TSDuck

Coding Guidelines

Version 3.2
April 2021

License

Contents

Acronyms and Abbreviations

ABI	Application Binary Interface
API	Application Programming Interface
BOM	Byte Order Mark (a feature of UTF-16 and UTF-32)
DRY	Don't Repeat Yourself (a good practice)
FOSS	Free and Open Source Software
GCC	GNU Compiler Collection
GPL	GNU General Public License
IDE	Integrated Development Environment (e.g. Eclipse, MS Visual Studio)
LGPL	Lesser GPL
MSC	Microsoft C/C++ Compiler
MSVC	Microsoft Visual C/C++
NIH	Not Invented Here (a bad practice)
OOD	Object-Oriented Design
OOP	Object-Oriented Programming
RTFM	The Most Important Acronym For Developers (yes, it is)
RTTI	Run-Time Type Information (C++)
STL	Standard Template Library (C++)
TDD	Test-Driven Development
UTF	Unicode Transformation Format (UTF-8, UTF-16, UTF-32, etc.)
XP	eXtreme Programming

References

[1] ISO/IEC 14882, "Programming Languages – C++", 1998 (the C++98 standard).

[2] ISO/IEC 14882:2011, "Programming Languages – C++", 2011 (the C++11 standard).

[3] ISO/IEC 14882:2014, "Programming Languages – C++", 2014 (the C++14 standard).

[4] http://en.cppreference.com/, Online reference of the C++ language and standard library.

[5] "The C++ Programming Language, Special Edition", Bjarne Stroustrup, Addison-Wesley, 2000.

[6] "The C++ Standard Library, A Tutorial and Reference", Nicolai M. Josuttis, Addison-Wesley, 1999.

[7] "Effective C++, Third Edition, 55 Specific Ways to Improve Your Programs and Designs", Scott Meyers, Addison-Wesley, 2005.

[8] "More Effective C++, 35 New Ways to Improve Your Programs and Designs", Scott Meyers, Addison-Wesley, 2008..

[9] "Effective STL, 50 Specific Ways to Improve Your Use of the Standard Template Library", Scott Meyers, Addison-Wesley, 2001.

[10] "Design Patterns, Elements of Reusable Object-Oriented Software", Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley, 1995.

[11] "SEI CERT C Coding Standard, Rules for Developing Safe, Reliable and Secure Systems", Software Engineering Institute, Carnegie Mellon University, 2016 Edition.

[12] "SEI CERT C++ Coding Standard, Rules for Developing Safe, Reliable and Secure Systems in C++", Aaron Ballman, Software Engineering Institute, Carnegie Mellon University, 2016 Edition.

[13] TSDuck Web site, https://tsduck.io/

[14] TSDuck source code repository, https://github.com/tsduck/tsduck

[15] TSDuck issues tracker, https://github.com/tsduck/tsduck/issues

1 Rationale for coding guidelines

In the computing community, there are many ways to develop software, many programming languages, many paradigms (functional, object oriented, etc.) and many conceptions of the *art of programming*. Each world has its own set of coding guidelines. But each world does have coding guidelines.

1.1 Foreword

This document is named *TSDuck coding guidelines*. But the name can be misleading.

- ⇨ First, this document is more than just a list of coding recipes for the TSDuck project [13]. It can be used as general coding guidelines for any C++ project. In fact, the first part can be used for any programming project since only chapter 3 is specific to the C++ programming language.
- ⇨ Second, ironically enough, the TSDuck project is not even fully compliant with these guidelines. TSDuck has been developed for a long period of time, 13 years at the time of this writing. The version 1 of TSDuck was not even written in C++ (and was not named *TSDuck* either). The experience of the author has evolved with the project itself. New guidelines have emerged. While they are applied to new code, old code is not retrofitted (see *[Rule 24] If it ain't broken, don't fix it*).
- ⇨ Third, the base of this document was not initially written for the TSDuck project either. The present document was derived from a more general document which was written by the author for professional software development. Because of this legacy, a maintainer of the TSDuck project may sometimes feel disconcerted while reading some parts of this document.

Year after year, the code base of TSDuck has proven to become more robust. The maintenance process improved the code quality through regular *refactoring* operations, preserving or improving the modular structure of the project. Modifications, improvements or implementation of new features were always developed in a very short time and limited to a local set of source files.

Applying the present set of guidelines has been of great help in this continuously improving robustness.

1.2 Industrial software requirements

In industrial software development, the keys to success are the management of the global cost of the software life-time, the reliability and the security of the software.

Costs: In the lifetime of a software product, the initial development often costs less than the maintenance. Consequently, an important requirement of industrial software development is reducing the maintenance cost. And sometimes this is at the expense of the initial development cost. Writing good and maintainable code may cost a little bit more initially but the final cost will be beneficial. Writing good and maintainable code requires a set of coding guidelines which are followed by the whole development team.

Reliability: The reliability of the software is planned from the design and is achieved through the quality of the code. The quality of the code also requires a common set of coding guidelines.

Security: The security of the code has been quite challenged by hackers in the last twenty years. Most attacks take advantage of flaws in the code, either plain bugs or complex vulnerabilities in code which is otherwise functionally correct. The experience on those vulnerabilities results in a set of secure coding guidelines.

1.3 What are coding guidelines?

There is no unique term for coding guidelines. In the industrial, academic or open-source software, coding guidelines are sometimes named coding standard or coding rules. But they all refer to the same type of document and serve the same purpose.

There are several types of coding guidelines. Some guidelines are generic; they describe general software development practices. Some guidelines are more specific; they apply to details in the writing of source code for specific programming languages.

This document proposes a set of coding guidelines, both general development guidelines and specific guidelines for each programming language.

Some rules are strictly necessary to write good code and are not negotiable in any environment.

Some others are simple coding conventions, for instance the naming conventions for identifiers. Even within one single programming language, many different coding conventions exist and are strongly advocated by their respective supporters. These types of conventions can be initially discussed but, once they are selected, they must be adopted by all developers in the organization or project.

Several sets of conventions can be individually satisfactory but using several sets of conventions in the same software project is not. Good and maintainable software must be easily understandable and consistent coding conventions are a key part of the understanding of the code.

1.4 The challenge of talented programmers

Adhering to common coding guidelines is sometimes hard to understand by the most talented programmers. Because of their talent, they have a strong opinion on the *right way* of writing code. And the conventions which were selected by the project or organization may not match. However, it is important to help them understand that the *right way* of developing good industrial software is a *common way* which takes advantage of the common talent of a development team and cannot be the sum of individual talents, how good they could be.

The most talented programmers are also often personal authors or contributors to various open source projects. It is also important to help them understand that their personal contribution to other public projects and their involvement in the production of industrial software are two distinct worlds with distinct rules.

The most talented programmers are welcome to contribute to *improve* the coding guidelines of the organization but not breaking them.

1.5 Classification of coding guidelines

This document adopts the following classification:

- ⇨ **Rule:** A mandatory regulation which must be applied without exception in all contexts. It cannot be negotiated or modified. A rule is preceded by **[Rule n°]** in the text.
- ⇨ **Recommendation:** A regulation which must be applied everywhere it is possible. But there are circumstances where it is difficult to apply. Situations where a recommendation must be applied or, on the contrary, may be omitted shall be documented. A recommendation is preceded by **[Recommendation n°]** in the text.
- ⇨ **Convention:** A mandatory regulation which must be applied without exception. Alternative conventions could have been possible in the first place but a single one had to be chosen. A convention is preceded by **[Convention n°]** in the text.

We use the word *guideline* as a generic term for rule, recommendation and convention.

In this document, we group conventions in separate sections. This separation is made on purpose to highlight the fact that rules and recommendations are not negotiable and apply to all projects while conventions may differ between projects.

If you select these guidelines, the conventions of this document shall be applied for any new project. However, when working on existing projects or external projects with pre-existing conventions, the developer may ignore the convention sections of this document and should apply the pre-existing coding *conventions* of their project instead.

However, rules and recommendations apply everywhere without exception, in legacy or new projects equally.

1.6 Purpose of this document

The ultimate purpose of this document is to **help developers**.

Consequently, this document wants to be user-friendly, readable, pragmatic and helpful. We want to avoid academic or normative style. This document is not an ISO standard; it must be the developer's companion.

Some sections may seem verbose or redundant. But, sometimes, verbosity and redundancy enforce clarity. What seems unclear and ambiguous in one paragraph will become clearer when rephrased in the next paragraph. Some sections or examples may seem useless. But what is useless to some can be useful to others.

This document also aims at convincing developers of the merits of these coding guidelines, rather than blindly imposing them. So, it is important that all developers deeply understand the rationale behind each guideline. This is why each guideline is followed by a rationale, examples or more details on its applicability or allowed exceptions. Counter-examples are sometimes even more important than examples to illustrate the risks and dangers you may encounter if you do not obey a given guideline.

Finally, this document is a permanent work in progress. All suggestions to improve or supplement it are welcome[1], as long as they preserve or improve its readability and usefulness.

1.7 Structure of this document

The chapter 2 describes rules, recommendations and conventions which apply to all programming languages and environments. The chapter 3 focuses on the C++ programming language.

All rules, recommendations and conventions are highlighted in a green box like this one.

The guidelines are sequentially numbered within this document. This means that the number of a specific guideline may change in different versions of the document.

The numbering of guidelines is purely sequential, regardless of any form of classification. Some coding guidelines documents with a very normative style use a class / type form of numbering such as RULE-CTYPES-027 for instance. The problem with that form is the difficulty to locate RULE-CTYPES-027 in the document. Which page? In the present document, locating [Rule 108] is simple: just open the book at some random page, look at the numbers there and you will immediately know whether you should browse forward or backward. Again, better useful and pragmatic than normative.

[1] TSDuck issues tracker, https://github.com/tsduck/tsduck/issues.

1.8 Enforcement of coding guidelines

Most programming languages are defined in non-ambiguous standards which clearly specify what is valid and what isn't. The syntactic and semantic correctness[2] of a program is enforced using an automated tool, the compiler. Programming language designers reject any new feature which couldn't be formally processed by a compiler.

On a theoretical standpoint, coding guidelines are useful only when they can be enforced using tools, typically automated technical ones.

However, the art of programming is not only a matter of theory. It requires a good deal of common sense and human practice. Many coding guidelines in this document are easily enforceable using standard code analysis tools. But some others are not. Quantitative or syntactic rules can be detected by technical tools. But qualitative rules can be evaluated by human experts only.

It is tempting to reject any rule which cannot be enforced by automated tools. However, this would greatly reduce the usefulness of the coding guidelines. As mentioned in 1.6, this document is not an ISO standard. Its purpose is to help developers to create robust, safe and cost-effective software. The eligibility of a rule is not based on its formal verifiability but on its practical usefulness for developers.

This is why some guidelines are enforceable using automated tools while others can be checked by human code review only. Listing which guidelines are automatically enforceable is beyond the scope of this document.

1.9 When to apply coding guidelines?

Put it simply, anytime you design or write some code.

However, do not rewrite existing code for the sole purpose of applying coding guidelines. When a team adopts the present document as reference for coding practices, it would be both counter-productive and dangerous to review and modify the existing code base to retrofit the new coding guidelines. In addition to the cost of such a lengthy operation, the risk to introduce bugs when modifying existing working code would be very high.

The coding guidelines apply only to new code or code which is modified for valid maintenance reasons.

This is consistent with [Rule 24] If it ain't broken, don't fix it.

[2] This is different from the functional correctness of the program. Otherwise, bugs wouldn't exist.

2 Generic coding guidelines

This chapter describes generic coding practices, independently of any specific programming language or framework.

Some of these guidelines may appear *very* generic in fact and evaluating the correctness of a code regarding these guidelines may be sometimes subjective or biased. But, as mentioned in 1.6, this document prefers to be helpful rather than normative. And while generic advices can hardly be considered as normative, they may be quite helpful to the developer.

2.1 Generic coding rules and recommendations

2.1.1 Software architecture

[Rule 1] The software architecture shall be driven by the following keywords:

- Simplicity
- Clarity
- Modularity
- Independence
- Reusability
- Maintainability

Simplicity: The simpler a system is, the more reliable it is. Apply the well-known KISS principle: *Keep It Simple and Stupid*.

Clarity: The software architecture shall be immediately understandable. And so must be the source code. Anyone shall be able to understand the source code without effort, especially newcomers.

Modularity: Break the design in smaller modules which are easier to maintain and understand.

Independence: The modules should be potentially used independently of others. Modularity is not the synonym of independence. Some modular systems have so many dependencies between modules than the modularity is only an artificial division of a very big spaghetti-like module. Typical pitfalls which break the independence of modules are the usage of global variables, excessive usage of implicit state, module interdependencies, etc. Drawing a dependency graph of the modules can give a first impression; if there are loops in the graph, the independence principle is broken.

Reusability: Reusability is the key to software cost reduction. A module shall be written in a generic way. It should be reusable in environments which are different from the original one. Each time you write a module, try to understand what is specific to your situation and what could be used outside of it. Write the module for the general case with a customized behavior from parameters and express your situations by applying parameters to your generic code. The "*Don't Repeat Yourself* "(DRY) principle is an application of reusability.

Maintainability: This is a corollary to all the preceding points. As mentioned in 1.2, the maintainability is the key to success in industrial software development.

2.1.2 Source code structure

This section describes the common rules on the structure of source files. Additional rules may be found in subsequent chapters for the various programming languages.

2.1.2.1 File size

[Recommendation 2] A source file should be limited to a maximum of 500 lines. For complex modules which can be hardly split into smaller pieces, 1000 lines is the maximum.

These limits are raw file size, including blank and comment lines, not only source code (but excluding the standard legal headers).

As explained before, simplicity is a key to maintainability. If your file is larger than these limits, it is probably poor quality and you should consider redesigning it.

2.1.2.2 Legal headers

[Rule 3] All source files shall start with a legal header which references the copyright and license information for the project.

All lines in the header must start with the appropriate syntax for comment lines in the language of the source file.

Application to TSDuck: the text of the legal header is the following:

```
TSDuck - The MPEG Transport Stream Toolkit
Copyright (c) 2005-2021, author's first name and last name
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice,
   this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
THE POSSIBILITY OF SUCH DAMAGE.
```

2.1.2.3 Comments

[Rule 4] A source file shall contain at least 30% comments. More precisely, after removing the initial standard legal header and all blank lines, at least 30% of the remaining lines in the file shall contain non-empty comments.

As explained before, clarity is a key to maintainability. A new maintainer shall not need to analyze the code to understand it, reading it should be sufficient. The code, its structure and its environment shall be described in comments. Comments must be everywhere, not only in the file header. Write comments both for you and for future maintainers. Explain why you invoke a function; do not assume that the maintainer knows what it is. If you use a specific code structure for a good reason, explain it in comments to avoid a future maintainer to "optimize" it later, breaking your intent.

This 30% comment requirement does not take into account the standard legal headers. The 30% proportion of comments shall apply to the description of the specific code and its environment, after removing all standard and non-significant headers. The standard comments which are processed by automated documentation tools such as Doxygen or Javadoc are included in the 30% comment requirement since they describe the actual code.

Automated tools may evaluate the amount of comments in source files and report insufficiently commented files.

[Rule 5] Comments should be written at the same time as the code and not after the code is completed.

When you write the code, you know precisely what you are doing at this specific time. This is the right time to explain what you are doing in comments. When the code is completed, some important thoughts you had when writing the code are already gone.

2.1.2.4 Self-documentation

[Rule 6] Use source code self-documentation tools such as Doxygen or Javadoc.

These tools use specially formatted comments to produce a set of HTML pages or PDF documents with the documentation of all the code. With these tools, it is no longer necessary to write separated detailed design documents; the documentation is automatically extracted from the code.

This is both a cost reduction and the guarantee that the code is consistent with the documentation.

Document everything correctly. A common pitfall is to build the minimum comment structure so that the documentation tool does not complain. The result is a short and obscure description of a function and a list of parameters without description. This is useless. Generate the documentation and read it or have it read by someone else. If the result is not satisfactory, complete the self-documentation comments.

[Rule 7] Maintain the code self-documentation.

Another pitfall is to modify the code later without applying the corresponding modifications (if necessary) to the self-documentation comments.

2.1.3 Revision control system

[Rule 8] All source files shall be managed in a revision control system.

The choice of the revision control system is outside the scope of this document and depends on the constraints of the organization (legacy environment, customer request, etc.)

Selecting a revision control system is sometimes subject to debates or folklore. Git is now a de-facto standard in the software industry. It is powerful enough to satisfy the vast majority of constraints and, more important, its usage is familiar to almost all developers nowadays.

Application to TSDuck: The project is managed using Git. The reference repository is on GitHub [14] within the organization named `tsduck`. Additional Git repositories are available in the same GitHub organization for unitary tests, third-party device drivers, etc.

[Rule 9] Each developer shall be accurately identified in the revision control logs. The identification includes the real name of the developer and his e-mail address.

This is necessary for accurate history tracking.

Example: Git tries to guess the real name and e-mail address of the author of each commit. However, the results are not always brilliant. A safe way to set the proper user identification for git is through the following shell commands:

```
$ git config --global user.name "Firstname Lastname"
$ git config --global user.email "username@domain.name"
```

The modifications are permanent. They are saved in the file *$HOME/.gitconfig*.

[Rule 10] Keep the source code repository clean and well organized. Specifically, ensure that dirty or non-original files are never pushed into the repository: temporary files, binary files, compilation products and more generally everything that can be regenerated from source files in the repository.

Most revision control systems have automatic ways of excluding dirty files, typically specifying file naming templates.

With Git, excluding dirty files is achieved through *.gitignore* files. See the man page *gitignore(5)* for more details. A template *.gitignore* file shall be provided.

Executables and objects are compilation products, not sources. They should be ignored in most cases. But UNIX executable files have no defined suffix. It is not possible to set up a global rule to ignore them (like ignoring all **.exe* in Windows). Thus, in each directory where one or more executables are produced, there must be a local *.gitignore* file which contains the exact names of the executables to ignore.

Binary objects or libraries (**.o*, **.a*, etc.) are usually ignored by some global rule. When a directory contains a third-party binary for which no source is available, this binary must not be ignored. There must be a local *.gitignore* file which contains a "not" ("!") directive to avoid ignoring this file.

Example: Content of a local *.gitignore* file illustrating these rules:

```
# Executable files to ignore
my_app
other_app
```

```
# Third-party binary library to include in the repository (not to be ignored)
!libthird.a
```

2.1.4 Internationalization

[Rule 11] All elements in a source file shall be written in English. Comments are in English. Identifiers use English words.

English is the corporate language.

Whether you like it or not, English has replaced Latin as the standard international communication language for decades. As of today, Klingon is not yet eligible as a valid international communication language.

[Rule 12] Text messages, if any, shall be written in English. If there is a need for internationalization, external localization files shall be used for non-English messages.

There are standard methods for this in the various environments and programming languages.

This rule is particularly important for GUI applications which are used by the general public.

Application to TSDuck: Since this is a very technical project, it is used by engineers who have at least some understanding of English. Consequently, there is no internationalization of the messages in TSDuck.

2.1.5 Modularity and compatibility

[Rule 13] Modularity is a contract definition. Define the contract and respect the contract.

A module shall be clearly defined by a public interface. This interface defines a *contract* between the module and its users. This contract shall be clearly described in the module interface, typically in the self-documentation comments. Whenever the module is modified, respect the original contract.

[Rule 14] Always preserve ascending compatibility.

A contract is a contract and you are not allowed to break it. If you make substantial modifications in an existing module, you may add features but you are not allowed to remove features or change the contract of existing features.

As a general rule, you must ensure that all existing code that could have used your module still compile and work correctly without modification.

If a service of your module seems no longer satisfactory, add a new better service with more parameters or whatever. But keep the old service with the same interface and semantic. You may mark it as *obsolete* to prevent new code from using it. Depending on the programming environment, the obsolescence can be only a description in the self-documentation comments or an attribute of the language ("*@Deprecated*" in Java for instance).

[Rule 15] Do not rename for fun or aesthetics.

This is a very bad practice which is unfortunately common in some environments. At the initial design of a module, some identifiers may have been poorly named. This is unfortunate but a contract is a

contract and you must preserve it. If the module has been already used, it is strictly forbidden to rename public entities. Such a renaming would break the ascending compatibility.

