
You have a function in which you handle errors. You only want to transport RELEVANT ERROR INFORMATION to your caller for reacting on it in the code; however, you want to keep detailed error information for later debugging.

Problem:
You want to make sure that in case of an error, it is possible for the programmer to later on find out the cause of that error. One way would be to transport very detailed error information to the caller, also for example error information indicating programming errors. To do that you can RETURN ERROR CODES to the caller who then displays the detailed error codes to the user. The user might get back to you (e.g. via some service hotline) to ask what that error code means and how to fix the problem. Then you'd have your detailed error information to debug the code and you could figure out what went wrong.

However, such an approach has the major drawback, that the caller, who does not care at all about that error information, has to transport the error information to the user only for the sake of having a way that this error information is provided to you. Also the user does not really care about such detailed error information.

In addition, to RETURN ERROR CODES has the drawback that you have to use the RETURN VALUE of the function to transport error information and you have to use additional OUT-PARAMETERS to transport the actual function results. In some cases, instead, you can transport error information via SPECIAL RETURN VALUES; however, that is not always possible. You don't want to have additional parameters for your function only to transport error information, because it makes your caller's code more complicated.

Solution:
Use different channels to transport error information which is relevant for the calling code and error information which is relevant for the developer. For example, write debug error information into a log file and don't return the detailed debug error information to the caller. In case an error occurs, the user of the program has to provide you with the logged debug information. For example, the user has to send you a log file via e-mail.

Alternatively, you could, at the interface between you and your caller, log the error (LOG AT DISTRIBUTION BOUNDARIES) and additionally RETURN RELEVANT INFORMATION to the caller. For example, the caller could be informed that some internal error occurred, but the caller does not see which detailed kind of error occurred. Thus, the caller could still handle the error in the code without requiring knowledge on how to handle very detailed errors and you'd while still not be losing valuable debug information.

For such valuable debug information, you should log information about programming errors and you should LOG UNEXPECTED ERRORS. For such errors it is valuable to store information about where the error occurred, for example, the source code file name and the line number, or the BACKTRACE.

Code Example

CALLER

```
func();
/* continue - no error
    handling required */
```

CALLEE

```
void func()
{
  if(somethingGoesWrong)
  {
     logInFile("something went wrong");
  }
}
```

Consequences:
You can obtain debug information without requiring your caller to handle or to transport this information. That makes life for the caller a lot easier, because the caller does not have to handle or transport the detailed error information which he is not at all interested in. Instead, you transport the detailed error information yourself.

Maybe in some cases, you just want to log some error or situation which occurred, but which is completely irrelevant to the caller. Thus, you don't even have to transport any error information to the caller. For example, if you LET IT CRASH in case the error occurs, the caller does not at all have to react on the error and still you can make sure to not lose valuable debug information if in such a case, you LOG ERRORS. So there are no additional parameters to your function required in order to transport error information and that makes calling your function a lot easier and it helps the caller to keep his code clean.

Still, you don't lose this valuable error information and can still use it for debug purposes, for example, to hunt down programming errors. To not loose this debug information, you transport it via a different channel - e.g. via log files. However, you have to think about how to get to these log files. You could ask the uses to send you the logfile via e-mail or, more advanced, you could implement some automatic bug report mechanism. However, with both of these approaches you cannot be 100% sure that the log information really gets back to you. If the user does not want that, he could prevent it.

Known Uses and Related Patterns:

- The Apache webserver code uses the function `ap_log_error` write errors related to requests or connections to an error log. Such a log entry contains information about the filename and line of code where the error occurred and it contains a custom string provided to the function by the caller. The log information is stored in a `error_log` file on the server.
- The B&R Automation Runtime operating system provides a logging system which allows programmers to transport logging information to the user via calling the function EVENTLOGWRITE from anywhere in the code. This makes it possible to transport information to the user without having to transport this information across the whole calling stack up to some central logging component.
- ASSERTION CONTEXT suggests to not only LET IT CRASH, but to additionally log information about the reason or about the position of the crash by adding a string statement inside the `assert` call. If the `assert` fails, then the link of code containing the assert statement will be printed and that then includes the added string.

Running example:

After applying the patterns, you'll get to the following final code for your registry software-module. This code provides the caller with relevant error information, but does not require the caller to handle errors about implementation details he is not interested in.

