



I/O Error Handling in ROOT

Summer Student Project Report

Raveena Dandona
CERN Summer Student
17.06.19 - 16.08.19

Supervisor: Jakob Blomer
EP-SFT



CONTENTS

1. Abstract	2
2. Error Handling	2
2.1. Return Error Codes	2
2.2. Exceptions	3
2.3. Comparing Error Code & Exception Handling	4
3. The Code	4
3.1. Example: reading a file	4
3.2. class RStatus	5
3.3. class RExceptions	5
3.4. Tests	5
4. RStatus Class Implementation	6
5. Difficulties Faced	7
6. Looking Ahead	8
7. Conclusion	8
8. References	9
9. Acknowledgement	9

1. ABSTRACT

ROOT is a software framework for data analysis and I/O; a powerful tool to cope with the demanding tasks typical of the state of the scientific data analysis. The aspect of ROOT dealt with in this project is the error handling interface, specifically in the I/O components. In case of errors some users prefer to get notified by error return codes while in other contexts it is preferable to throw exceptions. This project combines these two aspects of error handling by designing a class that can behave like an error code and only throws an exception if the error code is not checked.

2. ERROR HANDLING

Errors fall into three categories: syntax, semantic and runtime errors. A syntax error occurs when you write a statement that is not valid according to the grammar of the C++ language whereas a semantic error occurs when a statement is syntactically valid, but does not do what the programmer intended. In this project we are concerned with runtime errors that are not caught by the compiler, and can affect the program in the following ways:

- They may not show up at all
- Cause the program to produce the wrong output or corrupt data
- Cause the program to crash.

To tackle these issues, it is necessary to handle errors.

2.1. Return Error Codes

A return code or an error code is a numbered or alphanumeric code that is used to determine the nature of an error, and why it occurred. They are commonly found in cases when the code attempts to do something it cannot do i.e. executing something that is outside the scope of the code. They can also be passed off to error handlers that determine what action to take.

For instance, the command line execution should return zero when execution succeeds and non-zero value when execution fails. When execution fails, the returned non-zero value indicates the corresponding error number.

While numeric return codes are simple and fast mechanisms, these are cases where they are not very effective.

1. If a function returns 1, is it trying to indicate an error, or is that actually a valid return value?

2. Functions can only return one value, so what happens when you need to return both, a function result and an error code?
3. Deep call stacks could be a problem. The error would have to travel back and forth within the stack.
4. In sequences of code where many things can go wrong, error codes have to be checked constantly since it gets difficult to trace the origin of a particular error.
5. Constructors don't have return types.

2.2 Exceptions

Exception handling provides a mechanism to decouple handling of errors or other exceptional circumstances from the typical control flow of the code. They are able to separate the actual logic of the code and the error handling.

Exceptions are thrown at a precise location of the error. This allows more freedom to handle errors for a given situation, alleviating some of the issues that return codes cause

It is important to raise the errors at their precise location, because an I/O state can change outside a program.

Consider the following example:

```
if(File.Exists("file.txt"))  
    File.Delete("file.txt")
```

The file might have been deleted by another process right after the if statement, before the Delete() call. With exceptions, it is easy to simplify an error when the deletion fails.

When working with files there are also a lot more things to consider, that might not be able to catch with ifs, for example the file is on a network connection that got unavailable, access rights that change, hard disk failure etc.

2.3 Comparing Error Coding & Exception Handling

The following table compares the advantages and disadvantages of error codes v/s exceptions for handling execution time errors:-

Error Coding	Exception Handling
PROS • Writing the code is easier as it does not have a difficult syntactical language construction. • Preferred in cases where recovery is possible. • Faster execution (in most cases) when compared with exception handling.	PROS • Reporting errors from constructors is possible. • Exceptions are easily propagated from deeply nested functions. • Function signatures are simpler.
CONS • Easy to forget error cryptic codes. • Easy to forget to handle error codes. • Error code arguments are not clear for success v/s error codes.	CONS • Examine all of its transitive callers when adding a 'throw' statement. • Runtime penalty due to stack unwinding.

When using return codes, scope of a function is responsible for handling all error codes.

But when using exceptions, the program knows that it can fail, and usually puts counterfire catch at chosen strategic position in the code, which can be outside of the scope of the user's code.

With exceptions you only need to handle the error if it could be a serious problem for the program. Otherwise execution will be silently passed back to the caller.

3. THE CODE

3.1. Reading a File

As an example for I/O related code, we read the contents of a file using the following three functions:-

1. OpenFile()
2. ReadFile()

3. CloseFile()

The `OpenFile()` returns the file pointer in case of successfully opening the file whereas it returns an `RStatus` wrapped `nullptr` in a case where the file could not be opened.

The `ReadFile()` reads the contents of the file and returns the number of bytes read. The `CloseFile()` takes the filehandle from `OpenFile()` and closes it. Initially, none of the functions throw an exception.

3.2. Class `RStatus`

In order to prevent users from forgetting to handle an error, we now wrap the function' return codes into a specifically crafted class "`RStatus`". `RStatus` works as a templated class that takes one template parameter: the status type (e.g. `FILE *`, `int`, `bool`). The class behaves like an error code and only throws an exception if the error code is not checked, by means of its destructor i.e. when the `RStatus` object goes out of scope.

3.3. Class `RExceptions`

The `RException` class inheriting from `std::runtime_error` acts as the base class for all exceptions from ROOT. The class assigns exception IDs and checks for inner exceptions.

Basically, `RException` can hold the properties of both the currently handled exception (as its inner exception) and the other exception (its outer exception) making it possible to nest exceptions of arbitrary types within each other.