If the original name is *very* unfortunate and leads to confusion or misunderstanding, you may add a new service with the same semantic and a better name but do not remove the old one. Typically, move the implementation of the old poorly named service into the new one and simply invoke the new service from the old one. You may also mark the old service as obsolete to avoid using it in new code.

[Recommendation 16] Use Object Oriented Design (OOD) wherever it is possible.

OOD has proven its robustness and its efficiency. Use it. Most of the time, a set of functions apply to a common set of data. Rethinking in terms of OOD, this set of data is modelled as a data structure (an *object*) and the functions are *methods* of this object.

Object oriented *design* does not necessarily mean object oriented *language*. It is quite possible to apply OOD principles to programming languages such as C.

There are obvious exceptions to this rule, especially for some low level code. But even for low level code, think twice. You may consider that an interrupt service routine (ISR) is obviously not a candidate for OOD. But the ISR is certainly part of a driver. And a driver is mainly a set of routines around some device description data structure, which is the most basic definition of OOD.

[Rule 17] Do not expose implementation details.

The interface of a module shall expose only public declarations which are part of the contract.

Implementation details shall be hidden. Not only undocumented, but physically *hidden*. Application code trying to use implementation details should not compile. Even if you do not document them, someone will discover them in some definition or header file and may use them.

[Rule 18] Expose opaque data types only.

This is an application of the preceding rule. If you need to expose the existence of a data type (your *object* type maybe), you should hide its internal structure. Some fields may be strictly private, for internal use of your module. Some fields may be read-only for the user. Some fields may be modifiable by the user but with some control from your implementation.

And in the future, you may need to redesign the structure of the data type but you will also need to preserve the contract, preserve the logical access to these fields.

Thus, do not expose public fields. Use *accessors*, also named *getter* and *setter* methods.

If you think that this is a loss in performance, you are wrong. Most compilers have the ability to *inline* the code of a function. If you define an *inline* accessor method, the generated code will be only a data access. You get the same performance as a public exposure of the field but with a better contract and better maintainability of the code.

If you think that coding and documenting accessors is boring, well, you are right. But many IDE's (Eclipse for instance) can do most of the work automatically for you.

[Rule 19] Global variables are evil. Do not use them.

Global variables break the contract of a module. They also break the independence principle. And they also break the reusability principle.

[Recommendation 20] Static variables are evil. Avoid using them.

Static variables (in C parlance) are slightly different from global variables since they are not shared between modules. But in the OOD approach, a static variable is shared by all instances of its module. Most of the time, this breaks the reusability principle.

If some kind of persistent unique data is necessary, prefer the *singleton* design pattern over the static variable.

2.1.6 Naming conventions

This section lists a few generic naming conventions which apply to all programming languages. For rules which apply to a specific language, refer to the dedicated chapter of this language.

Usually, naming is addressed by interchangeable *conventions*. However, these conventions shall be managed by higher level common sense *rules*.

[Rule 21] Use meaningful names which are immediately understandable by the reader or maintainer.

Example:

```
int logicalFileIndex;   // OK, self-explanatory
int lfi;                // Questionable. Is LFI a really widely used acronym for
                        // Logical File Index? Or is it just your own convention?
int i;                  // OK if this is just a loop index in a small function.
                        // WRONG if this is a Logical File Index
```

[Rule 22] Do not use names which differ only in letter case, differ by one character only or too similar.

This is confusing for the reader and error-prone for the maintainer.

Counter-example: The following sets of identifiers can be easily confused.

```
int broadcastIndex;  // lower case ‘c’
int broadCastIndex;  // upper case ‘C’

int counter;
int counters[10];    // trailing ‘s’

int index1:          // digit 1
int indexl;          // lower case ‘L’
int indexI;          // upper case ‘i’
```

2.1.7 Coding principles

[Rule 23] Read The F... Manual. Read the documentation of all functions, classes or libraries you use.

This is such an obvious rule that it should not be worth mentioning. However, experience proves that it is not always followed.

Each time, **all the time**, **really**, read the corresponding documentation before invoking a function. Even if this is a function you personally developed some time ago. Understand the error cases, the returned error codes, the side effects, the exact usage of the parameters, etc.

Get prepared for reading documentation. Have a shell window ready for a *man* command, a browser with prepared bookmarks pointing to the online help, Doxygen or Javadoc pages. When coding on a secure network which is not connected to the Internet, make sure that a copy of all online documentations is available on a server within the secure perimeter.

[Rule 24] If it ain't broken, don't fix it!

Sometimes, a programmer reads some code and finds it clunky. Some programmers cannot resist the temptation to "fix" the ugliness of the code. This is a dangerous practice. In some cases, this modification may introduce a bug and turns a clunky but perfectly functional code into a buggy one.

In front of an existing and operational code, in the absence of other intent such as refactoring or generalization, the question shall never be "is it elegant?" but "does it work?" And if it works, do not modify it.

As a consequence of this rule, do not rewrite existing code for the sole purpose of applying coding guidelines. When a team adopts the present document as reference for coding practices, it would be both counter-productive and dangerous to review and modify the existing code base to retrofit the new coding guidelines. The coding guidelines apply only to new code or code which is modified for valid maintenance reasons.

[Rule 25] Do not uselessly over-optimise without profiling.

Do not make hypotheses on the performance of the code. Do not write more complicated code because you think it will be faster. You do not know what the compiler will generate; only the compiler knows. Let the compiler optimize the code. Recent compilers are very good at optimization.

Even if a local construct is slightly more efficient, you do not know the actual impact on the global execution time. It is dangerous to write more complicated code to gain 0.001% of execution time. On the contrary, because the code you write is more complex, it is more error-prone and more complex to maintain.

Write the code using a clean structure. When the final application code is ready, do some profiling in the real context of the application. This will give you the actual bottlenecks in the code. Then, you will know which parts of the code are worth optimizing.

[Rule 26] Write endian-neutral code. Make sure your code works identically on big-endian and little-endian processors.

Never assume that a piece of code will work forever on the same type of processor. Even embedded code will eventually be ported on other platforms.

Some CPU architectures are little-endian (Intel), some are big-endian (SPARC) and some can work in either mode (ARM, MIPS). To make provision for future ports, make sure that your code is not dependent on a specific endianness.

The endianness applies only to integer, floating-point data, UTF-16 and UTF-32 characters strings. ANSI and UTF-8 characters strings, cryptographic keys and other types of raw data do not have any endianness; they are just suites of bytes.

The endianness only matters for external data interchange or storage. Write or use *serialization* and *deserialization* libraries to switch back and forth between the native and external representations of data. Each programming language has specific ways of handling this gracefully.

Do not make any assumption about performance. For instance, if you handle external data in little-endian representation and your code targets a little-endian CPU, do not avoid serialization routines simply because you think that this would add a useless overhead. Well-designed serialization libraries do not add any overhead when the native and target representations are identical. In this case, the serialization and deserialization routines are inlined empty functions. But using them in the source code is beneficial. When you port the code to another platform with the opposition endianness, the program will work correctly without modification.

It is difficult, and sometimes impossible, to test that a program is truly endian-neutral, especially when the native and external data representations are identical. When possible, test your code on a little-endian and on a big-endian platform from the beginning.

[Recommendation 27] Use design patterns.

A number of design patterns are available in the literature. These patterns are proven constructs. Do not invent hazardous new designs, use proven design patterns. See [10] for instance.

2.1.8 Secure coding

[Rule 28] Do not assume anything.

This is the common root of many security flaws. Do not assume that the caller has checked a data, check it. Do not assume that this object is in such state because you think this is logical in this context, check it. Do not assume that a connection is established, check it. Do not assume that a buffer has such minimal size, check it. Do not assume that this pointer is non-zero, check it.

Etc. etc. etc.

[Rule 29] Do not trust any other code.

This is a corollary of the preceding rule: do not assume that a function does what its documentation says it should do. Specifically, if there is a simple way to improve the robustness and security of your code by checking what the other code did or returned, then check.

[Rule 30] Avoid temporary solutions or quick & dirty fixes.

If you are tempted to write a temporary fix to a problem, do not do that. Take your time and implement the good and final solution right away. Otherwise, because of the daily workload, you will delay the final solution, then you will forget about it and finally your temporary (and most certainly insecure) fix will become permanent.

Counter-example: A well-known product had a random number generation routine which was used to initialize cryptographic operations. The body of this routine was merely something like "**return 3**". It is clear that the developer intended to implement a proper solution later. But it never came.

[Rule 31] Inputs are evil. Never trust inputs. Check all inputs.

When you write a module, the inputs you receive come from an untrusted world by definition. You must always check all inputs. Addresses shall be validated (check null pointers, alignments and range constraints). Numerical values shall be checked for allowed ranges. The size of memory areas and buffers shall be checked. C-strings shall be checked for null termination. The format of special strings shall be checked (date and time, SQL request, etc.)

All functions shall check the validity of all inputs, always, without exception.

[Rule 32] Check all intra-application inputs also, at all levels.

Do not assume that the inputs of a function are valid simply because the caller is part of the same application. The caller code may have bugs. The caller code may be modified later and introduce new bugs. Your function can be reused in another context (remember that we encourage reusability) and the new caller may have bugs.

[Rule 33] Buffer overflow is your enemy. Check buffer sizes all the time.

If there is any theoretical, practical or syntactical possibility that some data could be larger than the memory area where you are going to copy it, check the data size, the index or whatever.

Use the capabilities of your programming language to define safe buffer objects with well-designed primitives instead of using raw memory buffers.

[Rule 34] Pay attention to the stack usage in a multi-threaded environment.

In a multi-threaded environment, all threads – or tasks – are not equal. The size of the stack of the main thread is often "unlimited", or at least extremely large, in the multi-mega-byte range. On the contrary, the size of the stack of all other threads is constrained. The default size depends on the platform but it is in the multi-kilo-byte range, much smaller than the main thread.

When you write a function, always remember that it will be potentially used in various environments, single-threaded or multi-threaded, with small stacks or large stacks.

Allocating very large objects on the stack may lead to unpredictable results if they overflow the current stack. If you are lucky enough, the implementation allocates non-accessible "guard" pages of memory before and after the stack of all threads. Thus, if you overflow a stack by a few bytes, the program will crash with an access violation error. However, if you allocate a very-very large object on the stack, the start of the object may step over the guard pages and lie into the stack of another thread. Thus, accessing the highest parts of your local object, you will corrupt local objects in other functions executing in another thread. Finding such a bug is a nightmare.

[Rule 35] Limit of size of "local" variables and all data which are allocated on the stack.

This is a corollary of [Rule 34]. Allocating large objects on stack may overflow the stack of a given thread. Large local objects shall be dynamically allocated on the heap and released before going out of scope.

[Rule 36] Always explicitly specify the size of the stack of all threads you create.

This is another corollary of [Rule 34]. If you do not explicitly specify the size of the stack of a thread, it will get the default size which is defined by the implementation. And this size is typically different from one platform to another. This means that your code is not deterministic. It may run on one platform and crash on another.

Be very conservative when computing the required size of the stack of a thread. Do not try to be too clever. It is useful to carefully compute the accumulated sizes of all local objects in all nested functions. But what should you do with the result? Clearly, do not use it as the stack size. The stack is cluttered with many stack frames or temporary working data which are allocated by the compiler. This extra size depends on the platform, the ABI, the compiler, the compilation options, etc. In practice, you cannot accurately compute the required stack size. So, use various sources of information. Compute your local data sizes, run tests and measure the maximum stack size and, at the end, double it just in case.

When you define an API, there are cases where the user passes the address of a "handler" or "callback", some code which is developed by the user but which is invoked by your library. If your library creates internal threads and some user "handler" or "callback" is invoked in the context of these threads, how do you compute the stack size of the threads you create? This cannot be transparent to the user. In the API of your library, you must give a way to specify the required stack usage of the user handler. The user shall be able to pass both the code address and required stack usage of his handler. Internally, to compute the stack size of your threads, you must add your own usage and the user's usage.

[Rule 37] Always check error codes. Take appropriate action and error processing.

Most functions return an error code or some indication that the processing failed somehow. Read the documentation of the function and **always** handle error cases.

This is specifically important for memory allocation routines.

But there are less trivial cases.

Example: Consider file management.

We probably always check the error code which is returned by a file opening or creation operation because we anticipate the risk that the file does not exist or we have no permission to create it.

When reading from a file, we generally always check the returned error code because we can reach the end of file at any time.

However, how many of us check the error code which is returned by a *write* operation? Nonetheless, writing to successfully opened files can fail at any time, because the volume is full, because the device has been removed, etc. In some cases, failing to write critical data such as transaction information or cryptographic keys without noticing can have disastrous effects.

[Rule 38] In case of error, fail safely.

When writing complex functions or module, you may invoke hundreds of external functions. Any of them can return an error situation. In the meantime, you may have opened, allocated, locked or somehow initialized many resources. If each error processing is individual, the code is bloated and potentially insecure.

Write a common error processing path which checks which resources should be cleaned and which safely cleans them.

Object-oriented languages can greatly help you in this process. Carefully designed destructors should always automatically clean up resources.

[Rule 39] Don't write spaghetti-like code, write state machines.

Avoid deeply nested **if** ... **then** ... **else** structures. Avoid multiple processing of the same cases in various places of the code. When applicable (and it is more often that you may think), redesign the code as a state machine.

[Rule 40] Physically clean up secure information from memory.

Passwords, encryption keys and less critical information are stored in memory while being processed. Once the local processing is completed, overwrite the corresponding memory with binary zeroes or random data. If some memory was dynamically allocated on the heap or on the stack, clean it up before freeing it from the heap or returning from the current function.

[Rule 41] Make sure that secure information is never accidentally written to disk in clear form.

Most high-level operating systems such as UNIX or Windows use virtual memory. When the physical memory of the system is about to fill, the system *swaps* or *paginates*, which means that some memory pages belonging to running processes are temporarily written to disk. If those memory pages contain sensitive data such as passwords or cryptographic keys, this becomes a security breach.

You must ensure that this never happens.

With virtual memory systems, make sure that all data buffers which contain sensitive data are locked in physical memory. Thus, the corresponding memory pages will never be written to disk.

Example: On Linux systems, use **mlock()** to lock a memory area in physical memory.

[Rule 42] Sensitive information shall be exchanged or written to disk in encrypted form only.

Persistent sensitive information usually needs to be stored or exchanged. Storage shall be performed in encrypted form only. Exchanging sensitive information shall be performed using an approved cryptographic protection layer, either individually or using a secure tunnel such as TLS 1.2. Note that all versions of SSL and versions 1.0 and 1.1 of TLS are considered insecure.

Encryption keys shall be adequately protected. Passwords and other similar data which need to be compared but not retrieved in plain form shall not be stored. Instead a salted hash is stored.

Use approved encryption algorithms only. Also use approved security protocols and approved key management methods only. Never use your own encryption algorithm or security protocol. In case of doubt, ask an expert.

[Rule 43] Usage of encryption shall be validated by a cryptologist or security expert.

Cryptography is a smoking gun. Encryption and security protocols are hard to secure. Naïve implementations or careless straight-forward usage of encryption primitives are known to be vulnerable to various types of attacks.

All the following mechanisms shall be validated by a cryptologist or a security expert before being used:

- Encryption, decryption.
- Signature generation and verification.
- Encryption keys and other random data generation.
- Key management and protection.
- Security protocols and secure data exchange.
- Usage of hash, nonce and challenge.
- Implementation of cryptographic primitives, choice of cryptographic libraries.

[Rule 44] Take care of reentrancy in reusable components.

Basic reusable components increase the productivity and the maintainability of the code. However, using the same component from concurrent threads may lead to race conditions and crashes.

Clearly document whether a component is thread-safe or not.

You may internally implement the required locking feature from the beginning to make the component thread-safe. However, this may have several drawbacks. There may be useless performance degradation in applications which do not share the component in multiple threads. Additionally, the concepts of *threads* and their associated *locking* features may be quite different between systems and it is not always easy to write portable thread-safe code.

One option is to write a simple and non-thread-safe component and then write a thread-safe variant of the component. Do not re-implement the component in the thread-safe variant. Encapsulate the calls to services of the non-thread-safe component in the services of the thread-safe variant.

Object-oriented languages with template or genericity features such as C++ and Java may help to write a unique portable thread-safe implementation. The component focuses on its core features only and accepts synchronization primitives or objects as template or generic parameters.

[Rule 45] Use the principle of “least privilege”.

This is a fundamental rule of secure software. It means that a code shall not run with any privilege it does not absolutely require for its nominal execution. In case of security breach, when an attacker takes control of your code, this attacker shall not be able to take advantage of extra privilege.

Example: If an application manipulates files on behalf of a specific user, the process shall have no more privilege than this user. More generally, the application shall run with the identity and privileges of the user it works for.

[Rule 46] Do not run any code using the *root* account or any kind of privileged user account.

This is a corollary of [Rule 45]. Only the operating system or low-level system services need system-wide privileges. There is usually no good reason to use the “root” account or “system” or whatever to

run an application. If you really think that you need to be "root", you are wrong. If you really insist, see next rule.

[Rule 47] Drop privileges as soon as possible.

This is another corollary of [Rule 45]. There are rare cases where your application needs to have elevated privileges for a few very specific operations. In that case, use the following scenario:

- Start the application with the strictly minimum set of privileges which are required to perform the privileged operations.
- Perform the privileged operations right at the beginning of the application.
- Drop all extra privileges immediately after.

Example: An HTTP server shall listen on TCP port 80 by default. This port number lies in the system range of ports. On UNIX systems, you need to be root to open this port. If your application implements an HTTP server on this port, it shall be started as root. At the very beginning of the application, opens the port 80. Do not do anything else. Do not work on files or system resources which can lead to security problems if the application is misused. When the TCP port 80 is successfully open, immediately switch to some non-privileged user account like **httpd**. The open file descriptor of the socket on port 80 is still valid and can be used from the unprivileged user account.

[Recommendation 48] Lock your application in an isolated enclave without access to the rest of the system.

This is yet another corollary of [Rule 45]. When your application is successfully attacked and control is taken by an attacker, the "pwned" application shall not be able to access the rest of the operation system.

Define the required environment for your application; define the corresponding boundaries and lock the environment within these boundaries inside a subsystem without access to the rest of the system. The available confinement mechanisms depend on the operating system.

Example:

- **chroot()** on Linux and some other UNIX systems.
- Linux containers (LXC).
- Virtual machines under control of a hypervisor (VMware, VirtualBox, XEN, HyperV, etc.)

2.1.9 Software evolution

[Rule 49] Avoid copy/paste source code. Use refactoring instead.

Most of the value in software engineering comes from the software reusability and its corollary, development and maintenance cost reduction. Software reusability means generic and reusable code. But writing clean generic code right from the beginning is not natural. The process of producing generic code is typically *extending* and *generalizing* existing code into generic software.

This is named *refactoring*, one of the most important concepts in modern software engineering.

Creating new code from existing code using copy / paste is *duplicating*, not refactoring. Duplicating code means duplicating bugs, duplicating maintenance costs and more generally increasing the entropy of the system.

Remember: Copy/paste propagates bugs but does unfortunately not propagate fixes.

[Recommendation 50] Avoid creating too many branches in the source code repository.

The presence of multiple persistent branches in the source code repository is a wide-scale variant of the copy / paste syndrome. It multiplies the maintenance costs and ruins the source code consistency. Fixing old bugs becomes a nightmare because of the multiple and divergent branches where the bug is present. Creating a branch may save some time on the very short term but the extra cost on the mid and long term is overwhelming.

Exception: Creating short-lived branches for exploration, integration or tests may be accepted under the firm condition that the branch will be rapidly merged into the main trunk and deleted.

2.1.10 Compilation errors and warnings

[Rule 51] Use the most paranoid "warning mode" of the compiler and fix all warnings without exception.

Compilation warnings are never gratuitous. A compiler warning always draws attention to a potential bug. Even if the code works as expected on the tested platform (i.e. the set of compiler & target execution system) the incriminated section of code may fail on another platform. Ignoring tons of uncorrected compilation warnings is a very bad practice. When porting to a new platform, you may have to fix hundredths of bugs that would have been avoided if the warnings were considered in time.

Even warnings which appear to be really harmless after analysis are dangerous because they create a culture of ignoring warnings. A listing of hundredths of "harmless" warnings hides new warnings which may indicate actual bugs.