REGISTRY API

```
/* max. size of string parameters
   (including NULL-termination) */
#define STRING_SIZE 100

/* Error codes returned
   by this registry */
typedef enum
{
  OK,
  CANNOT_ADD_KEY
}RegError;

/* Handle for registry keys */
typedef struct Key* RegKey;

/* Create a new registry key
   identified via the provided
   'key_name' (must not be NULL,
   max. STRING_SIZE characters).
   Returns a handle to the key or
   NULL on error. */
RegKey createKey(char* key_name);

/* Store the provided 'value'
   (must not be NULL, max.
   STRING_SIZE characters) to
   the 'key' (MUST NOT BE NULL) */
void storeValue(RegKey key,
                char* value);

/* Make the 'key' (must not be
   NULL) available for being read.
   Returns OK if no problem occurs
   or CANNOT_ADD_KEY if the
   registry is full and no more
   keys can be released. */
RegError releaseKey(RegKey key);
```

REGISTRY IMPLEMENTATION

```
#define MAX_KEYS 40

struct Key
{
  char key_name[STRING_SIZE];
  char key_value[STRING_SIZE];
};

/* macro to log debug info and to assert */
#define logAssert(X)                          \
    if(!X)                                    \
    {                                         \
      printf("Error at line \%i", __LINE__) \
      assert(false);                          \
    }

/* file-global array holding all registry keys */
static struct Key* key_list[MAX_KEYS];

RegKey createKey(char* key_name)
{
  logAssert(key_name != NULL)
  logAssert(STRING_SIZE > strlen(key_name))

  RegKey newKey = calloc(1, sizeof(struct Key));
  if(newKey == NULL)
    return NULL;

  strcpy(newKey->key_name, key_name);
  return newKey;
}

void storeValue(RegKey key, char* value)
{
  logAssert(key != NULL && value != NULL)
  logAssert(STRING_SIZE > strlen(value))

  strcpy(key->key_value, value);
}

RegError releaseKey(RegKey key)
{
  logAssert(key != NULL)

  int i;
  for(i=0; i<MAX_KEYS; i++)
    if(key_list[i] == NULL)
    {
      key_list[i] = key;
      return OK;
    }

  return CANNOT_ADD_KEY;
}
```

The code above is shorter compared to the earlier code of the running example, because:

- The code does not check for programming errors but aborts the program in case of programming errors. Invalid parameters like NULL pointers are not gracefully handled in the code, but instead the API documents that the handles must not be NULL.
- The code only returns errors which are relevant for the caller. For example, the `createKey` function does not RETURN ERROR CODES, but instead simply returns a handle and NULL in case of error, because the caller does not need more detailed error information.

While to code is shorter, the API comments grew. The comments now specify more clearly how the functions behave in case of errors. That helped that apart form your code, also the caller's code became simpler, because now the caller is not burdened anymore with that many decisions on how to react on different kinds of error information.

CALLER CODE

```
RegKey my_key = createKey("myKey");
if(my_key == NULL)
  printf("Cannot create key\n");

storeValue(my_key, "A");

RegError err = releaseKey(my_key);
if(err == CANNOT_ADD_KEY)
  printf("Key cannot be relased, because the registry is full\n");
```

The code above is shorter compared to the earlier code of the running example, because:

- The return value of functions which abort in case of error do not have to be checked
- Functions where no detailed error information is required directly return the requested item. For example, `createKey()` now returns a handle and the caller does not have to provide an OUT-PARAMETER anymore.
- Error codes which indicate a programming error, like for example an invalid provided parameter, are not returned anymore and thus do not have to be checked by the caller anymore.

The final code of the running example showed, that it is important to think about which kind of error should be handled in the code and how these errors should be handled. Simply returning all kind of errors and requiring the caller to cope with all these errors is not always the best solution, because maybe the caller is not interested with that detailed error information or maybe the caller does not want to react on the error in the application. Maybe the error is severe enough so that already at the point where the error occurs it can be decided to abort the program. Such measures make the code simpler and have to be considered when designing the API of a software-component.

5. CONCLUSION

This paper showed in form of patterns how a C programmer can implement error handling with the focus on transporting error information across software-module boundaries. The ERROR CODE pattern suggests to use the return value of functions to return error information. RETURN RELEVANT ERRORS suggests to only return error information in case the programmer can react on it in his code and SPECIAL RETURN VALUE and LET IT CRASH provide ways how to do that. LOG ERRORS suggests to have a separate channel for transporting debug information to the programmer.