3.4. Tests

The following functions test different scenarios in which the `RStatus` class is used:-

1. In a scenario where `OpenFile()` fails, and `RStatus` wrapped `nullptr` and also the return value is ignored, it should fire an exception as soon as the block in which `FileOpen()` was called ends.
2. If `FileOpen()` fails and its return value is checked. No exception should be fired.
3. If `FileOpen()` succeeds and it's return value is not checked. No exception should be fired.
4. Assume `OpenFile()` fails, it returns the `RStatus` wrapped `nullptr` and also the return value is ignored. Right after the `FileOpen()` is called, another exception concerning a different function is thrown. In this case, `RStatus` should not throw an exception of its own.
5. It should be impossible to create `RStatus` objects on the heap. `RStatus` should only ever exist on the stack.

4. RSTATUS CLASS

RStatus works as a templated class that takes one template parameter: the status type. The constructor takes a return code and stores it in a private member: fStatus.

If the instance of the RStatus class stores an unsuccessful state that is never checked (and only then), it should throw an exception.

The noexcept(false) operator is necessary because destructors are by default flagged as not throwing an exception.

The ~RStatus destructor is the location where almost everything concerned with checking if the error has been queried or not takes place.

An exception is thrown if the fStatus function returns a value which gives an error: IsError(fStatus), and if the error is not checked: !fIsChecked. The uncaught exceptions are checked in order to prevent throwing an exception if the object is deconstructed in the course of stack unwinding for another exception (as described in Test number 4). Thus if there are no uncaught exceptions: !std::uncaught_exceptions() holds and can be an exception thrown.

```
template <typename T>
class RStatus {
public:
    RStatus(const T &value) : fStatus(value) { }
    ~RStatus() noexcept(false)
    {
        if (!fIsChecked && IsError(fStatus)) {
            // Prevent from throwing if the object is deconstructed in
            the course of stack unwinding for another exception
            if (!std::uncaught_exceptions())
                throw RException("return value not checked");
        }
    }
}
```

In a scenario where everything goes as expected, i.e. the error has been checked then no exceptions are thrown and the value got by execution of the function fStatus is returned: return fStatus.fValue

```
// Use the RStatus<T> wrapper like a T
operator const T&() {
    fIsChecked = true;
    return fStatus.fValue;
}
private:
```

```
T fStatus;  
bool fIsChecked = false;  
};
```

We use a templated definition of RStatus in order to have a generic code that works with a variety of possible return types.

The RStatus<int> and the RStatus<FILE*> are the template class instantiations is very similar to working directly with FILE* and int and required only minor changes to the existing code. As seen below, working with RStatus<FILE*> and RStatus<int>.

<pre>RStatus<int> OpenFile(const std::string &path) { return fopen(path.c_str(), "r"); } { auto fp = OpenFile("path"); if (fp == nullptr) { ... } }</pre>	<pre>RStatus<FILE*> OpenFile(const std::string &path) { return fopen(path.c_str(), "r"); } { auto fp = OpenFile("path"); if (fp == nullptr) { ... } }</pre>
--	--

5. DIFFICULTIES FACED

1. Preventing exceptions from being fired during the course of stack unwinding:-
In the process of Stack Unwinding, the way to the catch block calls the destructors for objects allocated since the starting of any code block. Objects that were created within the block are deallocated in reverse order of their allocation.
By using std::uncaught_exceptions, the RStatus code is able to detect if the current thread has a live exception object, that is, an exception has been thrown or rethrown and not yet entered a matching catch clause, std::terminate or std::unexpected. In other words, std::uncaught_exception detects if stack unwinding is currently in progress.
It also detects how many exceptions in the current thread have been thrown or rethrown and not yet entered their matching catch clauses.
2. Preventing RStatus objects from being created on the heap:-
The stack is a special region of the computer's memory that stores temporary variables created by each function. The stack is a "LIFO" (last in, first out) data structure, that is

managed and optimized by the CPU quite closely. Once a stack variable is freed, that region of memory becomes available for other stack variables.

The heap is a region of the computer's memory that is not managed automatically and is not as tightly managed by the CPU. It is a more free-floating region of memory.

Once memory has been allocated on the heap, the programmer is responsible for using `free()` to deallocate that memory when needed any more. Because we rely on the fact that `RStatus` can throw an exception close to its creation when the enclosing scope is left.

It should be impossible to create `RStatus` objects on the heap for ROOT. `RStatus` should only ever exist on the stack.

To make `RStatus` non-creatable on the heap, the `new` operator has to be overwritten.

This condition has can be checked as follows:-

```
void Test005() {  
    auto status = new RStatus<int, -1>(0);    // <-- This should  
    not compile (heap allocated)  
    RStatus<int, -1> status(0);    // <-- This, however, should  
    compile (stack allocated)  
}
```

6. LOOKING AHEAD

The project is currently in the demonstrator state, independent from the ROOT codebase.

The next step is to integrate the code with ROOT and make it a part of the ROOT experimental namespace and then add it to the `RNTuple` class.

7. CONCLUSION

The project combines the two aspects of error handling by designing the `RStatus` class that can behave like an error code and only throws an exception if the error code is not checked.

8. References

- [1] <https://www.learncpp.com/>
- [2] <https://en.cppreference.com/w/>
- [3] <https://docs.microsoft.com/en-us/windows/win32/debug/error-handling>
- [4] <https://www.gribblelab.org/CBootCamp/>

9. ACKNOWLEDGEMENT

I would like to express my deep gratitude towards Jakob Blomer, my supervisor, for his patient guidance, enthusiastic encouragement and for his assistance in keeping my progress on schedule. My special thanks to Axel Naumann for his help especially during my first week. I would also like to extend my thanks to the EP-SFT Team and the Summer Student Team for having me as an intern at CERN.