The most paranoid warning mode depends on the compiler. Different compilers have different options. Some compilers find warnings that some other compilers don't. As a general rule, fix all warnings from all compilers. Your code must not generate any warning on any platform.

See sample sets of options in 3.2.1.

[Rule 52] Turn compilation warnings into errors.

Many compilers have an option which turns warnings into errors. Use it in all code without exception. Thus, the presence of one single warning prevents the code generation.

See sample sets of options in 3.2.1.

[Rule 53] Understand an error or a warning before fixing it.

This seems obvious but this is too often not applied in practice.

Fixing a compilation error or warning is not simply making it disappear. It is tempting to simply find a modification in the source code which gets the compilation message away. But is this the **right** fix? Does this remove the potential bug that was behind? Did you even understand the potential bug?

Example: You may get an error about a pointer type mismatch or a warning about a comparison between signed and unsigned integers. Of course, casting one object to the appropriate type removes the message.

And sometimes, this is the appropriate fix. But sometimes, this is not.

You need to deeply understand the situation to decide whether you should use a type cast or modify the code structure. Do you understand why "**if (a < b)**" produces a warning when **a** is a signed integer

and **b** an unsigned one (or the opposite)? Do you understand that the underlying risks include an index mismatch leading to a buffer overflow, one of the major sources of security breaches?

If you don't, do not fix the code yourself, get help first.

2.1.11 Makefiles

This section describes the rules to use GNU Make to build native software on UNIX platforms only (Linux, Cygwin, etc.) The cross-compilation environment is not yet addressed in this version of the document.

> [Recommendation 54] The command make should be usable at any point in the source tree. A makefile shall be present in all directories of the source tree. The name of the makefile shall be "Makefile".

For most projects, the source tree is complex. Sub-projects, sub-systems, test suites are implemented as sub-directory trees. Some developers work on specific sub-trees while others work the entire project tree (integrators for instance).

Consequently, running *make* should be possible anywhere in the source tree.

> [Recommendation 55] As a general rule, when run in a specific directory, the command make shall recursively operate on all subdirectories.

In the source tree, any directory and its entire tree of sub-directories shall be considered as a consistent sub-system to build. See the preceding rule.

Consequently, running *make* somewhere in the source tree shall build the corresponding sub-system, meaning the current directory and all subdirectories, recursively.

The setup from a common file shall provide the required tools to simplify this process (see next rule).

When a directory is simply an intermediate level where there is nothing to build locally, simply include the common file and nothing more. The first target in the common file shall recurse into the subdirectories.

> [Recommendation 56] All makefiles shall include a common file. This file contains the common set of rules, variables, options and targets which are used by all projects.

The actual content of this common makefile is outside the scope of this document.

Use a relative path to include the common file since the source code repository can be checked out at different locations on different systems.

Example:

```
include ../../Makefile.common
```

Application to TSDuck: There is a file named **Makefile.common** at the top level of the source tree. This makefile is extremely generic and is used in many different projects. Another more specialized makefile named **Makefile.tsduck** is present at the top level. This file first includes **Makefile.common** and then adds a few common definitions for the TSDuck project. So all makefiles in the source tree of TSDuck start with: something like:

```
include ../../Makefile.tsduck
```

2.1.12 Unit testing

[Rule 57] Use unitary and non-regression tests automation.

Use a unitary testing framework (Cunit, CppUnit, Junit, etc.) For each module, build the corresponding unit tests and integrate them in the framework. Cover most common legal, illegal and limit cases in unit tests.

Use a continuous integration framework (Jenkins, for instance) to run all unitary tests nightly and report failures.

Application to TSDuck: The test suite is split in two parts. The first part addresses low-level unitary tests, typically for C++ classes. The second part includes non-regression tests for TSDuck commands and plugins. See the TSDuck online programmer's guide, section *Testing TSDuck* [13].

[Rule 58] Use a Test-Driven Development (TDD) approach.

Using TDD, each module is written in parallel with its unitary tests. Modern methods such as Agile or eXtreme Programming (XP) advocate that the module and its tests shall be developed in parallel by distinct developers.

The typical scenario is the following:

- Create the module in the main development area. The module is initially empty. Make sure it is compiled and inserted in the main product (library, executable, whatever).
- Create the corresponding test in the unitary tests area. Make sure that the test suite is correctly built and executed, although the test suite is initially empty.
- Develop one feature of the module.
- Develop the corresponding unitary tests.
- Run the tests and debug all potential problems.
- Write another feature and its tests.
- And so on.

This approach has several advantages:

- The developer is forced to define a clear API of the module from the beginning. Otherwise, the unitary tests are difficult to define.
- In case of failure, the unitary tests provide a simple environment to debug.
- The module is immediately inserted in the automated non-regression tests. During further developments of the module, if you later break something that was previously tested, the new bug will be automatically caught.
- The integration time is reduced since each module is individually tested at development time.

[Rule 59] Add new tests for each fixed bug.

Each time a bug is found beyond unit testing, in integration phase or even production, there is a bug in the code. But there is also a bug in the unit test suite because it failed to catch the bug in the code.

The proper fix scenario is:

- Fix the unit test suite first: Add a unit test which exhibits the bug.
- Run the unit test suite and verify that it fails (the bug is caught).
- Fix the application code.
- Re-run the unit test suite and verify that it passes.

This method is also well suited to fight the "*resurrecting bug syndrome*". We have all encountered situations where a bug was fixed in the past but reappears later. This is usually due to some human error with the configuration management system. By enriching the unit test suite with test cases for all known bugs, we are able to detect the resurrection of outdated buggy versions.

2.1.13 Integration of open source software

[Rule 60] Never use Free and Open Source Software (FOSS) without a valid legal advice. In this context, "using" means integrating and linking with FOSS modules as well as copy / paste of FOSS source code.

This rule typically applies to proprietary software. Some FOSS may be impacted however when some combinations of incompatible licenses are used.

Today, a vast amount of generic or reusable software is available for free ("*free as a beer*") in the form of Free ("*free as in freedom*") and Open Source Software. It is tempting to reuse FOSS in current projects. While this is quite appropriate when writing open source software, many legal threats exist when writing proprietary software.

In the latter case, three topics must be studied when using FOSS:

- The license of the target FOSS.
- Your usage of the target FOSS.
- The planned deployment of your software.

Many different types of licenses exist for FOSS. The Wikipedia article "*comparison of free and open source software licenses*" lists more than 40 different licenses. Some of them are obscure or written by legal illiterate programmers and are difficult to understand. Actual legal cases which are based on these licenses are disputed or even contradictory.

Generally speaking, there are two main categories of FOSS licenses: the *permissive* licenses (the BSD License for instance) and the *invasive* or *copyleft* licenses (GPL for instance). While the former may be safely used in proprietary software with some care, the latter must be avoided at all costs.

The way the target FOSS is used is also important. A license may apply differently when the FOSS is copied / pasted in source form, statically linked or used as a side component. The LGPL variant of the GPL is typically designed for dynamically linked libraries. But some libraries are LGPL while others are GPL. And some cases are borderline. Dynamic reference ("*dlopen*") of GPL'ed shared libraries on Linux is a disputed subject for instance.

Finally, the type of deployment of your software matters. For proprietary software which is *distributed* outside the company (commercially or even for free), the license of the FOSS applies. But for internal tools which are *never* distributed, you may usually do what you want.

In all cases, this subject is too complicated for us, the developers, get a legal advice first.

2.2 Generic coding conventions

This section is present to fulfil the required separation of immutable rules and recommendations from potentially replaceable conventions, as explained in 1.5.

2.2.1 Character encoding

There is always some misunderstanding between the characters in a text and their binary representation in a file. In the most general case, a character is universally represented by a 32-bit UNICODE value[3]. But files are rarely represented in UTF-32[4] format.

When a file is transferred between systems or accessed from a shared disk by multiple heterogeneous systems, the physical content of the file remains the same but the various operating systems or production tools often interpret this physical content in different ways, leading to unpredictable results.

The following rules attempt to mitigate this problem.

[Convention 61] Restrict the character set of source files to the 7-bit ASCII set.

There are few reasons to use characters outside the 7-bit ASCII set in software source files. The syntax of all modern programming languages uses ASCII only. All identifiers and comments shall be written in English which is ASCII only. Any people name in comment (author, developer, etc.) shall be used without accent or local "decoration".

[Convention 62] Use exclusively line-feeds (LF, '\n', 0x0A) as line delimiters.

This is the usual UNIX vs. Windows or LF vs. CR-LF line format. All compilers on Windows understand files with LF as line delimiters. On the contrary, some compilers or interpreters in the UNIX world have problems with CR-LF line delimiters. The bash shell, for instance, considers the CR as part of the line.

Most editors and IDE's have an option to force the usage of LF as line delimiters, even on Windows systems.

Automatic clean-up scripts should be used on a regular basis on the code base to convert CR-LF sequences into LF.

Exception: When working with Git, it is possible to automatically switch back and forth between LF and CR-LF during check-out and commit on all or selected types of text files. This feature is managed though the *autocrlf* configuration option and *.gitattributes* files in the repository tree. If you use that feature, make sure that text files are committed with LF lines. If you want some Windows-specific text files (such as *.sln* or *.vcxproj* files) to be committed with CR-LF, make sure to properly reference them in a *.gitattributes* file.

[Convention 63] Do not use the tabulation character, use spaces.

In source code, the indentation is essential. It is tempting to use tab characters to indent. However, the size of a tab character may vary between tools. Most system tools use 8 characters per tab. But many IDE's use 4 characters (the usual single indentation width). When editing a source file, depending on the typing and automatic indentation features, the result is often a mixture of tabs and spaces in the indentations. When such a file appears correctly in an IDE with 4 spaces per tab, for instance, the same file appears totally messed up in an editor with 8 spaces per tab.

Most editors and IDE's have an option to force the usage of multiple spaces instead of a tab.

---

[3] Some environments, such as Windows, erroneously name "UNICODE" a 16-bit representation which is actually UTF-16 with surrogate characters. UTF-16 means Unicode Transformation Format – 16 bits.

[4] UTF-32 is untransformed UNICODE in fact.

Automatic clean-up scripts should be used on a regular basis on the code base to convert tab characters into spaces (but a uniform tab width must be applied and may not match the original environment).

Exception: The presence of an actual tab character is required in some specific file formats such as makefiles. But most editors and IDE's can handle this exception.

[Convention 64] Use UTF-8 encoding when internationalization is required.

It is sometimes required to use international texts. Always uses specific internationalization files for non-English languages. The actual format of these files depends on the toolset. Unless specified otherwise by a toolset, always use the UTF-8 encoding for these files. When present, the option "UTF-8 without BOM[5]" should be used.

When the syntax of a file format allows the explicit specification of the character encoding, use it to specify UTF-8.

Example: All XML files shall be saved in UTF-8 format and start with

```
<?xml version="1.0" encoding="UTF-8"?>
```

Exception: Some tools may impose or generate files using their specific encoding. Always use the character encoding which is the safest one for such tools.

[5] BOM: Byte Order Mark, a feature of UTF-16 and UTF-32 which is normally useless in UTF-8.

3 C++ coding guidelines

This chapter defines the coding guidelines for the C++ language.

3.1 C++ coding rules and recommendations

3.1.1 Language selection

[Recommendation 65] Unless it is impossible for a very good reason, do not use C, use C++.

C is an old and unsafe language. C++ is much safer than C. It is possible to improve the safety of C using all advanced features of ANSI C99 and the most paranoid warning mode of the C compiler, but C will never be as safe as C++.

Good reasons to use C instead of C++ include:

- Maintenance of a legacy application written in C.
- Embedded system development on a platform without C++ compiler.

All other reasons are usually bad reasons.

Modern embedded systems often use GCC as compiler front-end. Only the compiler back-end is specific to the target processor. And if a GCC back-end is available for a given processor, using it for all languages which are supported by GCC[6] is relatively cheap for the compiler team. So, if an embedded system supports C only but uses a GCC front-end, it is worth asking for C++ support.

If you need at least two excellent reasons to use C++ instead of C, one is named *constructors* and the other one is named *destructors*. These features are the essential bases of safe coding and they are not available in C.

For high-level applications, Java or one of the many Javascript-based frameworks are often even better choices than C++.

[Rule 66] Performance is never a good reason for not using C++.

There is a common misconception that C++ code is slower than C code. This is wrong.

The confusion comes from the fact that C++ has much more features than C and seems more complicated. But more features do not mean more generated code. Most C++ features are source-level structure and safety. If you are compiler aware, you realize that the generated code overhead is insignificant. It is quite possible to write good object-oriented C++ code with performance in mind. As an example, a good object design involves lots of very small methods. But if you declare them as *inline*, you get the performance of C with the features of C++.

[6] Remember that GCC does not mean GNU *C Compiler*, it means GNU *Compilers Collection*. It is a compiler framework which supports many languages front-ends and a common code generation back-end per target.

3.1.2 Modularity

[Rule 67] For each file **module.cpp** which is not a main program, there must be exactly one corresponding **module.h** which contains the declaration of the public interface of the module. No internal or private definitions shall be present in the header. No part of the public interface of the module shall be declared in another header file.

This is the natural C++ implementation of a module. A module must have exactly one public interface and one implementation. There is one file for each.

Exception: In some cases, there is no need for an implementation and the *.cpp* file is not present, everything is present in the header file. This can be the case where the module contains declarations only or when all functions and methods are short enough to be declared *inline*.

[Rule 68] Each module shall contain only one C++ class. If the class has inner classes, the inner classes are declared and defined in the same module at the top-level class.

This simplifies maintenance.

It is not possible to declare an inner class[7] in a different header file from its enclosing class. To keep the one-to-one association between **.h** and **.cpp** files, the definitions of the inner class methods are implemented into the same **.cpp** file as the enclosing class.

[Rule 69] If a data type is used only inside a module and is consequently private to the module, it shall be declared within the **module.cpp** file and not in any header file.

If the type is private, it should not be useable by any other module. If the type were declared in a header file, even a specific one, different from the **module.h** public interface, it could be included by another module and used outside its normal scope.

Note that in the early times of the C language, in the "Kernighan & Ritchie" era, it was common to declare all data types in one header file, outside of the source files. This practice is clearly outdated and should be banned. This separation of data types and code is purely based on the syntax, not on the semantics and is a violation of the principle of modularity.

[Rule 70] A module implementation file **module.cpp** shall start with an **#include** directive of its own interface file **module.h**.

Thus, the successful compilation of **module.cpp** validates two important points:

- The header file is self-sufficient.
- The header file is consistent with its implementation.

[Rule 71] A header file must allow multiple inclusions without generating errors.

You cannot manage how your header will be used. The compilation of a user application which includes your header file shall never generate an error simply because of your header.

[7] Inner classes (also known as nested classes) shall not be confused with subclasses.

Use case: A user code explicitly includes your header **module.h** as well as some other header **other.h**. If this **other.h** happens to also include your **module.h**, there must be no error.

In C++, the #**pragma once** is defined for this purpose. The old C tradition of conditional compilation using **#ifdef *MODULE*_H** is outdated and should be avoided.

Example: C++ header file **module.h**:

```
#pragma once
// ... file content here ...
```

[Rule 72] The structure of a header file shall be self-sufficient. The users of the header file must not have to specify other header files in order to use it. There must be no ordering requirement of the #include directives for the user.

This means that a header file must include all other header files which are needed by the declarations it contains.

Note that this rule is implicitly validated at compilation time using [Rule 70].

[Rule 73] The header file for a C module shall be safely compiled when included from a C++ module. A C module shall be safely linkable with C++ modules.

In a C header file, do not use C++ reserved names.

In a C header file, all language constructs which are specific to C or otherwise incompatible with C++ shall be conditionally compiled using the predefined macro **__cplusplus** (with two leading underscores). This macro is defined by [1] §16.8 when the compilation occurs in a C++ environment.

In a C header file, all C declarations which generate a reference to a linker symbol (function or data) shall be enclosed in **extern "C" {...}** when compiled in C++.

Example: The following structure is inserted at the beginning of the declarations in the C header file:

```
#if defined (__cplusplus)
extern "C" {
#endif
```

And the following structure is inserted at the end of the declarations:

```
#if defined (__cplusplus)
}  /* extern "C" */
#endif
```

[Rule 74] Use the anonymous namespace to define elements (data, classes, functions, etc.) which are internal to a module.

This avoids name clashes and namespace pollution.

This is the C++ equivalent of **static** declarations in C. Note that **static** has a different meaning in C++.

Example:

```
namespace {
    void SomeInternalMethod()
    {
        ...
    }
```

```
}
```

[Rule 75] Never use **setjmp()** and **longjmp()**.

This is an old and dangerous feature. It should never be used, except maybe in low-level routines of the kernel of an operating system.

[Rule 76] Never use **static** variables in **inline** functions.

If you do that, there will be one static variable per module where the function is used, breaking the semantic of the **static** attribute.

3.1.3 Naming and syntax formatting

Most of these topics are covered by coding **conventions** in 3.3. However, a few topics are clearly rules because they impact the reliability and good understanding of the code.

[Rule 77] Always use a namespace when declaring entities. Never define anything in the default namespace.

This avoids the name clashes during the link.

Application to TSDuck: Everything is defined in the namespace named **ts** or in one of its inner namespaces.

[Rule 78] The directive "**using namespace**" is strictly forbidden. There is no exception.

This avoids ambiguities. This also avoids some portability issues, especially with Visual C++ on Windows platforms.

All C++ standard entities must be referenced with the **std::** prefix.

Remember that there is no need to explicitly specify the namespace to reference an entity which is declared in the current namespace or an outer one.

[Rule 79] Explicitly use the "::" prefix for predefined entities which are defined in the default namespace.

This indicates an explicit reference to the default namespace and avoids ambiguities with entities which are declared in the current namespace or an outer one.

Example:

```
open("file");    // WRONG: may conflict with an open() method of this class
::open("file");  // OK, there is no ambiguity
```

[Rule 80] Do not place code after a comment on the same line.

This is confusing. The trailing code can easily remain unnoticed.

Example:

```
// comment line is OK
a = 1; // comment after code is OK
```

```
b = x /* WRONG: code after comment is confusing */ + 3;
```

[Rule 81] The comments always start with a // and extend up to the end of line. The usage of the C comment syntax /* … */ is discouraged.

A common problem with the C syntax **/*…*/** is the potential absence of closing ***/**. In some cases, the subsequent lines of codes are silently "swallowed" by the comment, up to the end of the next comment. Such bugs are very difficult to track.

By using the **//** syntax, all comments automatically terminate at the end of line.

Counter-example:

```
/* innocent comment */
a = 1;
/* WRONG: unterminated comment
b = 2;
/* the preceding comment actually ends here -->*/
c = 3;
```

In this example, the line "**b = 2**" is excluded from the compiled code. Good luck to find the bug!

[Rule 82] Using the { } braces in conditional and loop statements is mandatory, even if there is only one or no instruction in the block.

Omitting the braces is dangerous for the maintainability of the code. If the indentation is incorrect or if the unique statement in the block is complicated or spans multiple lines, omitting the braces makes the code hard to understand.

Good code:

```
if (x > a) {
    x = 0;
}

for (i = 0; i < max; i++) {
    print(a[i]);
}

while (readFile()) {
}
```

Bad code:

```
if (x > a)
    x = 0;

for (i = 0; i < max; i++)
    print(a[i]);

while (readFile());
```

Even worse:

```
if (x > a) x = 0;
```

Notes and anecdotes:

- Yes, I hate Python! Know your history. Using indentation for structuring code was abandoned in the 70's after the Fortran era.
- Failing to apply this rule was the root cause of the famous security vulnerability nick-named "gotofail" on macOS and iOS.
- The Linux kernel coding rules specifically prohibit the usage of braces when there is only one instruction. Do these guys ever heard of "gotofail"?

3.1.4 Coding style

[Rule 83] Avoid numerical constants, use "**static const**" declarations for these values.

This is the C++ way. Preprocessing macros shall not be used for constants in C++. Use preprocessing macros only for conditional compilation or for numerical values which are evaluated in the context of conditional compilation.

Exception: Usual constants such as **0** and **1** which are used in obvious contexts (reset, increment) shall be used without names since they are self-explanatory.

[Rule 84] The value of all preprocessing macros shall be enclosed into parentheses (in the absence of other syntactic constraints).