This paper is part of a series of papers on C programming [Preschern 2015][Preschern 2016][Preschern 2017][Preschern 2018a][Preschern 2018b]. 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...

REFERENCES

Thomas Aglassinger. 1999. *Error Handling in Structured and Object-Oriented Programming Languages*. Master's thesis. University of Oulu.

Pekka Alho and Jari Rauhamäki. 2013. Patterns for Light-Weight Fault Tolerance and Decoupled Design in Distributed Control Systems. In *Proceedings of VikingPLoP 2013 Conference*.

George Candea and Armando Fox. 2003. Crash-Only Software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*.

Andy Longshaw and Eoin Woods. 2004. Patterns for Generation, Handling and Management of Errors. In *Proceedings of the 9th European Conference on Pattern Languages of Programs (EuroPLoP)*.

Andy Longshaw and Eoin Woods. 2005. More Patterns for the Generation, Handling and Management of Errors. In *Proceedings of the 10th European Conference on Pattern Languages of Programs (EuroPLoP)*.

Doug Moen. 1992. A Discipline of Error Handling. In *Proceedings of the Summer 1992 USENIX Conference*.

Portland Pattern Repository. Portland Pattern Repository. http://c2.com/cgi/wiki. (????).

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. 2018a. C Patterns on Objects and their Lifetime. In *Proceedings of the 23rd European Conference on Pattern Languages of Programming (EuroPLoP)*.

Christopher Preschern. 2018b. Patterns for Returning Data from C Functions. In *Proceedings of the 23rd European Conference on Pattern Languages of Programming (EuroPLoP)*.

Klaus Renzel. 1997. Error Handling for Business Information Systems. http://www.objectarchitects.de/arcus/cookbook/exhandling/. (1997).

Axel-Tobias Schreiner. 1993. *Object oriented programming with ANSI-C*. Dept. of Computer Science (GCCIS)–E-prints.

Adam Tornhill. 2014. *Patterns in C*. Leanpub.

Rebecca J. Wirfs-Brock. 2005. Toward Exception-Handling Best Practices and Patterns. *IEEE Software* 23, 5 (2005), 11–13.

APPENDIX - PATLETS

Pattern Name	**Pattern Solution Sketch**
RETURN ERROR CODES	Use the return value of a function to return error information to the caller. Return the actual result of the function via OUT-PARAMETERS.
SPECIAL RETURN VALUE	Use the return value of a function to return the result of a function. Return a special reserved value in case an error occurs. E.g. when returning a pointer, NULL indicates an error.
LET IT CRASH	For critical errors, do not leave the caller the decision how to handle the error. Instead directly abort the program [Alho and Rauhamäki 2013].
LOG ERRORS	Provide the programmer with detailed error information in a log instead of returning all this information to the calling code.
RETURN RELEVANT ERRORS	Only return error information on which the caller can react on this information in his code.
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 [Preschern 2018b].
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 [Preschern 2018b].
ASSERTION CONTEXT	Add information about the context of a assertion Add this information by placing fixed string with this information in the assertion. Place the string inside a condition which always evaluates to "true" [Tornhill 2014].
SAMURAI PRINCIPLE	Return from a function victorious or not at all. If there is a situation for which you know that an error cannot be handled, abort the program [Preschern 2015].
GUARD CLAUSE	Check whether your function has pre-conditions and immediately return if these pre-conditions are not met. For example, check for the validity of input data. [Preschern 2015].
LOG UNEXPECTED ERRORS	Error conditions that are expected to arise in the course of normal domain processing should not be logged but handled in the code or by the user. Hence, any logged error should be viewed as requiring investigation [Longshaw and Woods 2005].
LOG AT DISTRIBUTION BOUNDARY	When technical errors occur, log them on the system where they occur passing a simpler generic SystemError back to the caller for reporting at the end-user interface [Longshaw and Woods 2005].
BACKTRACE	Once an error occurs, log a stacktrace, because the state of the function where the error occurred yields important information [Renzel 1997].
ENCODED ERROR CODES	The actual error is encoded in the return code. The return code is a value from an enumeration [Portland Pattern Repository].
ERROR CODE WITH ERRNO	Returns a value indicating an error, and returns the actual error information out of band through a global or thread-local variable, typically called errno [Portland Pattern Repository].