This avoids compilation ambiguities.

Example:

```
#define TS_GOOD (TS_FOO + 3)
#define TS_BAD   TS_FOO + 3    // WRONG: what about "a = 5 * TS_BAD" ?
```

[Rule 85] In all preprocessing macros, the parameters shall be enclosed into parentheses when referenced in the definition.

This avoids compilation ambiguities.

Example:

```
#define TS_GOOD(x) (2 * (x))
#define TS_BAD(x)  (2 * x)    // WRONG: what about "TS_BAD(a + b)" ?
```

[Rule 86] In all preprocessing macros, each parameter shall be referenced exactly once in the definition.

Macro preprocessing is only text substitution. When the actual parameter of a macro has side effects, it is evaluated as many times as referenced in the macro definition while the caller expects exactly one evaluation.

Example:

```
#define TS_GOOD(x) (f((x)))       // OK:    TS_GOOD(a++) => a incremented
#define TS_BAD1(x) (g((x), (x)))  // WRONG: TS_BAD1(a++) => a incremented twice
#define TS_BAD2(x) (h())          // WRONG: TS_BAD2(a++) => a not modified
```

Exception: There are macros which are reserved to specific usages where the actual parameters are not general-purpose expressions but must be *lvalues* or special syntactic structures.

Alternative: When it is required to reference a macro parameter several times, you cannot use a macro. You should implement a real function and declare it *inline*.

[Rule 87] Preprocessing macros should be avoided for code structures. Use **inline** function definitions.

This is the C++ way. This avoids all macro substitution pitfalls which are described in the preceding rules.

There is no difference in terms of code size or performance. The generated code for inline functions is expanded inline, just like a preprocessing macro. If the inline function is not used, no code is generated.

[Rule 88] The preprocessing directives **#ifdef** and **#ifndef** should not be used. Use **#if** followed by an expression instead.

This is more consistent with directives containing multiple conditions or **#elif** directives. For complex conditions, the code is more compact and more readable.

Example:

Good code:

```
#if defined(A)
    #define X 1
#elif defined(B) && defined(C)
    #define X 2
#endif
```

Bad code:

```
#ifdef A
    #define X 1
#else // not A
    #ifdef B
        #ifdef C
            #define X 2
        #endif // C
    #endif // B
#endif // A
```

[Rule 89] Use the preprocessing directive **#error** wherever there is no valid default alternative.

The **#error** will draw the attention of the developer when the code is compiled in a new environment which was not previously addressed. The developer who does the porting can precisely locate where there is some specific work to do.

Example: A simple list of mutually exclusive alternatives, ensuring that one branch is taken.

```
#if defined(TS_WINDOWS)
    typedef ... tsSystemError;
#elif defined(TS_UNIX)
    typedef ... tsSystemError;
#else
    #error "unknow O/S, please update tsSystemError in this header file"
#endif
```

Example: A more complex list of alternatives. But something must be declared at the end.

```
#if (defined(__i386) || defined(__x86_64)) && !defined(__little_endian)
    #define __little_endian 1
#elif (defined(__sparc) || defined(__powerpc)) && !defined(__big_endian)
    #define __big_endian 1
#endif

// ... more definitions
```

```
#if !defined(__little_endian) && !defined(__big_endian)
    #error "unknow endian, please update this header file"
#endif

#if defined(__little_endian) && defined(__big_endian)
    #error "conflicting endianness, please review this header file"
#endif
```

[Rule 90] When declaring multiple variables with the same base type, create multiple individual declarations, one per line.

Example:

Good code:

```
int i;
int j = 0;
int* p;
int a[10];
```

Bad code:

```
int i, j = 0, *p, a[10];
```

The two code excerpts are perfectly valid according to the C and C++ standards and have identical effects. However, in the bad code example, it is not clear that the line declares objects which are not **int** (the last two variables are a pointer and an array). It is also not clear that **j** is pre-initialized to zero while **i** is not.

[Rule 91] The order of evaluation of the operators in C and C++ is complex. Do not hesitate to add extra parentheses in order to make the code more readable.

Do you know what "**++p->a**" means? Does it increment the pointer first and then dereference it? Or does it dereference the pointer first and then increment the pointed field? Actually, the second alternative is right. But how many maintainers will know that?

By the way, did you even know that the self-increment operator **++** has distinct precedencies when it is used as a prefix or a suffix?

Example: Adding theoretically useless but helpful parentheses:

```
++p->a;     // WRONG: nobody really knows what this mean
++(p->a);   // GOOD: same as "++p->a" but more readable
(++p)->a;   // not the same as "++p->a" so parentheses are required anyway
```

[Rule 92] A switch statement must not contain any "fall through" path, i.e. all cases must end with a **break**.

The switch/case fall-through path is a rare and confusing construct. Most of the time, no **break** at the end of a case is a forgotten one, i.e. a bug. By allowing fall-through path, we cannot clearly identify if a missing **break** is a bug or a feature. So, to be safe, we forbid it.

Exception: It is allowed to have no **break** when there is no instruction at all after the **case**. This situation is not a real fall-through, this is a **case** with multiple equivalent values.

Example:

```
switch (x) {
    case 1:
```

```
            print("something");
            // WRONG: fall-through path is confusing, bug or feature ?
        case 2:
            print("else");
            break;
        case 3:  // OK, not a real fall-through but a multiple case entry
        case 4:
            print("ok");
            break;
        default:
            print("error");
            break;
    }
```

[Rule 93] A **switch** statement must end with a **default** entry.

This ensures that unexpected values are always processed, typically with an error processing.

This is especially useful when the *switch* is applied on all values of an **enum** type. You may think that the **default** alternative is useless since all values are handled. However, what will happen when you update the **enum** type and add a value? If several **switch** statements are used in various modules, you will probably forget to update the list of cases in one of them. Without **default** alternative, the new value will be silently ignored and the bug will be either hard to find or (worse) remain undetected. In the presence of a **default** alternative which triggers error code, the run-time error will immediately detect the bug.

[Rule 94] If a local variable is required in a **case** or **default** entry in a **switch**, declare a local block using **{ }** for the entry.

Do not declare a local variable in a **switch case** without enclosing **{ }** block. The scope of the local variable extends to the subsequent entries in the **switch**, which is usually not what you indented.

Example:

Good code:

```
switch (x) {
    case 1: {  // <== note the "{"
        int i; // local to this entry
        ...
        break;
    }          // <== note the "}"
    case 2:
        ...
        break;
    default:
        ...
        break;
}
```

Bad code:

```
switch (x) {
    case 1:
        int i; // C: compilation error
        ...
        break;
    case 2:
        // C++: i is still visible here
        ...
        break;
    default:
        ...
        break;
}
```

[Rule 95] In conditional expressions, explicitly compare against zero for non-boolean values such as integers or pointers.

The implicit interpretation of an integer or pointer value as a boolean is confusing. Sometimes, the "common interpretation" is even the opposite of the reality.

Example: The common sense is that a function which returns a boolean value would return true on success and false on error because the word "true" reveals a positive feeling while "false" reveals a negative one. Many Unix system calls do not return a boolean value. They return an integer which is zero on success and a non-zero error code on error. See how this can be confusing.

Confusing code:

```
if (close(fd)) {
    // You think it's a success ?
    // In fact, it's an error
}
```

Cleaner code:

```
if (close(fd) != 0) {
    // error processing
}
```

[Rule 96] Never use boolean operators (!, &&, ||) on non-boolean values such as integers or pointers.

Same reasons as the preceding rule.

[Rule 97] Do not use assignments in conditional expressions, even for boolean variables.

This is confusing. The lazy reader will misinterpret "**if (a = b)**" as "**if (a == b)**". So, assign first, then use the resulting variable in the conditional expression:

Example:

Confusing code:

```
if (big = isLargeFile(f)) {

}
```

Cleaner code:

```
big = isLargeFile(f);
if (big) {

}
```

[Rule 98] Reduce the scope of the variables to the smallest possible block.

This is more readable and easier to maintain.

Example: Loop indexes should be declared inside the **for** statement when possible:

Good code:

```
for (int i = 0; i < max; i++) {
    const int j = 3 * i;
    ...
}
```

Bad code:

```
int i, j;
...
for (i = 0; i < max; i++) {
    j = 3 * i;
    ...
}
```

[Rule 99] Delay the declaration of variable until they are used for the first time.

This avoids lengthy initial declarations which are not immediately useful. This also gives you a chance to declare the variable as **const** for instance.

Example:

Good code:

```
int a;
...
a = ...;
...
const int max = 2 * a + 1;
for (int i = 0; i < max; i++) {
    ...
}
```

Bad code:

```
int a, i, max;
...
a = ...;
...
max = 2 * a + 1;
for (i = 0; i < max; i++) {
    ...
}
```

[Rule 100] To set the initial value of an object, always prefer the initialization syntax over an assignment.

This is faster to execute. In the case of the initialization by assignment, the default constructor is first invoked, then the assignment operator is invoked.

Example: Objects **b**, **c** and **d** have an identical initial value. But the initialization of **b** is faster.

```
ts::Foo a;
ts::Foo b(a);    // Invoke the copy constructor
ts::Foo c = a;   // Invoke the default constructor, then the assignment operator
ts::Foo d;       // Invoke the default constructor
d = a;           // Invoke the assignment operator
```

In the case of objects **c** and **d**, the default constructor does something which is immediately undone by the assignment operator.

[Rule 101] The use of **goto** is prohibited.

The **goto** statement is the most discussed and infamous construct in computer software history.

Exception: The only acceptable exception is a common error path within one function, at the end of the function, after the **return** of the normal (non-error) path.

Example: Valid and acceptable usage:

```
class Foo
{
public:
    Foo(); // default constructor
private:
    FILE* f1;
    FILE* f2;
};

Foo::Foo(): f1(0), f2(0)
{
    // Open each resource
    f1 = fopen("file1", "r");
```

```
        if (f1 == 0) {
            goto error;
        }
        f2 = fopen("file2", "r");
        if (f2 == 0) {
            goto error;
        }

        // Do other initial processing
        // ...
        return;

    error:
        if (f1 != 0) {
            fclose(f1);
            f1 = 0;
        }
        if (f2 != 0) {
            fclose(f2);
            f2 = 0;
        }
    }
```

[Rule 102] A single function shall not exceed 50 lines of actual code or approximately 100 raw lines including comments.

Very long functions are hard to understand and maintain. They should be redesigned to extract local treatments in other well documented local functions, even if these local functions are called only once.

Exception: A very large **switch** construct with a lot of short entries may be acceptable. In fact, breaking it apart into several functions may be less maintainable.

[Rule 103] Do not pass parameters of non-elementary type by value.

Passing a parameter by value is legal but it requires copying the parameter value, typically on stack. While this is harmless for elementary types (integers, floats, enums and pointers), it can be costly for structured types.

In C++, this can even be devastating with polymorphic types (classes with at least one virtual function) when the type of the formal parameter is a superclass of the actual parameter. In this case, we get the *slicing problem*: the pointer to the **vtable** and the superclass parts are copied while the derived parts are not. The result is an inconsistent object with virtual methods possibly using fields that do not exist. In C++, the common way is to pass such parameters by *reference* (a C++-specific mechanism, not to be confused with passing by *address* – or by *pointer* – in C).

Example:

```
class Foo {...};

void f(int x);         // OK: elementary type by value (“in” parameter)
void f(int* x);        // OK: elementary type by address (“out” parameter)
void f(Foo x);         // WRONG: structured type by value, don’t do that
void f(const Foo* x);  // OK: structured type by address (“in” parameter)
void f(Foo* x);        // OK: structured type by address (“out” parameter)
void f(const Foo& x);  // OK: structured type by reference (C++)
```

3.1.5 Strict typing

To detect as many potential errors as possible directly at compilation time, use strict types and attributes on all declarations and definitions.

[Rule 104] Use the **const** attribute wherever an entity is used as read-only.

This applies to parameter declarations in function profiles and in object declarations.

Example: A value is computed once and reused later without modification.

```
const size_t max = count * sizeof(something) + offsetFoo;
size_t i;
for (i = 0; i < max; i++) {
```

Example: If a reference or pointer value is used to access a read-only area (at least through this pointer or reference), use the **const** attribute. This is especially important for function declarations since this establishes a more precise contract with the caller.

```
void LogMessage(const char* msg, const Time& time);
```

[Rule 105] Use the **volatile** attribute wherever an entity is potentially asynchronously updated by another thread or some hardware mechanism.

This guarantees that the generated code will not optimize the access to the variable by caching it in a register for instance.

[Rule 106] Do not use **volatile** on a non-elementary types (i.e. not integer and not boolean).

The compiler can hardly guarantee an atomic access on such types and there is a risk to get randomly corrupted values.

Redesign the code so that accesses to non-elementary types are explicitly synchronized.

[Rule 107] Be careful when using the **const** attribute on pointers. Do you want a variable pointer to constant data, a constant pointer to variable data or a constant pointer to constant data? A good practice is to use **typedef** for complex pointer types.

The order of the **const** attribute, the type name and the ***** sign makes the difference. This is often the source of obscure compilation error messages.

Example:

```
const char* a;              // a variable pointer to “const char”
char* const b = ...;        // a constant pointer to “char”
const char* const c = ...;  // a constant pointer to “const char”

a = "abcd";   // OK
*a = 'x';     // ERROR
b = "abcd";   // ERROR
*b = 'x';     // OK
c = "abcd";   // ERROR
*c = 'x';     // ERROR
```

It is more readable to use type declarations:

```
typedef char* CharPointer;
typedef const char* ConstCharPointer;

ConstCharPointer a;
const CharPointer b = ...;
const ConstCharPointer c = ...;
```

[Rule 108] Never use built-in integer types, unless explicitly required by the context.

The C and C++ standards do not define the implementation of the **int** type and its variants. Specifically, the types **int** and **long** are known to have different sizes on different platforms. Using them reduces the portability of the code by introducing subtle side effects on large values.

Of course, in the presence of a legacy library which uses **int** or some of its variants in an API, you are obliged to use the same type for the data which are used with this API.

Use the following standard types when the size of integer values matter. They are defined in the standard header **stdint.h**.

```
int8_t    uint8_t
int16_t   uint16_t
int32_t   uint32_t
int64_t   uint64_t
```

[Rule 109] Use the predefined type size_t for values which hold the size in bytes of a C/C++ object.

This is the standard definition in the C/C++ language. The C and C++ standards define **size_t** as the return type for the **sizeof** operator. The modern C/C++ standard libraries use this type in the profile of their functions for size values.

The actual implementation of **size_t** differs from one platform to another. It has typically the same size as a pointer. But there is no unique correspondence with the built-in integer types.

For instance, some 64-bit platforms define **int** as 32-bit and **long** as 64-bit while other 64-bit platforms define both **int** and **long** as 64-bit. But in all cases, **size_t** is 64-bit on these platforms. This is especially true when porting between Windows / Visual Studio and Linux / GCC, which is a quite common situation.

So, there are clearly at least three distinct sets of integer semantics:

- Built-in types (**int** et al.): not portable, no specific semantic, unpredictable, do not use.
- Size of objects (**size_t**).
- Values represented by a given number of bits (**uint8_t** et al.)

Variant: The standard type **ssize_t** declare a signed integer type of the same size of **size_t**.

[Rule 110] Never use the built-in type char for any other purpose that 7-bit ASCII characters.

Do not use **char** to represent array of bytes, use either **int8_t** or **uint8_t**, as defined in **stdint.h**.

Do not use **char** to represent characters outside the 7-bit ASCII range. In the context of internationalization, the binary representations vary. This can be a dedicated 8-bit character set (Latin-1, Latin-9, etc.) or some binary representation of UNICODE (UTF-8, UTF-16, UTF-32, etc.) See also 2.2.1.

In all cases, the management of internationalized character strings shall be encapsulated into some dedicated library.

Application to TSDuck: The types **ts::UChar** and **ts::UString** implement Java-like characters and strings (UTF-16 with surrogate pairs).

[Rule 111] Never use the built-in type **unsigned char**, nowhere, never.

Unsigned char represents nothing. It has no semantics at all. Either a data is an ASCII character and it is a **char** or it is a byte and it is an **int8_t** or **uint8_t**.

[Rule 112] Always use literal constants which are type-compatible with the context.

Examples:

```
int   i = 17;
long  l = 17L;      // 'L' suffix means a literal of type 'long'
char  c = '\0';     // do not use integer literal such as: char c = 0;
void* p = nullptr;  // C++ notation for a null pointer, not always binary zero
```

It is important to understand why using the '**L**' suffix is essential in integer literals which are used in the context of an expression the final type of which is **long**. Consider the following example.

```
const int i = 1;
const long l1 = i + 0xFFFFFFFFL;
const long l2 = i + 0xFFFFFFFF;   // same literal without 'L' suffix
```

You may think that both **l1** and **l2** have the same value. This may be the case. But they don't on some platforms where **l1** gets **4294967296** (the expected value) while **l2** gets zero.

Why?

On some platforms, **int** is a 32-bit value while *long* is a 64-bit value. In the case of **l2**, the intermediate context into which the literal is evaluated is **int** since **i** is an **int**. So, the literal **0xFFFFFFFF** is *first* evaluated into the context of an **int** and its value is -1. The addition is performed on **int** values, giving zero. *Finally*, the result is promoted to **long**, staying zero.

On the contrary, in the case of **l1**, the literal is explicitly of type **long**. Thus, the intermediate context into which the addition is performed is **long**. This time, **i** is *first* promoted to **long** and *then* the addition is performed on **long** values, giving the expected result.

Portability issue: The '**L**' suffix is the standard way to represent **long** literals. But we discourage the usage of predefined integer types (see [Rule 108]) for portability reasons. For integer types which are explicitly smaller than 64 bits (**int8_t**, **int16_t**, **int32_t**, etc.), using **int** literals is usually safe. But for explicit 64-bit integer types (**int64_t**, **int64_t**), there is no standard syntax to guarantee that a literal will be represented as a 64-bit value (remember that **long** is not a 64-bit type on all platforms).

Most compilers use a proprietary and non-portable syntax to represent 64-bit literals. Some common header file shall define specific portable macros which select the appropriate syntax.

Application to TSDuck: The common header **tsPlatform.h** defines such macros as illustrated below.

Declaration:

```
#if defined(_MSC_VER)  // Microsoft C/C++
    #define TS_CONST64(n)  n##i64
```

```
    #define TS_UCONST64(n) n##ui64
#elif defined(__GNUC__) // GCC
    #define TS_CONST64(n)  n##LL
    #define TS_UCONST64(n) n##ULL
#else
    #error "Unknown compiler, update TS_[U]CONST64 in this header"
#endif
```

Usage:

```
int64_t  a = TS_CONST64(0x7FFFFFFFFFFFFFFF);
uint64_t b = TS_UCONST64(0xFFFFFFFFFFFFFFFF);
```

[Rule 113] Use the keyword nullptr for null pointers, not the literal 0, not the NULL macro.

This is the only non-ambiguous way of specifying a null pointer.

The old macro `NULL` was specified for the C language, not C++. Early specifications of the C++ language commanded to use the literal `0` for null pointers. This has led to many ambiguous or incorrect code since this literal has two distinct semantics: integer zero of type `int` and null pointer to any type.

Note that GCC and LLVM (clang) have the option `-Wzero-as-null-pointer-constant` to emit a warning each time a literal `0` is used in the context of a pointer.

[Rule 114] Never change the order of enum values in a type declaration.

The **enum** types are strictly defined in the C and C++ standards. Specifically, in the absence of explicit numerical value, **enum** values are defined as implicitly consecutive numerical values. It is safe for C/C++ code to compare **enum** values.

If you change the order of the declarations in an **enum** type for simple aesthetic reasons (sort in alphabetical order for instance), you change the semantics of the type and you potentially break the user's code. This is why re-ordering the values of an **enum** type is forbidden.

If you add a new value in an **enum** type, think about why you add it.

If this is simply a new value, add it at the end of the type, regardless of aesthetic reasons.

You may insert it somewhere else only if there is a very good reason to insert it between two existing values, for instance because this is new logic state between two previously consecutive states in a strictly ordered state machine.

[Rule 115] Always use a type definition (typedef) for each meaningful type. Do not use basic types which somehow reflect the implementation of the type.

Example:

```
typedef uint16_t FooIndex;
```

This facilitates the maintenance if the implementation changes someday.

In the above example, if a 16-bit integer becomes too short some day for a **Foo** index, just change the **typedef**. Otherwise, you would have to change all references to **uint16_t** as **uint32_t** in all variables which have the semantics of a **Foo** index and only those **uint16_t**. Needless to say that the probability to introduce a bug at this stage is very high.

3.1.6 Assertions

Assertions are statements which are inserted in the middle of executable code. They *assert* a condition and abort the execution if this condition is false. Assertions are included in the generated code under some specific compilation options and removed by the compiler for the final production code. Assertions are very useful to check the consistency of the code.

[Rule 116] Assertion must be used to check the internal consistency of code. Assertions must never be used to check input values or other external conditions.

Assertions are debug tools exclusively.

An assertion shall be used only to check the internal consistency of the code. This means that an assertion shall be used to assert something that you know is always true, regardless of the external environment, if your code is right.

In other words, an assertion contains a condition which is always true if your code is bug-free, even in the presence of incorrect inputs or incorrect environment.

For instance, if your code is such that an index **i** shall always be less than some maximum value at the end of a loop, you may write "**assert (i < max)**".

Similarly, if you are about to write a data structure **ds** into a memory area **ma** which is defined as an array of **uint8_t**, you may write "**assert (sizeof (ds) <= sizeof (ma))**" and then "**memcpy (&ma, &ds, sizeof (ds))**".

But, under no circumstances, you should use some external condition such as an input parameter of the current function in an assertion. The semantic of an assertion is that it shall have no effect on correct code. The compiler may include or remove the assertion from the generated code and the execution of correct code should be exactly the same in both cases.

Checking an external or input parameter in an assertion is illegal since an invalid external condition is not a symptom of incorrect code (at least *your* code). If you use an input parameter in an assertion in correct code, this code will either run or fail in the presence of invalid input, depending on the compilation option.

The correct way of checking inputs is to use plain checking code, not assertions. In the presence of invalid inputs, write the appropriate error management path for the context (return an error code, log an error, perform a default action or whatever is required by the context).

Example:

```
int8_t buf1[(3 * TS_SIZE1) + 4];
int8_t buf2[TS_SIZE1 + TS_SIZE2];

// correct, sizes are compile-time constants
assert(sizeof(buf1) >= sizeof(buf2));
memcpy(&buf1, &buf2, sizeof(buf2));

// incorrect, depends on system run-time resources
int8_t* buf3 = (int8_t*) malloc(someComputedSize);
assert(buf3 != nullptr);

// correct, valid run-time check regardless of compilation options
if (buf3 == nullptr) {
    // do some error processing
}
```

```
// incorrect, depends on a run-time value
assert(sizeof (buf1) >= someComputedSize);

// correct, valid run-time check regardless of compilation options
if (sizeof(buf1) < someComputedSize) {
    // do some error processing
}
memcpy(&buf1, buf3, someComputedSize);
```

[Rule 117] Assert everything that could be asserted.

Use assertions as often as necessary. When writing code, always try to locate all the implicit assumptions you make (data size, **enum** values ordering, etc.). Locate all the conditions that must be true and which would corrupt something if incorrect (final index values, counters, etc.) Assert all of this.

When writing code, you easily assume conditions. You know that they are true by design. But you may be wrong (nobody is perfect). And some future maintenance may break the design. So, even if this seems futile, write assertions on critical conditions.

And remember that assertions are automatically removed by the compiler in production code. So, do not hesitate to use assertions, they are performance-free on production code.

[Rule 118] The debug code which is conditionally compiled (typically using some preprocessor symbol such as "**DEBUG**") shall not modify the functional behavior of the code.

The reasons are the same as for the previous rule.

Such debug can typically only log debug information.

Be aware, however, that the presence of debug code may have a significant impact on timing.

3.1.7 Secure coding

[Rule 119] Always initialize data with predictable values. Use zero for integers and pointers if there is no other meaningful values in the context.

Uninitialized variables are a common source of bugs which may be difficult to track. But getting different behaviors on different platforms or versions depending on different unmanaged initial values is even worse.

Example:

```
uint32_t x = a + b;
uint32_t y = 0;
uint32_t* p = 0;
```

[Rule 120] Do not use **memset()** to initialize structure. Use appropriate initial value for each type.

All binary zero is often not appropriate as initial value for a data structure.

Example:

Good code:

```
struct Foo {
    uint8_t* p;
    size_t   size;
};

Foo obj;
obj.p = 0; // “zero pointer”, not always binary zero
obj.size = 0;
```

Bad code:

```
Foo obj;
memset(&obj, 0x00, sizeof(obj));
```

[Rule 121] Make no assumption about the size or memory layout of a data structure.

Each platform has its constraints on data alignment. The compiler takes them into account when representing a data structure. The same data structure may have different sizes or memory layout on different platforms.

Example:

```
struct Foo {
    uint8_t  a;
    uint32_t b;
} Foo;
```

The size of this structure may be 5 or 8 bytes, depending on the platform. The field **b** may be at offset 1 or 4 from the beginning of the structure.

If your code depends on the fact that **b** is assumed to be at a specific offset, you should assert this. The following example asserts that **b** is at offset 1 after *a*:

```
assert(((char*)&((Foo*)0)->b - (char*)&((Foo*)0)->a) == 1);
```

This is quite a strange expression since zero, the **null** pointer, is explicitly dereferenced. But it is used only to evaluate an address and not a content. Thus, the pointer is never really dereferenced.

[Rule 122] Make no assumptions about the evaluation order of operands or function parameters.

The C and C++ languages do not specify the order of evaluation in expressions and function calls. If any operand or parameter has side effects, the final result is unpredictable and may vary from one platform to another.

Counter-example: Bad code which rely on the order of evaluation:

```
int a = 2;
int b = a * a++;        // WRONG: result can be either 4 or 6
f(print(a), print(b));  // WRONG: a and b may be printed in any order
```

[Rule 123] Do not assume that a null pointer (nullptr in C++) is represented by binary zeroes.

The C and C++ standards state that a zero value in a pointer context is interpreted as a *null pointer*. But, on some rare platforms where zero is a valid address, the compiler may use a non-zero bit pattern to represent the concept of a null pointer.

This rarity makes the problem even more dangerous. If your code relies on the fact that a null pointer is represented by a zero bit pattern, it will work on most platforms. Later, when recompiling the code on a new platform where null pointers are represented differently, the code will no longer work. And it will take ages to find the problem!

[Rule 124] In a function, never return a pointer or a reference to a local variable or a parameter of this function.

The local variables go out of scope upon return. The returned pointer or reference points to a portion of the stack which is no longer used. Worse, the referenced memory area will be reused by the local variables of another function call.

Consequently, returning pointer or a reference to a local variable or a parameter may lead to subtle random memory corruptions in the later functions. This kind of problem is a nightmare to debug.

[Rule 125] As a generalization of the preceding rule, always consider the lifetime of an object before passing it using a pointer or a reference to some other entity.

The preceding rule addresses this problem for statically allocated objects in the context of a mono-thread application. But similar problems exist with dynamically allocated objects and multi-threaded applications.

Object lifetime: First, what is the *lifetime* of an object? This depends on the nature of the object. If the object is directly declared in a function or code block, its birth is its declaration point and its death occurs when the execution flow exits from the function or block. If the object is dynamically allocated (**malloc()** or equivalent in C, **new** in C++), the allocation is its birth and the corresponding deallocation (**free()** or equivalent in C, **delete** in C++) is its death.

Dynamic allocation: When a function dynamically allocates an object and returns this pointer or passes it to multiple modules, the function loses all tracks of the usage of the pointer by other entities. When shall this object be deallocated is a complex question which must be addressed at the application design level[8].

Multi-threaded applications: If a function passes the address of one of its local variables to another thread, there is a risk that the function returns while the other thread still uses the referenced data. To protect from this, the function may synchronize its return point with the termination of the other thread (*join* operation). Otherwise, this problem must be addressed at the application design level.

[Rule 126] When you declare a function accepting some form of array or raw data buffer, always add a parameter to specify the size of the data (input) or maximum size of the buffer (output).

Never assume that the format of the data will reliably define the end of the data.

Example:

```
// WRONG: how much data should I send?
void SendData1(const void* data);

// OK
void SendData2(const void* data, size_t dataSize);
```

[8] C++ may mitigate this problem using implementations of the smart pointer design pattern.

Of course, in the second case, the correct declaration does not mean that the implementation is correct. But it is possible to write a correct implementation for this function.

On the other hand, the first function can never be safe and reliable.

[Rule 127] Never call any function with the following characteristics:

- A parameter is an address where the function is supposed to write to.
- The function may write more than one element at that address.
- No parameter gives the maximum number of elements to write to.

Many functions write data to a user-supplied buffer without letting the user specify the size of this buffer. These functions are simply badly defined and badly designed. They must be thrown away, simply.

[Rule 128] As a corollary of the preceding rule, never use any unsafe predefined string functions such as **sprintf()**, **strcpy()**, **strcat()**, all other **strXXX()** functions of the standard **libc** and many other standard functions.

Many standard functions are inherited from the old days of the Unix operating system in the 70's. They are obviously flawed in their definition. Just think twice about using standard functions, especially old ones.

Note that safe alternative versions exist for most of these functions: **snprintf()**, **strncpy()**, etc. Although safer than their ancestors, these functions still require some care. The pain with C-strings is the final null termination character. When a safe standard C-string function (one with an '**n**' in the middle of the name) copies a string into a buffer and the string is potentially larger than the buffer, the final string is truncated to the size of the buffer. In this case, truncation means no final null character. This also means that the result is not a valid C-string. If you use it as a C-string, the results are unpredictable (collecting memory garbage or crash).

Thus, when using the safe '**n**' C-string functions, you must always either check the return value to detect overflow (if supported by the function) or systematically overwrite the last byte of the buffer with a null character.

Example: The function, here **strncpy()**, does not indicate if a truncation occurred:

```
char s[10];
strncpy(s, longText, sizeof(s));
s[sizeof(s) - 1] = '\0';  // force valid C-string if truncated
```

Example: The function, here **snprintf()**, returns a value which can be used to detect the truncation:

```
if (snprintf(s, sizeof(s), "%s", longText) >= (int) sizeof(s)) {
    // result was truncated,
    // either process error or force valid C-string as above
}
```

[Rule 129] Zero out pointers after free (C) or delete (C++).

By zeroing the pointer, any accidental usage will result into an access violation (immediate crash) instead of a *reuse-after-free* or *double-deallocation* (subtle memory corruption which may run undetected for sometimes and hard to debug).

Example:

```
Foo* p = new Foo(...);
...
if (...) {
    delete p;
    p = 0;
}
*p = ...;      // immediate crash if preceding branch was taken
```

[Rule 130] Never cast a pointer into an integer.

First, converting between integers and pointers is bad design. Second, you do not know the size of a pointer and, on some platforms, casting a pointer to an integer results in a loss of information.

If you need to declare a field which stores a generic pointer, use **void***.

If you need to perform low-level address arithmetic on bytes (instead of C elements), cast the pointers into **uint8_t***.

[Rule 131] Be careful when mixing the sizeof operator with pointer or array index arithmetic. Remember that the pointer arithmetic uses the element size as a base while the sizeof operator returns a value in bytes.

Do not use **sizeof** to compute the last index in an array; this works only for arrays of **char** and 8-bit integers.

Counter-example: Incorrectly setting the last element of an array, leading to a crash (at best) or some random memory corruption (at worst).

```
uint32_t a[100];

// WRONG: not the last element but far, far away in memory
a[sizeof(a) - 1] = 0;
```

[Rule 132] To determine the number of elements in an array, do not use the explicit size from the declaration of this array. Use sizeof in an appropriate formula.

Otherwise, the maintenance becomes difficult if the declaration of the array is modified.

Example:

```
Foo array[SOME_COUNT];

// WRONG: what if the declaration of array is modified ?
const size_t numberOfElements = SOME_COUNT;

// This method is independent from the declaration of the array
const size_t numberOfElements = sizeof(array) / sizeof(array[0]);
```

[Rule 133] Do not declare arrays as local variables.

This is a corollary of [Rule 33] and [Rule 34].

An array can be very large and may overflow the stack of the current thread.

Moreover, using an array is subject to buffer overflow, even if you make your best efforts to avoid that. On a security standpoint, a buffer overflow is much more dangerous on the stack than on the heap. The stack is full of "code addresses", typically return addresses or exception handlers. A buffer overflow on the stack is the typical vulnerability which can be exploited for malware code injection. Even if there are some "code addresses" in the heap, they are less common. This is why a buffer overflow is less dangerous on the heap than on the stack.

This problem is more frequent in C than C++. With C++, we do not use arrays; we use *vectors* or other container classes. In instance of such a class is typically implemented as a short data structure containing pointers. The actual data are contained in dynamically allocated areas which are transparently managed by the container class.

[Rule 134] Never mix signed and unsigned integers.

Mixing signed and unsigned integers, typically dealing with index or offset values, is extremely dangerous and should be banned. The weird effect appears when offset values are decremented towards zero. After zero, the signed value goes negative while the unsigned one wraps up to 2^n-1. The two values are subsequently desynchronized.

Some coding guidelines advocate the usage of unsigned integers to manipulate sizes "because they never become negative". This is not always a good idea. An unsigned value never becomes negative for sure, but it becomes "extremely large" instead, which is not better or even worse in some case. It is true that the C/C++ standards use the type **size_t** for sizes and **size_t** is unsigned. But standards may have bad ideas too.

Counter-example: The following code is a very common way to explore a buffer backward.

```
i = size;
while (--i >= 0) {
    // do something on buffer[i]
}
```

The code is valid when the iterator **i** is signed. However, if **i** is unsigned, **size_t** for instance, this code loops forever (or most certainly crashes if **i** is used to access memory).

Do you understand why Java, a language "designed with security in mind", has no unsigned type?

[Rule 135] Anticipate and prevent integer arithmetical overflow and underflow.

In C/C++, the integer types always implement *modular arithmetic*. All operations are performed *modulo* 2^n. Arithmetical operations never overflow, they wrap up or down.

In some cases, this behaviour is fine. But in most cases it is not. In fact, integer arithmetic and modular arithmetic are two different domains with distinct usages but C/C++ implements only one of the two. Other programming languages have a different approach. The Ada language, for instance, defines two distinct families of integer types: integer types and modular types. All operations on integer types remain in the range of the type and throw an exception in case of underflow or overflow. Modular types, on the other hand, implement the same semantics as C/C++. Based on the application requirements, the developer chooses the appropriate type. But this is not possible in C/C++ or Java. The developer has to implement all checks manually.

The type of check depends on the operation (addition, subtraction, multiplication) and the signed / unsigned property of the integer type. Note that an integer division never overflows or underflows but the divider shall be checked to prevent "divide by zero" errors.

To check overflows on addition and subtraction, we need to know the maximum and minimum values of the integer type. To check overflows on multiplication with signed types, we need to know the proper function to compute an absolute value.

Characteristics of integer types in C

In C, the standard header **<limits.h>** declares constants for the minimum and maximum values for the various predefined signed and unsigned integer types. The header **<stdlib.h>** declares functions which return the absolute value for signed types. The following table lists these characteristics.

Type	Minimum value	Maximum value	Absolute value
`char`	`CHAR_MIN, SCHAR_MIN`	`CHAR_MAX, SCHAR_MAX`	`abs()` (promoted to `int`)
`unsigned char`	`0`	`UCHAR_MAX`	n/a
`short`	`SHRT_MIN`	`SHRT_MAX`	`abs()` (promoted to `int`)
`unsigned short`	`0`	`USHRT_MAX`	n/a
`int`	`INT_MIN`	`INT_MAX`	`abs()`
`unsigned int`	`0`	`UINT_MAX`	n/a
`long`	`LONG_MIN`	`LONG_MAX`	`labs()`
`unsigned long`	`0`	`ULONG_MAX`	n/a
`long long`	`LLONG_MIN`	`LLONG_MAX`	`llabs()`
`unsigned long long`	`0`	`ULLONG_MAX`	n/a
`size_t`	`0`	`SIZE_MAX`	n/a

Note that there is no generic way to determine the minimum and maximum values of application-defined integer type, such as **"typedef ... Foo"**. So, how can we check an overflow if we don't know the minimum or maximum value of the type? There is no generic solution to this problem in C.

Characteristics of integer types in C++

In C++, the standard header **<limits>** declares the standard class **std::numeric_limits**. This class provides a generic mechanism to determine various properties of numeric types, either integer or floating point. The header **<cstdlib>** declares overloaded versions of **std::abs()** for all predefined signed integer types.

The characteristics of an application-defined integer type **Foo** are:

- Minimum value: **std::numeric_limits<Foo>::min()**
- Maximum value: **std::numeric_limits<Foo>::max()**
- True when the type is signed: **std::numeric_limits<Foo>::is_signed**
- Absolute value: **std::abs(value)**

For some reason, in **std::numeric_limits**, **min()** and **max()** have the syntax of a function while **is_signed** has the syntax of a constant. But the functions are inlined and return a constant. So, there is no performance penalty.

Detecting underflow on subtraction with unsigned integers

The following code illustrates a very dangerous situation of underflow.

```
size_t start = ...;              // some starting index inside an array
size_t current = ...;            // current exploration index
size_t size = current - start;   // size the explored area
```

When **current** is less that **start**, the operation **current - start** underflows since **size**_t is an unsigned type. The variable **size** receives a very large positive value, close to 2^{32} or 2^{64}, depending on the platform. When **size** is later used as an actual data size, the most likely result is a buffer overflow, the best known security vulnerability, leading to all sorts of malware infections.

So, the code should be rewritten as follow:

```
size = current < start ? 0 : current - start;
```

The test **current < start** anticipates and detects the possible underflow in the subsequent subtraction. In case of detected overflow, the code uses an appropriate alternative (here a zero value).

Detecting overflow on addition with unsigned integers

Detecting overflows is a bit more complicated than underflows because the *wrapping value*, where the overflow occurs, depends on the size of the integer types. The following code illustrates how to detect a potential overflow:

```
size_t start = ...;  // some starting index inside an array
size_t size = ...;   // size of an area inside the array
size_t next;         // will receive the index after the area

if (start > SIZE_MAX - size) {
    // process arithmetic overflow
}
else {
    next = start + size;
    if (next >= size_of_array) {
        // process buffer overflow
    }
    else {
        // now you can safely process the array at index "next".
    }
}
```

This is the minimum required code to safely move forward into a buffer using index values!

Checking the potential overflow must be done using operations which *never* overflow or underflow. The operation **SIZE_MAX – size** is safe because the integer type is unsigned and **SIZE_MAX** is its maximum value.

On the other hand, the naive test "**if (start + size > SIZE_MAX)**" is both wrong and useless. It is wrong because the addition may overflow and its value is not reliable. And it is useless because the test is always false since no **size_t** value can be greater than **SIZE_MAX**.

Detecting underflow or overflow on addition with signed integers

With signed integers, the addition generates an error in the following cases.

- Overflow: Both operands are positive and the result is negative.
- Underflow: Both operands are negative and the result is positive.

Summary of underflow or overflow detection by type and operation

The following table summarizes the right ways to detect error conditions on the various arithmetical operations on signed and unsigned types. The generic names **MIN**, **MAX** and **ABS()** are used to designate the minimum value, maximum value and function returning the absolute value for the given

type. In C, you have to select the right constants and functions. In C++, use the generic mechanisms of the language. The operands are named **a** and **b**; the result is named **c**.

Addition (unsigned):

```
if (a > MAX - b) {
    // ERROR
}
else {
    c = a + b;
}
```

Addition (signed):

```
c = a + b;
if ((a > 0 && b > 0 && c <= 0) || (a < 0 && b < 0 && c >= 0))
{
    // ERROR
}
```

Subtraction (unsigned):

```
if (a < b) {
    // ERROR
}
else {
    c = a - b;
}
```

Subtraction (signed):

```
c = a - b;
if ((a > 0 && b < 0 && c <= 0) || (a < 0 && b > 0 && c >= 0))
{
    // ERROR
}
```

Multiplication (unsigned):

```
if (b != 0 && a > (MAX / b)) {
    // ERROR
}
else {
    c = a * b;
}
```

Multiplication (signed):

```
if (b != 0 && ABS(a) > ABS(MAX / b)) {
    // ERROR
}
else {
    c = a * b;
}
```

Division (signed or unsigned):

```
if (b == 0) {
    // ERROR
}
else {
    c = a / b;
}
```

Note: Due to rounding effects, the check on multiplication is a bit too aggressive. It detects all error cases but some marginal valid cases, close to the limit, may be incorrectly detected as errors. However, implementing accurate overflow detection on multiplication in the general case is very costly in terms of performance. The proposed solution is a tradeoff.

Application to TSDuck: The header file **tsIntegerUtils.h** declares template functions such as **ts::BoundedAdd()** or **ts::BoundedSub()**.

[Rule 136] Anticipate and mitigate floating point rounding

Floating point types are less prone to underflow and overflow than integers because they are always signed and their minimum and maximum values are very large. If you really need to check for overflow and underflow, use the same technique as for signed integers.

The main problem with floating point types is rounding and the accumulation of imprecisions after an arbitrary large sequence of arithmetical operations.

Never use the equality (**==**) or inequality (**!=**) operators on floating point values because these operators check the (in)equality of binary representations without taking into account rounding and imprecisions.

There are various complex ways to safely compare floating point values. But an accurate implementation should take into account the complete sequence of arithmetical operations which led to the values to compare. For a given sequence of operations, there is a given accumulated imprecision and this imprecision must be taken into account when comparing values. In practice, this is too complicated – and way too slow – to implement.

So, we must adopt a pragmatic approach when comparing floating point values.

Example 1: If we need a relative precision, we should reduce the comparison toward **1.0** and compare it with the *epsilon* value of the floating type. However, since *epsilon* is defined in the C++ standard as the "*the difference between 1 and the least value greater than 1 that is representable in the given floating point type*", the accumulated impression may have exceeded *epsilon*. So, in practice, it is recommended to compare to some small multiple of *epsilon*.

The following C++ function implements a generic numerical equality operator. It takes an additional parameter named **impr** (for imprecision) which is the small multiple of *epsilon*. The default value is 4 and is adequate in most cases. However, if you compare values which were created by long sequences of arithmetical operations, it is probably better to use a higher value.

```
template <typename T>
bool probablyEqual(T a, T b, int impr = 4)
{
    if (std::numeric_limits<T>::is_integer) {
        // T is an integer type, equality is valid.
        return a == b;
    }
    else {
        // T is a floating point type, equality is fuzzy.
        const T c = std::max(std::abs(a), std::abs(b));
        return c == 0.0 || std::abs(a - b) / c <= impr * std::numeric_limits<T>::epsilon();
    }
}
```

Usage:

```
float a, b;
...
if (probablyEqual(a, b)) {
    // Consider that a “pragmatically” equals b.
}
```

This C++ example is neat. With C, we must write three different implementations for this function, for **float**, **double** and **long double**.

Example 2: In some contexts, a relative precision is useless and an absolute precision is sufficient in practice. If you handle financial values for instance, the smallest amount of currency is usually a cent.

This means that everything smaller than **0.01** is irrelevant. So, regardless of the values and how they were computed, it is much simpler to write comparisons this way:

```
float a, b;
...
if (std::abs(a - b) < 0.005) {
    // Consider that a “pragmatically” equals b for financial data.
}
```

3.1.8 C++ classes

[Rule 137] The **struct** data types shall be avoided. Use classes instead.

The **struct** data types are allowed in C++ for C compatibility only. They are strictly equivalent to classes except that their fields without explicit scope are public by default (private by default in classes). So, there is no good reason to use **struct** in C++.

[Rule 138] Always declare a method of a class as **const** when it does not modify the object instance.

The contract is better defined. It asserts that the method will not modify the object. It also allows the method on **const** objects.

Example:

```
class C
{
public:
    void reset();        // this method will potentially modify the object
    void print() const;  // this method will not modify the object
};

void f(const C& c)
{
    c.print();   // OK
    c.reset();   // ERROR: cannot modify a const object
}
```

[Rule 139] Avoid public data members in class declarations.

Public data members give no control on the way the data members are used. Further maintenance is more difficult if you need to modify the internal structure of the class since you need to maintain a flawed contract.

Instead of publishing the data members, keep them private and export public *getters* and *setters*.

Good code:

```
class Foo
{
public:
    int getCount() const {return _count;}
    void setCount(int c) {_count = c;)
private:
    int _count;
```

Bad code:

```
class Foo
{
public:
    int count;
};
```

```
};
```

These two implementations have identical performances, thanks to the inlining of the setter and getter.

But the good code is much more maintainable. You may modify the internal structure of the class, you may completely remove the **count** field, you may add bound checking in the setter, etc. But you will always be able to maintain the same contract (the two public methods **getCount** and **setCount**).

[Rule 140] When you provide a method with a **const std::string&** parameter, consider the advantages of providing an overload with **const char*** parameter.

The type **std::string** provides a constructor from **const char***. So, you may invoke a method with a **std::string** parameter using a null terminated C-string or a string literal. But, you should inspect how you use the **std::string** parameter within the method. If you simply use its null terminated C-string representation (method **std::string::c_str()**), then it is probably faster to overload the method.

Example:

Slower code:

```
void f(const std::string& s)
{
    something(s.c_str());
    ...
}

std::string aString;
f(aString);
f("abcd"); // converted to std::string
```

Faster code:

```
void f(const char* s)
{
    something(s);
    ...
}

inline void f(const std::string& s)
{
    f(s.c_str());
}

std::string aString;
f(aString);
f("abcd"); // no conversion
```

[Recommendation 141] Try to define *interface classes* when possible. An interface class describes the API for a *service* which can be implemented by concrete classes.

The characteristics of an interface class are:

- All methods are public.
- All methods are *pure virtual*.
- There is no field.
- There is no static item.
- There is no constructor.
- There is an empty virtual destructor.

By convention, the name of an interface class ends with the suffix *Interface*.

The concept of *interface class* is not defined by the C++ standard. The C++ standard defines the concept of *abstract class* but it is less restrictive; an abstract class is a class with *at least* one pure virtual function.

The interface class is the C++ equivalent of the concept of *interface* in Java. The interface class is interesting because it proposes a reasonable approach to multiple inheritance.

Because of its characteristics, an interface class is typically limited to a header file. There is no **.cpp** file for an interface class.

Example:

```
class MutexInterface
{
public:
    virtual bool acquire() = 0;    // pure virtual method
    virtual bool release() = 0;    // pure virtual method
    virtual ~MutexInterface() {}   // empty virtual destructor
};
```

[Rule 142] Do not use "actual" multiple inheritance. A class should inherit from at most one non-interface class. A class may inherit from multiple interface classes.

Multiple inheritance is a powerful but dangerous concept. Using multiple inheritance from more than one concrete class often leads to structures which are complex and difficult to understand and maintain. After several levels of multiple inheritances, the result can be inconsistent sometimes.

The coding rules of some organizations even completely ban all forms of multiple inheritance.

Our coding rules propose a reasonable trade-off which is derived from Java: one "real" superclass and as many interface classes as necessary.

The [Recommendation 141] defines the non-standard concept of *interface class*.

[Recommendation 143] At the beginning of the declaration of each subclass, define a **typedef** with the name **SuperClass** for the superclass.

This convention gives an easy and reliable way to explicitly invoke a method of the superclass. This is the C++ equivalent of the **super** keyword in Java.

If the class hierarchy is refactored and a new intermediate class is inserted, simply modify the **typedef**, there is no need to browse the entire implementation to track explicit invocations of the superclass.

Note that C++ allows multiple inheritance and it is not possible to define a single superclass in the general case. But our [Rule 142] limits the number of non-interface superclasses to one. This single non-interface superclass is considered as the "actual" superclass which is declared with the **SuperClass** typedef.

Exception: If a class has no non-interface superclass, the **SuperClass** typedef is omitted.

Example:

```
class A
{
public:
    void f();
};

class B: public A
{
public:
    // Standard name for superclass, modify it if you change the inheritance
    typedef A SuperClass;

    // Override method
    void f()
    {
        // Explicit (and maintainable) invocation of superclass
        SuperClass::f();

        // Specific additional processing in subclass
        ....
    }
};
```

[Rule 144] In the documentation of a method of a class, clearly indicate if the method needs to be explicitly invoked by subclasses which override the method. If necessary, also indicate if the superclass method shall be invoked at the beginning or at the end of the subclass method.

This is entirely dependent on the implementation of the superclass. Only the developer of the superclass can decide what is necessary.

Example:

```
class File
{
public:
    // Invoke at beginning of overridden method in subclasses
    void open(const std::string& fileName);

    // Invoke at end of overridden method in subclasses
    void close();
};
```

When we specialize the class **File** to add buffering capabilities for instance, the methods **open()** and **close()** must be typically overridden to add the setup and flush of the buffer, respectively. But, while the buffer management is handled in the subclass, the file management shall remain in the superclass.

```
class BufferedFile: public File
{
public:
    typedef File SuperClass;

    void open(const std::string& fileName)
    {
        SuperClass::open(fileName);
        // ... setup the buffer
    }

    void close()
    {
        // ... flush the buffer
        SuperClass::close();
    }
};
```

Exception: This rule does not apply to constructors and destructors. The compiler automatically invokes the superclass constructor *at the beginning* of the subclass constructor. It also automatically invokes the superclass destructor *at the end* of the subclass destructor.

[Rule 145] Never overload methods which differ only in the type of integer or pointer parameters.

In some contexts, the default integer type conversions give unexpected results. In case of maintenance of the code, you may even accidentally break the contract of the class.

Example:

```
class C
{
public:
    void f(int);
    void f(long); // BAD PRACTICE
};

int a = 1;
long b = 2;

C x:
x.f(a);
x.f(b);
```

In this example, **x.f(a)** invokes **C::f(int)** and **x.f(b)** invokes **C::f(long)**.

But let's assume that the initial version of the class C only had one method **C::f(int)**. The user code **x.f(b)** invoked **C::f(int)** since this was the only possible method at that time and the C++ compiler automatically downgraded *b* from a **long** to an **int** before the call.

Now, in the maintenance phase, assume that you add the **C::f(long)** method. When recompiling the existing user code, **x.f(b)** no longer invokes **C::f(int)**. Now it invokes **C::f(long)** since this is a better match. And the two methods may perform very different actions. So, by adding the overload

C::f(long), you break the contract and break the ascending compatibility of your class and the compiler does not even tell you it is different now. This is strictly forbidden by [Rule 13] and [Rule 14].

Note 1: This rule is also valid for pointers and integers. Using the literal '**0**' can be indifferently interpreted as an integer literal or as the null pointer.

Note 2: The terms *override* and *overload* shall not be confused. A method of a class *overrides* another when it redefines a method with the same name and same profile in a superclass. A method *overloads* another when it redefines a method with the same name and a *different* profile in the same context.

Example:

```
class A
{
public:
    void f(int i);
};

class B: public A
{
public:
    void f(int i);                 // override A::f(int)
    void g(int i);                 // overload B::g(const std::string&)
    void g(const std::string& s);  // overload B::g(int)
};
```

[Recommendation 146] When an overloaded method is overridden in a subclass, override all overloaded variants, unless there is a good reason to hide some of them or to add a new one.

Hum? This sounds a bit mysterious...

Counter-example: Explanations are always better with an example.

```
class A
{
public:
    void f(int i);
    void f(const std::string& s);
};

class B: public A
{
public:
    void f(int i);
};

A a;
B b;
a.f(1);       // OK, call A::f(int)
a.f("foo");   // OK, call A::f(const std::string&)
b.f(1);       // OK, call B::f(int)
b.f("foo");   // ERROR, does not compile, B::f(const std::string&) is hidden
```

In class A, the method **f()** is overloaded. There is one version taking an integer as parameter and one version using a character string. In class B, **f()** is overridden but only one version is declared, the integer one. As a consequence, the string one is hidden.

In C++, when you override a method in a subclass, all overloaded versions (methods with the same name in the superclass) are hidden. Only the explicit declarations in the subclass can be used. This is why **B::f(const std::string&)** is hidden in the above example. You cannot use it.

As a general rule, a subclass specializes a contract. It adds or replaces services but does not remove any. This rule is broken in the example.

Exception: In some cases, it can be legitimate to remove (hide) or add overloaded versions of a method. But this is tightly linked to the semantics of the object.

[Rule 147] Never change the default values for the parameters of a function after the function is published.

Changing the default values changes the contract.

Consider the following function:

```
void paint(Color c = BLACK);
```

All users which invoke **paint()** without argument are confident that the color will be black. If you later change the default value to **BLUE** and recompile the code, the users of the contract will obtain a different result.

[Rule 148] When overriding a virtual method with default parameters in a derived class, use the same default values for the parameters.

Counter-example: Let's review the consequences of breaking this rule. Assume that all *shapes* should be painted in black by default, except the *rectangles* which should be painted in blue by default.

See the following WRONG implementation:

```
class Shape
{
public:
    // "I see a red door and I want it painted black"
    virtual void paint(Color c = BLACK);
};

class Rectangle: public Shape
{
public:
    // WRONG: overriden method with other defaults
    virtual void paint(Color c = BLUE);
};
```

See why this implementation gives incorrect results:

```
Rectangle r1, r2;
Rectangle* p1 = &r1;
Shape* p2 = &r2;

p1->paint();  // Rectangle r1 is painted in BLUE
p2->paint();  // Rectangle r2 is painted in BLACK !
```

The default values for the parameters are always evaluated in the context of the caller, not the callee. When an object is accessed through a pointer to a superclass, the context of the caller is the superclass and the superclass' default parameters are applied.

This is why we impose to keep the same default values for parameters.

To solve the specific problem of the example where actual defaults should be different between the superclass and the subclass, a safe alternative would be to use overloading instead of default parameters as illustrated below.

```
class Shape
{
public:
    virtual void paint(Color c);
    virtual void paint() {paint(BLACK);}
};

class Rectangle: public Shape
{
public:
    virtual void paint(Color c);
    virtual void paint() {paint(BLUE);}
};
```

In this case, the parameter **BLUE** is evaluated in the context of the callee, which is **Rectangle::paint()**.

3.1.9 C++ constructors and destructors

[Rule 149] For a class type, always declare both default constructor and copy constructor. If any of them should be disabled, make the declaration private and provide no definition.

The default and copy constructors are implicit if not defined by the developer. But the default implementation may not be adequate to your class. So, you shall either provide an explicit version or explicitly disable it.

If you don't and if you do not want a default or copy constructor because they are meaningless in your context, the compiler will implicitly generate wrong one for you. If any default object declaration is used, the buggy implicit constructor will be used.

Example: Disabling default and copy constructor:

```
class C
{
public:
    // Only my explicit constructors are valid (definition required)
    C(const std::string&);
    C(int, int);
private:
    // Private declaration of default and copy constructors (no definition provided)
    C();
    C(const C&);
};
```

Alternatively, the C++11 standard allows the following syntax. It means that the two constructors must not have implementations. So, if you provide one by accident, the compiler will reject it.

```
private:
    C() = delete;
    C(const C&) = delete;
};
```

[Rule 150] Always free all resources which are private to the class in a destructor.

Well-designed classes safely prevent resource leaks: memory, open files, locked resources, etc.

[Rule 151] Always provide a virtual destructor.

If your destructor is not virtual, it will not be invoked when an object is destructed using a **delete** of a pointer to one of its super-classes.

Counter-example: Class C1 is a super-class. Subclass C2 has a non-virtual destructor while class VC2 has a virtual destructor. This example illustrates how instances of class C2 can be incorrectly destructed, leading to potential resource leaks.

```
class C1 { ... };
class C2:  public C1 { public: ~C2(); };
class VC2: public C1 ( public: virtual ~VC2(); };

// later in the code:
const C1* x = new C2();
const C1* y = new VC2();
delete x;  // destructor ~C2 is NOT executed !
delete y;  // destructor ~VC2 is executed
```

[Rule 152] All destructors shall be idempotent.

In rare obscure cases, a destructor may be invoked twice on the same object. This may especially happen if the destructor is explicitly invoked once and will be likely implicitly invoked later when the object goes out of scope.

Thus, the implementation of a destructor shall be coded so that it is safe when invoked more than once. This is the meaning of *idempotent*.

Counter-example: Non-idempotent destructor:

```
class C
{
private:
    int* _p;
public:
    C(): _p(new int[8])
    {
    }
    virtual ~C()
    {
        delete[] _p; // double deallocation if invoked twice
    }
};
```

Example: The following alternative destructor is idempotent:

```
virtual ~C()
{
    if (_p != 0) {
        delete[] _p;
        _p = 0;
    }
}
```

[Rule 153] In a constructor or destructor, never invoke any virtual method of the class.

During the execution of the constructor and destructor, the **vtable** of an object instance points to the virtual methods of the class which defines the current constructor or destructor. On the other hand, during the rest of the lifetime of the object, the **vtable** points to the virtual methods of the actual class of the object. This can be very misleading and hard to debug.

Counter-example: Consider the following class **C1** which incorrectly invokes a virtual method in the constructor and destructor.

```
class C1
{
public:
    virtual void vf(const std::string& s);

    C1() // constructor
    {
        vf("in constructor");
    }

    ~C1() // destructor
    {
        vf("in destructor");
    }

    void f() // some standard method
    {
        vf("in method f()");
    }
};
```

Now consider a subclass C2 which overwrites the virtual method.

```
class C2: public C1
{
public:
    virtual void vf(const std::string& s);
};
```

Given the nature of virtual methods, for a given instance of **C1** or any of its subclasses, it may be expected that the same virtual method **vf()** will be used in all contexts. But this is not true.

Let's assume that **vf()** simply displays a message:

```
void C1::vf(const std::string& s)
{
    std::cout << "C1::f: " << s << std::endl;
}

void C2::vf(const std::string& s)
{
    std::cout << "C2::f: " << s << std::endl;
}
```

Consider the following applications code:

```
int main()
{
    C2 c;
    c.f();
```

```
}
```

The virtual method **vf()** of the instance **c** is invoked three times. But this is not the same **vf()** in all cases. The application displays this:

```
C1::f: in constructor
C2::f: in method f()
C1::f: in destructor
```

Thus, invoking virtual methods in constructors and destructors is error-prone. This may even fail if the methods are pure virtual in the superclass.

Debugging hint: If an application fails with a message similar to "*pure virtual method was called*", look for invocations of virtual methods in constructors and destructors.

[Rule 154] All constructors shall properly initialize all its super-classes and member fields.

This is an application of [Rule 119]. All data shall be initialized.

[Rule 155] In the definition of a constructor, the field initializers must be in the same order as their declarations in the class declaration.

The order is significant since the field initializers can make a reference to each other. In the recommended paranoid warning mode, some compilers even reject the constructor definition when the field initializers are in a different order from the class declaration.

Example:

```
class C                 C::C(int x):
{                           a(0),
public:                     b(x),
    C(int x);               c(x + 1)
private:                {
    int a;                  ...
    int b;              }
    int c;
};
```

[Recommendation 156] If a constructor has only one argument or if the second and subsequent arguments have default values, this constructor shall be prefixed by the **explicit** keyword.

Invoking a constructor with one argument is implicitly used by the compiler to perform an automatic type conversion. In some cases, this is the right thing to do. But in most cases, this isn't. As a general rule, you must analyze the semantic of a constructor with one argument. Does it make sense to convert back and forth between the class of the constructor and the class of the first argument? If the answer is yes, then leave the constructor alone. If the answer is no, use **explicit**. In this context, the keyword **explicit** means that type conversions shall be explicit and the compiler will never user this constructor for implicit type conversions.

Example: In the following code, we declare two classes, City and House, which implement two different concepts. It does not make sense to implicitly convert a House into a City or vice-versa. But it is legitimate that an instance of House has a property of class City because a house

is located into a city. So, we provide a House constructor with an argument of type City, specifying the location of the constructed city.

```
class City { ... };

class House
{
public:
    explicit House(const City& location, const char* name = "") {}
};
```

Since the second argument of the constructor has a default value, it can be invoked with only one argument of type **City**. To avoid the accidental implicit conversion of a **City** into a **House**, we use the keyword **explicit**.

Let's examine the effect of the absence of **explicit** keyword using the following incorrect code.

```
void selectHouse(const House& h) { ... }

City paris;
selectHouse(paris);  // incorrect usage, we want to select a house, not a city
```

The function **selectHouse()** is probably used to select a specific existing house. So, calling **selectHouse()** with an argument of type **City** is incorrect and should be rejected by the compiler. Indeed, there is a compilation error here, thanks to the **explicit** keyword. Without this keyword, however, the compiler would implicitly use the constructor to convert the object of type **City** into **House** and **selectHouse()** is called with a new object with all default values in the city of **paris**. This is does not make sense.

Exception:

When the class of the constructor and the class of its first argument have equivalent semantics with different representations, it makes sense to allow conversions between the two types. The standard type **std::string** has a constructor with one argument of type **const char*** (i.e. a C string). The two types are semantically identical, they are character strings. So, converting between the two is both legitimate and useful. When a function uses a parameter of type **std::string**, it is useful to be able to call it with a string literal such as **"foo"** (which has type **const char***). The compiler implicitly creates a new object of type **std::string** from the string literal. This would not be possible if the **std::string** constructor had the keyword **explicit**.

Note: for this specific case, see also [Rule 140].

3.1.10 C++ operators

[Rule 157] Always provide an assignment operator with the same scope as the copy constructor.

The two operations, assignment operator and copy constructor, are closely related. They should be both enabled and implemented or both disabled.

Example:

Both enabled: Both disabled:

```
class C                              class C
{                                    {
public:                              private:
    // implementation required           // no implementation provided
    C(const C&);                         C(const C&) = delete;
    C& operator=(const C&);              C& operator=(const C&) = delete;
};                                   };
```

[Rule 158] When provided, the copy constructor and the assignment operator shall have consistent implementations.

The following two sequences shall have identical results.

Copy constructor:	Assignment operator:

`C a;`
`C b (a);`
	`C a;`
	`C b;`
	`b = a;`

But beware that, although *consistent*, the two implementations are usually *not identical*. The copy constructor works on an uninitialized object with no pre-existent state. The assignment operator, on the contrary, works on an initialized object which may need some clean-up before copy.

[Rule 159] Prevent self-assignment in an assignment operator.

A self-assignment "**a = a**" is a valid but void operation which must be filtered specifically.

Example:

```
class C
{
public:
    C& operator=(const C&);
    ...
};

C& C::operator=(const C& other)
{
    if (this != &other) {
        // implement actual assignment here
        ...
    }
    return *this;
}
```

Consider a class which encapsulates complex dynamic data structures. Assume that you decide that the semantics of the assignment operator is a deep copy. In the assignment operator, you must first free the previous content of the object and then duplicate the new content from the assigned value. If the test "**if (this != &other)**" is not present, the effect of the self-assignment would be a *reuse-after-free* bug.

[Rule 160] In the assignment operator of a derived class, make sure that the superclass fields are properly assigned using an explicit invocation to the superclass assignment operator.

Failing to do this may leave the superclass fields in an inconsistent state.

Example:

```
class D: public C
{
public:
    typedef C SuperClass;
    D& operator=(const D&);
    ...
};

D& D::operator=(const D& other)
{
    if (this != &other) {
        // assignment of superclass fields through
        // explicit invocation of superclass
        SuperClass::operator=(other);
        // implement actual assignment of subclass fields here
        ...
    }
    return *this;
}
```

[Rule 161] All assignment operators (=, +=, -=, etc.) shall return ***this**.

This is required to be consistent with the semantic of these operators.

[Rule 162] The comparison operators, when implemented in a class, must be consistent with each other.

If you implement the operator **==**, be sure to also implement **!=** and ensure that for any instances **a** and **b** of the class:

```
(a != b) == !(a == b)
```

Similarly, if you implement any of **<**, **<=**, **>** or **>=**, you must implement them all and ensure that for any instances **a** and **b** of the class:

```
(a < b)  == !(a == b) && !(a > b)
(a <= b) == !(a > b)
(a > b)  == !(a == b) && !(a < b)
(a >= b) == !(a < b)
```

In practice, it is only necessary to provide a deep implementation for operators **==** and **<**. All other operators can be implemented using references to the first two.

[Rule 163] When you do not use the final result of an expression using the increment (**++**) or decrement (**--**) unary operators, prefer the prefixed notation (**++i**, **--i**) to the postfix notation (**i++**, **i--**).

The postfix notation needs to create a temporary object holding the previous value of the object and use this temporary object as the result of the expression.

For integer or pointer types, this is usually optimized away by the compiler. But for more complex object classes which redefine these operators, the useless construction and destruction of the temporary object cannot be avoided. And this can cost some time for nothing.

This is especially the case for the iterators in the standard template library. The following example illustrates two functionally identical ways of walking through a list of **int** but the second way is uselessly slower.

```
std::list<int> x;

// Use ++i on iterator: good
for (std::list<int>::iterator i = x.begin(); i != x.end(); ++i) {
    ....
}

// Use i++ on iterator: identical but uselessly slow
for (std::list<int>::iterator i = x.begin(); i != x.end(); i++) {
    ....
}
```

3.1.11 C++ object management

[Rule 164] Never use **malloc()** and **free()** in C++, use the **new** and **delete** operators.

The C memory allocation routines return raw memory, not objects. Casting the result of **malloc()** to a pointer to an object instance is incorrect. This memory area is not a properly initialized object.

On the contrary, the C++ allocation operators manipulate objects. They invoke the proper constructors and destructors.

[Rule 165] Objects which were allocated using **new[ ]** must be deallocated using **delete[ ]**.

This is a really annoying feature of C++. The operators for allocating single objects and arrays are different and the corresponding delete operator must be used. But when a pointer is declared as **Foo***, there is no implicit indication how the pointed object was allocated. The developer must take care of this.

[Recommendation 166] Never use dynamically allocated arrays (operator **new[ ]**).

The preceding rule demonstrates that using the operator **new[ ]** may create confusion when it comes to deallocation. Moreover, [Recommendation 179] explicitly recommends avoiding arrays of C++ objects for other reasons, whether they are statically or dynamically allocated.

Use standard containers from the *C++ Standard Template Library* (STL) instead of dynamically allocated arrays. The standard container **std::vector**, for instance, is a much better alternative.

If you think that you need an array because you will pass it to a C routine which requires the address of an array, a **std::vector** is still better than a dynamically allocated array. The C++ Standard specifies than the underlying representation of the current content of a **std::vector** must match the representation of an array. Thus, the following sample code is both legal and safe.

```
// A C routine which expects an "array"
void legacyFunction(int* buffer, size_t intCount);

std::vector<int> v;
v.resize(100);  // or fill the vector the way you like
legacyFunction(&v[0], v.size());
```

[Rule 167] Check the size of an instance of **std::vector** before using the index operator [ ].

To reference an element in a vector, the C++ standard specifies that the method **std::vector::at()** performs bound checking while **std::vector::operator[]** does not. The difference is typically for performance reason. It is up to the developer to choose between safety and speed of code for each access.

Example:

```
int x = 0;
std::vector<int> v(4);  // declare a vector with 4 elements
x = v.at(6);            // throw exception std::out_of_range
x = v[6];               // read an invalid value in uninitialized memory
```

Some implementations[9] of the C++ STL add an assertion on the index value in **std::vector::operator[]**. This means that, in case of invalid index value, the application is aborted when compiled in debug mode (assertions on). In production mode, the assertions are off and the invalid index value is unnoticed.

This is normally fine. You should not use invalid index values anyway and the assertion helps the developer in debug mode. But this is not always the case. There are rare cases where using an invalid index is correct and the application should not fail in debug mode. See the following counter-example.

Counter-example:

```
void legacyFunction(int* buffer, size_t intCount);

std::vector<int> v;
legacyFunction(&v[0], v.size());
```

If the vector **v** is empty, **0** is an invalid index value and **v[0]** may fail in debug mode. However, this is valid code since **v.size()** is zero and any address value, even an incorrect one, is fine as first parameter for **legacyFunction()** because the function will not access the memory area anyway.

Be sure to identify that kind of usage and rewrite the code as follow:

```
legacyFunction(v.size() == 0 ? 0 : &v[0], v.size());
```

[9] This is the case for the Microsoft Visual C++ library.

[Rule 168] In function declarations, when a parameter is not an elementary type, not a pointer type or is a template type, always use a reference. If the contract is to not modify the object, use a const qualifier.

Not using a reference forces a copy of the object. This can be costly and this may even not compile in the case of an actual template parameter type without copy constructor.

Example:

Good code:

```
void f(const Foo& x);

template <typename T>
void g(const T& x);
```

Bad code:

```
void f(Foo x);   // force a copy of a Foo object

template <typename T>
void g(T x);     // call will not compile if actual type
                 // for T has no copy constructor
```

[Rule 169] When catching an exception, always use a reference to avoid a copy of the object.

This is an application of the preceding rule to the exception handling.

Example:

```
try {
    ....
}
catch (const std::exception& e) {  // reference to const exception object
    ....
}
```

[Rule 170] Multiple exception handlers in a try/catch structure must be ordered properly.

In a **try** / **catch** structure with multiple exception handlers, an exception is checked against all **catch** clauses until a matching one is found. Only the first matching one is used. So, when exception classes are derived from each other, the most specific **catch** clauses must come first. Otherwise, they are useless.

Example:

```
class A: public std::exception {...};
class B: public A {...};
```

Good code:

Bad code:

```
try {
    ...
}
catch (const B& e) {
    // catch all B exceptions
}
catch (const A& e) {
    // catch all A exceptions,
    // including subclasses of
    // A, except B which are
    // handled above
}
```

```
try {
    ...
}
catch (const A& e) {
    // catch all A exceptions,
    // including all subclasses of A
}
catch (const B& e) {
    // never used since B is a subclass of A
    // and was already caught in previous handler
}
```

[Rule 171] Use the *guard* design pattern to fail safely. Define guard classes for resources. Use destructors to fail safely.

It is sometimes necessary to reserve, lock or allocate a resource temporarily. This means that something shall be done at the end of a sequence of statements (release, unlock, deallocate). A problem occurs when an exception is thrown or a return is inadvertently executed in the middle of the sequence. In this case, the resource does not get cleaned up.

The *guard* design pattern is a technique to avoid this. For a given type of resource which needs to be locally reserved, define an associated guard class which has basically only two operations: a constructor which reserves the resource and a destructor which releases the resource (whatever *reserve* and *release* means for the given resource). For each sequence of code which reserves the resource, simply create a block of **{ }** where a local guard object is defined.

Example: This is how the guard pattern is used.

Unsafe code:

```
Resource myResource;
...
myResource.reserve();
...
// some exception or
// return here
...
myResource.release();
```

Equivalent safe code using a guard class:

```
Resource myResource;
...
{
    ResourceGuard(myResource); // implicit reserve()
    ...
    // some exception or return here
    ...
} // implicit release() even on exception or return
```

Example: This is how the guard pattern can be implemented.

```
class Mutex
{
public:
    void acquire() {...}
    void release() {...}
};

class MutexGuard
{
public:
    MutexGuard(Mutex& mutex): _mutex(mutex) {_mutex.acquire();}
    ~MutexGuard() {_mutex.release();}
private:
    Mutex& _mutex;
```

```
};
```

And this is how this implementation is used.

```
Mutex globalMutex;
...
{
    MutexGuard guard (globalMutex); // implicit globalMutex.acquire()
    ...
    // some exception or return here
    ...
} // implicit globalMutex.release() even on exception or return
```

Application to TSDuck: The classes **ts::Guard** and **ts::GuardCondition** are used to safely manage the mutexes and conditions.

[Rule 172] Use the *safe pointer* design pattern to prevent memory leaks and complex memory tracking.

When comparing C++ to Java, the main C++ nightmare is the memory management: when should I free memory?

A "safe pointer" (or "smart pointer") is a design pattern which replaces the usage of plain pointers by a template class which tracks all accesses to a resource. When no more active reference to the object exists, the object is automatically reclaimed.

Great care shall be taken when designing the safe pointer class (especially the multi-threading aspect). But once available, C++ memory management becomes almost as easy as Java.

Very large applications have proven to be memory-safe[10] when using safe pointers and zero explicit deallocation on any arbitrary large number of dynamically allocated objects with a limited life-time.

Warning: There are pathological cases where the safe pointer design pattern is ineffective. For instance, when two objects reference each other but are no longer referenced anywhere else, they are lost. They are not automatically freed by the safe pointers because references exist. They should be both freed at the same time. But this situation is typically the symptom of a poorly designed data model. So, the safe pointer design pattern removes the burden of the technical aspects of the memory management but not the requirement for a correct design!

Application to TSDuck: The template class **ts::SafePtr** is an implementation of safe pointer with optional thread-safe protection.

[Recommendation 173] Take care of reentrancy in reusable components. Use an abstract mutex interface and template reusable components.

If a class is not thread-safe, it is possible to make it thread-safe *on the user's request* without much effort and without performance penalty when thread-safety is not required.

Solution 1: Compile-time selection of the thread-safety model.

Define two mutex classes with identical services.

```
class NullMutex                         class RealMutex
{                                       {
public:                                 public:
```

[10] Based on *valgrind* results on Linux platforms.

```
    // acquire() and release() are
    // empty and inlined as no code
    void acquire() {}
    void release() {}
};
```

```
    // Implement acquire() and release()
    // in the .cpp file
    void acquire();
    void release();
};
```

The second class implements the actual locking features on one or more environments.

Now consider that the reusable component to implement is a safe pointer (see preceding rule). Define the safe pointer class as follow:

```
template <typename T, class MUTEX> class SafePointer {...};
```

In the implementation of the **SafePointer** class, define an object of the generic type **MUTEX** to implement the locking.

Whenever you need a thread-safe or non-thread-safe pointer to objects of class **Foo**, use one of the following:

```
SafePointer <Foo, RealMutex> threadSafePointer;
SafePointer <Foo, NullMutex> nonThreadSafePointer;
```

The first pointer class uses thread-safe code while the second one uses fast code (the inline empty locking services are optimized away by the compiler). There is exactly zero overhead for the non-thread-safe version but the two versions share the same source code.

Solution 2: Run-time selection of the thread-safety model.

Define a general abstract mutex interface which declares the basic synchronization services as pure virtual functions:

```
class MutexInterface
{
public:
    virtual void acquire() = 0;
    virtual void release() = 0;
    virtual ~MutexInterface() {}
};
```

Define two subclasses **NullMutex** and **RealMutex**. The first one implements inline empty services which do nothing. The second one implements the actual locking features on one or more environments.

A reusable component can, based on some run-time condition, allocate either a **NullMutex** instance or a **RealMutex** instance. In the non-thread-safe case, the **acquire()** and **release()** operations are simple indirect calls to an empty routine which returns immediately.

Comparison: Performances of the two versions are almost identical in the thread-safe case. The solution 1 offers the best performance in the non-thread-safe case. But it is slightly more constraining: the selection is done at compile-time, the mutex guard class must be a template one and the code footprint is larger when both thread-safe and non-thread-safe usages of the same reusable component are present in the same application.

The recommended trade-off is to use the **MutexInterface** with a template reusable component.

[Rule 174] Do not use C-style casts **(type).** Use C++ cast operators **reinterpret_cast**, **dynamic_cast**, **static_cast** and **const_cast**.

The C++ cast operators provide a better control over which type of casting you want. More appropriate checks are performed by the compiler or even at run-time (**dynamic_cast**). The code is also more readable, your intention is more clearly explained to the maintainer.

The most general cast operator is **reinterpret_cast** and is equivalent to a C-style cast. There is no check at all. It is dangerous and most of the time inappropriate. Usually, we need to cast between related types and **static_cast** or **dynamic_cast** is a better option.

Beware of **const_cast** when it is used to remove the "*constness*" of an object. By doing so, you break the API contract if the **const** object is provided by your caller. Is the **const_cast** required because you invoke another API which is badly defined (an argument is not declared as **const** but not modified anyway)? Or is the **const_cast** required because you will actually modify the object? While the former situation is legitimate, the latter is not.

[Recommendation 175] Use **dynamic_cast** to check a subclass type.

If a general method working on a super-class needs some specific processing when the object belongs to a given sub-class, this is usually the symptom of a bad design. A virtual method should be defined in the super-class.

But sometimes it is not possible to modify the super-class and it is necessary to use specific code. In this case, **dynamic_cast** is the only reliable and safe way to test the sub-class of the object.

Example:

```
class C {...};
class C1 : public C {...};
class C2 : public C {...};

void f(const C& c)
{
    // generic processing on super-class view “c” of the object
    c.someMethod();

    // is this object an instance of C2?
    const C2* p2 = dynamic_cast<const C2*>(&c);
    if (p2 != 0) {
        // yes, the object is an instance of C2
        // specific processing on C2 sub-class view “*p2” of the object
        p2->someMethodOfC2();
    }
}
```

Note 1: Be aware that **dynamic_cast** is allowed only if the super-class is polymorphic, i.e. if it has at least one virtual method. A virtual destructor is sufficient.

Note 2: Using **dynamic_cast** requires that the C++ compiler supports *Run-Time Type Information* (RTTI). All modern C++ compilers on desktop and server platforms support RTTI. But, on some constrained embedded platforms, the C++ compiler may not support RTTI and **dynamic_cast** is rejected by the compiler. In that case, you have to determine the actual subclass by some other way and then use **static_cast** after verifying the actual subclass type.

[Rule 176] Do not use **exit()**.

In C++, **exit()** only guarantees the proper destruction of global objects. The destructors of pending local objects are not invoked. If those destructors have external effects (flushing a cache, closing a file, etc.), the state of some external resources may not be consistent.

To perform an early exit of the program, throw an exception. Use a dedicated application-defined exception and ensure that no function catches this exception (beware of "**catch (...)**" structures). In multi-threaded applications, this is a little bit more complicated; you need to synchronize the termination of all threads.

In the main program, to return an error code to the operating system, do not call **exit()**, perform a **return** instruction with the appropriate exit code.

[Rule 177] Do not use **va_start**, **va_arg** and **va_end** on class types.

These macros were designed for C and not C++. They do not properly invoke constructors.

In C++, variable argument lists are handled in a much safer way using **std::initializer_list**.

Application to TSDuck: The class **ts::ArgMix** and its subclasses are designed to transparently support type-safe heterogeneous variable argument lists. Sample usages can be found in methods **ts::UString::Format()** and **ts::UString::scan()**.

[Rule 178] Never declare a function with an argument being an array of objects when subclasses exist for this class.

This practice has very dangerous side effects on subclasses. This introduces bugs which are very hard to track.

Example:

```
class A
{
public:
    int a;
    A(): a(1) {}
};

class B: public A
{
public:
    int b;
    B(): b(2) {}
};

void f(A array[], size_t elemCount)
{
    // update second element of array
    if (elemCount >= 2) {
        array[1].a = 47;
    }
}

B x[2];
f(x, 2);  // do you think that x[1].a is updated ?
```

```
std::cout << x[0].a << ", " << x[0].b << ", "
          << x[1].a << ", " << x[1].b << std::endl;
```

The last instruction prints "**1, 47, 1, 2**". The function **f()** has corrupted the subclass fields of **x[0]**.

The same effect is obtained when the function is defined using a pointer instead of an array:

```
void f(A* array, size_t elemCount)
```

Alternatives: If you really need that function, you must overload it for all possible subclasses of **A**. And if new subclasses of **A** are created in the future, you must remember to overload the function for all new subclasses. Since this is a maintenance challenge, this kind of function should really be avoided anyway.

[Recommendation 179] Avoid using arrays of C++ objects. Use standard containers from the STL instead.

The preceding rule is a good illustration of the danger of arrays of objects in C++.

There is almost no case where using an array is better than using the standard container **std::vector**, neither in performance nor in functionalities.

[Rule 180] Never assign objects through dereferencing a pointer to a base class.

The assignment operator cannot be virtual (at least in its strict definition). By using the assignment through dereferencing a pointer to a base class, the actual assignment operator is the one from the superclass. The assignment of the object will be partial (*slicing effect*) and the object can be left in an inconsistent state.

Example:

```
class Fruit
{
public:
    Fruit& operator=(const Fruit&);
    ...
};

class Apple: public Fruit
{
public:
    Apple& operator=(const Apple&);
    ...
};

class Tomato: public Fruit
{
public:
    Tomato& operator=(const Tomato&);
    ...
};

Apple  apple;
Tomato tomato;

Fruit* p1 = &apple;
```

```
Fruit* p2 = &tomato;

*p1 = *p2;  // Legal but WRONG !
```

The above assignment is legal and compiles. However, since the assignment operator cannot be virtual, it invokes the assignment operator of the superclass, i.e. the method **Fruit::operator=(const Fruit&)**. In practice, the common **Fruit** fields of an **Apple** object will be assigned with the common **Fruit** fields of a **Tomato** object. The specialized **Apple** fields of the object are left unmodified and, thus, possibly inconsistent with the new values of the common **Fruit** fields.

Honestly, this is weird to assign a tomato into an apple, even if both are fruits!

We reach here the limitation of the virtual vs. non-virtual method model. This model works fine as long as the virtual / non-virtual method works on one object only (**this** object). But when the method is designed to work on two objects globally, as it is the case for the assignment, there is no good solution.

3.2 C++ toolset

This section lists rules and recommendations in the configuration of the various usual C++ toolsets.

3.2.1 C++ compilers

[Rule 181] Use the hereby documented set of compilation options which enforces rules from this document. Additional options can be locally added for debug or performance reasons but shall not lower the security or strictness of the mandatory options.

The options below enforce the following rules for various compilers:

- [Rule 51] Use the most paranoid "warning mode" of the compiler and fix all warnings without exception.
- [Rule 52] Turn compilation warnings into errors.

Additional security checks are added when available (**-fstack-protector-all** with GCC for instance).

GCC as C++ compiler:

```
-std=c++11
-fstack-protector-all
-fno-strict-aliasing
-Werror
-Wall
-Wextra
-Wpedantic
-Wformat-nonliteral
-Wformat-security
-Wswitch-default
-Wuninitialized
-Wshadow
-Wfloat-equal
-Wundef
-Wpointer-arith
-Wcast-align
-Woverloaded-virtual
-Wctor-dtor-privacy
-Wnon-virtual-dtor
-Woverloaded-virtual
```

```
-Wsuggest-override
-Wsign-promo
-Wzero-as-null-pointer-constant
-Weffc++
-Wsign-promo
-Wstrict-null-sentinel
-Wno-unused-parameter
```

Note that the naming of the GCC options is misleading. The option **–Wall** does not enable all warnings. Extra warnings are enabled using the option **–Wextra**.. But even this option does not enable everything. This is why additional individual **–W** must be specified.

Microsoft C++ compiler from the command line:

```
/GS /WX /W4
```

Note that the option **/Wall** which enables all possible warnings is extremely aggressive and most predefined Microsoft system headers do not even pass the compilation with this option. This is why we limit the warning reporting to **/W4**, the immediately preceding level.

Microsoft Visual Studio for C++, set of properties:

```
<ClCompile>
  <BufferSecurityCheck>true</BufferSecurityCheck>
  <TreatWarningAsError>true</TreatWarningAsError>
  <WarningLevel>Level4</WarningLevel>
</ClCompile>
```

This piece of XML is typically used in property sheets and project properties. It is equivalent to the Microsoft C/C++ compiler command line options above.

Application to TSDuck: These options are enforced in the common Makefile (Linux, macOS) and in the common Visual Studio property sheets.

3.2.2 Makefiles

[Rule 182] Do not try to explicitly enumerate the header file dependencies of C++ files. Setup an automatic dependency resolution system.

It is difficult to manually maintain such dependencies. Include directives are added and removed from nested header files and there is no reliable way to manually maintain this on the long run. This can lead to subtle bugs when a module is not recompiled simply because its dependency on a recently updated header file was missing.

The idea is to automatically generate a text file **module.dep** which contains a makefile dependency rule for each **module.cpp**. The **module.dep** file lists all header files which are directly or indirectly included by **module.cpp**. Generating such a file is a feature of the C preprocessor (at least with GCC).

All **.dep** are automatically included in the makefile. The makefile automatically regenerates obsolete **.dep** before including them. This is possible using the GCC compiler and GNU Make.

Other building systems such as cmake, qmake, MSBuild (Microsoft Visual Studio) implement their own automatic dependency resolution system. Use the one that suits you or use the above method with plain makefiles, but do not try to maintain the dependencies manually.

Application to TSDuck: The automatic generation of **.dep** files is implemented in the common Makefile. For Visual Studio, the dependencies are automatically handled.

3.2.3 Unit testing

[Recommendation 183] Use the CppUnit framework for unitary tests of C++ modules.

The [Rule 57] mandates the usage of a unitary testing. CppUnit can be used for C++ code.

Based on command line options, the main executable of a unitary tests suite shall outputs the results either in text mode on the standard output (by default) or in an XML file. The resulting XML file can be used by a continuous integration framework for automated non-regression tests.

Hint: Use a dedicated library on top of CppUnit to simplify the development of the unitary tests.

Application to TSDuck: All unitary tests are implemented on top of CppUnit (see directory **src/utest**).

3.2.4 Doxygen self-documentation

[Rule 184] Doxygen shall be used to document all C++ header files. The Doxygen tags shall document the file itself (tag *@file*) and all the public entities which are declared in the header (functions, classes, namespaces, macro definitions, constants, etc.)

The header files define the public entities of the various modules and the relationships between them. It is essential that they are documented.

[Recommendation 185] Doxygen may be used to document the internals of the C++ source files (*.cpp* files).

Documenting the internals of the modules can help the maintenance of these modules.

There are downsides however. By publishing the internal documentation of a library, the users are aware of details they should not know.

Application to TSDuck: The TSDuck library contains most of the code. All header files in this library (directory **src/libtsduck**) are documented using Doxygen and published on the TSDuck web site. The internals of the library as well as the command line wrappers and plugins do not use Doxygen.

[Rule 186] If Doxygen is used to document the internals of the C++ source files, the corresponding documentation shall remain internal to the corresponding development team. It shall not be made available to the users. Two sets of documentation shall be produced: a public one with the header files only and an internal one with all source files.

This is the consequence of the preceding recommendation.

To generate two different versions of the documentation, use two distinct Doxygen configuration files (the **Doxyfile**). The configuration file for the public version is identical to the internal one (it may even include it using an **@INCLUDE** directive) but it must add the following rule:

```
EXCLUDE_PATTERNS += *Template.h *.c *.cpp
```

[Rule 187] When an entity is in a Doxygen-documented source file, all its components shall be documented without exception.

The brief description of the entity (**@brief**) is mandatory. The detailed description (**@details**) is optional if the entity is so simple that the brief description is sufficient.

Functions and methods shall document all their parameters and their returned value (if any).

Enumeration types shall document all their values individually in addition to the documentation of the type.

Hint 1: The mandatory documentation of the function parameters can be enforced using the following directive in the **Doxyfile**:

```
WARN_NO_PARAMDOC = YES
```

Hint 2: The **@brief** and **@details** commands do not need to be explicitly present. The first item is implicitly the brief description and the detailed description is implicitly everything that follows a blank line after the brief description. The implicit notation is more readable in the source code for developers and produces the same documentation.

Example: The following two Doxygen blocks are equivalent.

```
//!
//! @brief This is the brief description of the next item.
//! @details This is the long and detailed description.
//! The Doxygen commands can be omitted if a blank line
//! separates the brief and detailed description.
//!

//!
//! This is the brief description of the next item.
//! This is the long and detailed description.
//! The Doxygen commands can be omitted if a blank line
//! separates the brief and detailed description.
//!
```

This behavior is allowed when the **Doxyfile** contains the following directive:

```
JAVADOC_AUTOBRIEF = YES
```

Hint 3: Individual parameters or values can be documented on one line after the element itself if the documentation contains only a brief description.

Example: The following two notations are equivalent. Note that the first one is more compact and probably more readable.

```
//!
//! Flags for the @c Hexa family of functions
//!
enum HexaFlags {
    HEX_HEXA  = 0x0001,  //!< Dump hexa values
    HEX_ASCII = 0x0002,  //!< Dump ascii values
    ....
};

//!
//! Flags for the @c Hexa family of functions
//!
enum HexaFlags {
```

```
        //!
        //! Dump hexa values
        //!
        HEX_HEXA  = 0x0001,
        //!
        //! Dump ascii values
        //!
        HEX_ASCII = 0x0002,
        ....
    };
```

3.3 C++ coding conventions

This section is present to fulfil the required separation of immutable rules and recommendations from potentially replaceable conventions, as explained in 1.5.

3.3.1 Source code formatting

[Convention 188] The file name extensions by file type are:

- C++ header file: .h
- C++ source file: .cpp

For C++ files, several conventions exist: .h, .hpp, .hxx, .H for headers, .cpp, .cxx, .C for source files. The selected convention has a large adoption and is good enough.

[Convention 189] Indentation: use 4 space characters without tabulation.

Using less than 4 spaces is not clear enough. Using 8 spaces moves too fast to the right.

3.3.2 Modularity

[Rule 190] In large organizations, the allocation of public prefixes is centralized into one unique document.

The centralization of the prefix allocation avoids name clashes within the same organization. Developers are encouraged to use short but meaningful prefixes (4 or 5 characters maximum). Using acronyms is acceptable.

The assignment of the person or team in charge of the allocation of prefixes is beyond the scope of this document.

[Convention 191] When a module is specialized in the management of one data type (object-oriented design), the base name of the module files is the data type name, using the same lower/upper case letters.

Example: The class **ts::Data** is declared in file **tsData.h** and defined in file **tsData.cpp**.

[Convention 192] The file name of a module containing a C++ class is built from the concatenation of all nested namespaces and the class name.

This way, the class declaration can be easily found.

Example: The class named **ts::proj::MyClass** is declared in file **tsprojMyClass.h** and defined in file **tsprojMyClass.cpp**.

[Convention 193] Non-inline template methods or methods of template classes are grouped into one single header file. This template header file has the same base name as the main header file with *Template* suffix. The template header file is included from the main header file.

The portability of the C++ templates is a challenge. Most compilers require the definition of the template methods to be available during the compilation of the modules which *use* them. So we need to have them in a header file. But mixing them in the same header file as the one which declares the interface of the module is not a good design.

Example: Header file named **tsModule.h** with at least one template method:

```
#pragma once

namespace ts {
    class Module
    {
    public:
        void plain(const std::string&);

        template <class C>
        void generic(const C&);
    };
}
#include "tsModuleTemplate.h"
```

The definitions of template methods are in header file **tsModuleTemplate.h**:

```
#pragma once

template <class C>
void ts::Module::generic(const C& a)
{
    ...
}
```

And here is the implementation file **tsModule.cpp**:

```
#include "tsModule.h"

void ts::Module::plain(const std::string& a)
{
    ...
}
```

3.3.3 Naming conventions

There are many naming conventions in the C and C++ communities. This is sometimes both the most sensitive part for the developers and the less important for the quality of the code. Discussing which

naming convention is the best one is a waste of time. There is no *best* one. But many are good enough. The quality of the code only depends on the strict and consistent application of one single good enough naming convention. This section describes the naming conventions for the C and C++ languages in the TSDuck project. Simply use them.

[Convention 194] All entities which are defined within the company shall be declared within the namespace named "`ts`".

This is a trade-off between readability and usability. Using a more signification but longer namespace such as **tsduck** would appear as painful to developers who could be tempted to disobey the "no *using namespace* directive" rule.

[Rule 195] Multiple nested namespaces may be defined within the namespace **ts**. Typically, there is one inner namespace per project or logical group of classes. The allocation of inner namespaces is centralized into one unique document.

Using inner namespaces increases the independencies of development groups. Centralization avoids namespace clashes.

Developers are encouraged to use short but meaningful namespaces (4 or 5 characters maximum). Using acronyms is acceptable.

[Convention 196] Namespaces are composed of lowercase letters or digits only.

No capital letter, no underscore.

As mentioned in the preceding rules, all entities are in the namespace **ts** or in one of its inner namespaces.

[Convention 197] Preprocessing macro names (**#define**) are composed of uppercase letters and digits only. Words are separated by underscores. All macro names start with the prefix "**TS_**".

Example:

```
#define TS_FOO_VERSION 47
#define TS_INCR(x) (...)
```

[Convention 198] Use a verb as the central component of the name for functions that do something. Use an imperative form for the name of functions returning a boolean value. Such functions typically start with "**is**" or "**has**", depending on what they return.

Example:

```
namespace ts {
    class Foo
    {
    public:
        Foo(...);
        ~Foo();
        void save(...);
```

```
            bool isEmpty() const;
            bool hasChildren() const;
        private:
            // ... internal fields
        };
    }
```

[Convention 199] Class names, type names, public static functions, non-member (global) functions use a mixed case convention, without underscore, starting with an uppercase letter.

See example below.

[Convention 200] Non-static public member fields and functions, local variables (in functions) use a mixed case convention, without underscore, starting with a lowercase letter.

When reading code, it is useful to differentiate static and non-static members. Static members are class-wide; their name starts with an uppercase letter. Non-static members applies to one instance; their name starts with an lowercase letter.

See example below.

[Convention 201] Public constants use uppercase letters with underscores to separate words.

See example below.

[Convention 202] Private entities of any sort use the same convention as their public counterpart but start with an underscore.

When reading code, it is important to understand the scope of an action, public or private. Modifying a public field may break the interface contract of the class while modifying a private field only requires a local analysis. Private fields are identified by their leading underscore.

Example:

```
    namespace ts {
        void GlobalFunction();                    // global function, not in a class
        typedef Foo* FooPtr;                      // type name
        class ClassName                           // class name
        {
        public:
            static const size_t MAX_SIZE = ...;   // constant
            int somePublicField;                  // public member field
            void someMemberFunction();            // public member function
            static void SomeStaticFunction();     // public static function
        private:
            int _somePrivateField;                // private member field
            void _somePrivateMemberFunction();    // private member function
            static void _SomePrivateStaticFunc(); // private static function
        };
```

```
}

void ts::ClassName::someMemberFunction()
{
    uint32_t messageIndex;                    // local variables
}
```

[Convention 203] Local variables (in functions) use a mixed case convention, without underscore, starting with a lowercase letter.

Example:

```
void tsFoo(void) {
    uint32_t messageIndex;  // local variables
    ...
}
```

Note: Not applied everywhere in TSDuck because of a heavy legacy.

[Convention 204] In all *mixed case* conventions, uppercase letters are used at the beginning of a word exclusively. Digits can be used. Acronyms are considered as words; i.e. they start with one uppercase letter followed by lowercase letters.

Example:

```
typedef uint16_t TcpOrUdpPort;
TcpOrUdpPort tcpDefaultPort = 80;
static TcpOrUdpPort _udpDefaultPort = 80;
```

[Convention 205] Values in **enum** types are composed of uppercase letters and digits only. Words are separated by underscores. A common prefix which is derived from the type name is prepended to all values.

Example:

```
namespace ts {
    enum Counter {
        COUNTER_ONE,
        COUNTER_TWO
    };
}
```

[Convention 206] Conditional compilation of debug code is allowed using the symbol **DEBUG**. When **DEBUG** is undefined, no debug code shall be compiled.

This symbol shall not be defined in any header file. It shall be defined by the compilation environment (makefile, IDE compilation profile, etc.)

As an exception, the macro **DEBUG** does not start with the required **TS_** prefix but **DEBUG** is a common symbol which is recognized by many libraries to trigger debug-specific code.

3.3.4 Syntax formatting conventions

Just like naming conventions, the syntax formatting conventions are subject to discussion. And, similarly, there no best one. We must simply adopt one and use it consistently in all code.

Most IDE's can be configured to automatically enforce these settings when typing code. This is possible with Emacs, Eclipse, Microsoft Visual Studio or Qt Creator.

[Convention 207] Add a space character in the following locations:

- Before and after binary or ternary operators
- Before a group of one or more '(' in an expression
- After a group of one or more ')' or ']' in an expression
- After a ','

Exception: no space before ',' or ';'.

Example:

```
a = (b + c) * ((e / 4) % 3);
x = y > size ? y : size;
p = f(a, b, c[i] + 1);
```

[Convention 208] Do not use any space character before '(' in a function declaration, definition or call. Similarly, do not use any space character before '[' in an array declaration or reference.

Example:

```
void foo(char* name, size_t size);
char buffer[200];
buffer[0] = 'a';
foo(buffer, sizeof(buffer));
```

[Convention 209] Comment lines must be aligned on the code they refer to.

Example:

Good code:

```
// about f
void f(int x)
{
    // about the test
    if (x > 1) {
        // about the zero
        x = 0;
    }
}
```

Bad code:

```
// about f
void f(int x)
{
// about the test
    if (x > 1) {
            // about the zero
        x = 0;
    }
}
```

[Convention 210] Multi-lines comments shall be formatted as follow:

```
// This is a comment
// on several lines.
```

Exception: For self-documentation (Doxygen for instance), use the required conventions for the corresponding documentation extraction tool.

[Convention 211] The function definition has its initial "{" on a new line. The parameters are listed on one line, if this makes the line not too long.

Example:

```
void FunctionCode(const char* name, size_t size)
{
    ...
}
```

[Convention 212] In a function declaration, definition or call, when the parameter list is too long to fit on one line, the parameters are aligned on the opening parenthesis and there must be exactly one parameter per line.

Example:

```
MyFunction("abcd", size, 47);  // short line

MyFunction(tsCallingSomethingVeryVeryLongAndUnreadable(a + 45 * x, "title"),
           size,
           47);    // long statement: one parameter per line
```

In the first example, using one parameter per line would be counter-productive. The code would be bloated and less readable. So, if everything fit on one line, use one line.

In the second example above, putting the two parameters "**size, 47**" on the same line could make the non-cautious reader think that there are only two parameters to the function.

So, either put all parameters on one line or one parameter per line, but not a mixture of the two.

[Convention 213] The conditional and loop statements have their initial '{' on the same line. The closing '}' is on its own line. The "**else if**" construction is on one line.

Example:

```
if (x == a) {
    ...
}
else if (y < b) {
    ...
}
else {
    ...
}
```

[Convention 214] The **switch** statement has its initial '{' on the same line. The various **case** entries and the optional **default** entry are on individual lines. They are indented from the **switch**.

Example:

```
switch (state) {
    case TS_START:
        print("started");
        break;
    case TS_CONTINUE:
    case TS_END:
        print("ok");
        break;
    default:
        print("error");
        break;
}
```

[Convention 215] The namespace declaration has its initial '{' on the same line. The closing '}' is on its own line. The content of the namespace is indented.

Example:

```
namespace ts {
    namespace proj {
        void foo();
    }
}
```

[Convention 216] The class declaration has its initial '{' on a new line. The **public**, **protected** and **private** keywords are aligned on the '{'.

Example:

```
class C
{
public:
    void init();
private:
    void _reset();
};
```

Remember that the closing "}" of a class declaration must be followed by a ";". This is a typical C++ oddity and a common source of compilation syntax errors.

[Convention 217] In template class declarations and definitions, the **template** part is on a separate line with the same alignment as the declaration or definition.

Example:

```
template <typename T, class MUTEX>
class SafePointer
{
```

```
    ...
};

template <typename T, class MUTEX>
ts::SafePointer<T,MUTEX>& operator=(T* p)
{
    ...
}
```

[Convention 218] In the definition of a constructor, the field initializers are indented and separated one per line.

Example:

```
class C
{
public:
    C(int x);
private:
    int _a;
    int _b;
    int _c;
};

C::C(int x):
    _a(0),
    _b(x),
    _c(x + 1)  // be careful, no comma on last field initializer
{
    ...
}
```

[Convention 219] If a sequence of stream output operators (**<<**) is too long to fit on one line, the **<<** operators are aligned on multiple lines.

Example:

```
std::cout << "title: " << title
          << ", message: " << message
          << std::endl;
```

3.3.5 Doxygen self-documentation

[Convention 220] All Doxygen commands in embedded comments in source files shall be introduced with the "**@**" character (e.g. **@file**, **@param**, **@return**, etc.)

There are two syntaxes for Doxygen commands, starting with a "**@**" or with a "****". For consistency, all developers shall use one single common convention. The syntax using "**@**" is preferred since it is compatible with Javadoc.

[Convention 221] In C++ source files, the Doxygen comment blocks shall use following notation:

```
//!
//! ... Doxygen documentation commands
//!
```

There are two main syntaxes for Doxygen comment blocks in C++, one using the C-style comment "**/*" and one using the C++-style comment "**//**". For consistency, all developers shall use one single common convention. The [Rule 81] explains why single-line C++-style comments are safer than multi-line C-style comments.