(/)

Develop (/develop/)

Release process (/develop/release-process/)

Roadmap (/develop/roadmap/)

Maintainers (/develop/maintainers/)

Browse code (/develop/browse-code/)

Contributing code (/develop/contributing-code/)

Code reviews (/develop/contributing-code/code-reviews/)

Coding style (/develop/contributing-code/coding-style/)

Get ready (/develop/contributing-code/get-ready/)

Licensing (/develop/contributing-code/licensing/)

Out of tree (/develop/contributing-code/out-of-tree/)

Submit (/develop/contributing-code/submit/)

Supporting material (/develop/contributing-code/supporting-material/)

Tools (/develop/tools/)

Home (/) > Develop (/develop/) > Contributing Code (/develop/contributing-code/) > Coding style (/develop/contributing-code/coding-style/)

Coding style

The bulk of hours spent working on code is in some form of maintenance (often not performed by the initial author). There are many disparate programming and commenting styles in use today, however. This implies a need for a common set of conventions so that maintainers do not have to learn, recognize and adapt to a constantly changing code base.

Since the maintainer of the code is often not the author, extensive commenting is very important in order to allow a maintainer to quickly come up to speed on the design, conceptual models and implementation of the code. A common coding and commenting scheme can make for easier to read, easier to understand, easier to maintain and more bug-free code.

When writing code for ns-3, we therefore ask that you follow these relatively widely accepted coding standards. This document thus outlines guidelines we plan to enforce on each component integrated in ns-3 to ensure uniformity of the codebase which is a first step towards readable code.

The following recommendations are not strict rules and some of them are conflicting but the point here is to outline the fact that we value more common-sense than strict adherence to the coding style defined in this document.

A lot of the syntactical rules described in this document can be easily enforced with the help of the *utils/check-style.py* script which relies on a working version of uncrustify (http://uncrustify.sourceforge.net/) so, do not hesitate to run this script over your code prior to submission.

Code layout

The code layout follows the GNU coding standard (http://www.gnu.org/prep/standards/) layout for C and extends it to C++. Do not use tabs for indentation. Indentation spacing is 2 spaces (the default emacs C++ mode) as outlined below:

```
void           (/)
Foo (void)
{
  if (test)
    {
      // do stuff here
    }
  else
    {
      // do other stuff here
    }

  for (int i = 0; i < 100; i++)
    {
      // do loop
    }

  while (test)
    {
      // do while
    }

  do
    {
      // do stuff
    } while ();
}
```

Each statement should be put on a separate line to increase readability, and multi-statement blocks following conditional or looping statements are always delimited by braces (single-statement blocks may be placed on the same line of an if () statement). Each variable declaration is on a separate line. Variables should be declared at the point in the code where they are needed, and should be assigned an initial value at the time of declaration. Except when used in a switch statement, the open and close braces “{” and “}” are always on a separate line. Do not use the C++ “goto” statement.

The layout of variables declared in a class may either be aligned with the variable names or unaligned, as long as the file is internally consistent and no tab characters are included. Examples:

```
int varOne;
double varTwo;   // OK (unaligned)
int      varOne;
double   varTwo;  // also OK (aligned)
```

Naming (/)

Name encoding

Function, method, and type names should follow the CamelCase (http://en.wikipedia.org/wiki/CamelCase) convention: words are joined without spaces and are capitalized. For example, “my computer” is transformed into MyComputer. Do not use all capital letters such as MAC or, PHY, but choose instead Mac or Phy. Do not use all capital letters, even for acronyms such as EDCA: use Edca instead. This applies also to two-letter acronyms, such as IP (which becomes Ip). The goal of the CamelCase convention is to ensure that the words which make up a name can be separated by the eye: the initial Caps fills that role. Use PascalCasing (CamelCase with first letter capitalized) for function, property, event, and class names.

Variable names should follow a slight variation on the base CamelCase convention: camelBack. For example, the variable “user name” would be named “userName”. This variation on the basic naming pattern is used to allow a reader to distinguish a variable name from its type. For example, “UserName userName;” would be used to declare a variable named userName of type UserName.

Global variables should be prefixed with a “g_” and member variables (including static member variables) should be prefixed with a “m_”. The goal of that prefix is to give a reader a sense of where a variable of a given name is declared to allow the reader to locate the variable declaration and infer the variable type from that declaration. Defined types will start with an upper case letter, consist of upper and lower case letters, and may optionally end with a “_t”. Defined constants (such as static const class members, or enum constants) will be all uppercase letters or numeric digits, with an underscore character separating words. Otherwise, the underscore character should not be used in a variable name. For example, you could declare in your class header my-class.h:

```
typedef int NewTypeOfInt_t;
const uint8_t PORT_NUMBER = 17;

class MyClass
{
  void MyMethod (int aVar);
  int m_aVar;
  static int m_anotherVar;
};
```

and implement in your class file my-class.cc:

```
int MyClass::g_anotherVar = 10;
static int g_aStaticVar = 100;
int g_aGlobalVar = 1000;
void
MyClass::MyMethod (int aVar)
{
  m_aVar = aVar;
}
```

As an exception to the above, the members of structures do not need to be prefixed with an “m_”.

Finally, do not use Hungarian notation (http://en.wikipedia.org/wiki/Hungarian_notation), and do not prefix enums, classes, or delegates with any letter.

Choosing names

Variable, function, method, and type names should be based on the English language, American spelling. Furthermore, always try to choose descriptive names for them. Types are often english names such as: Packet, Buffer, Mac, or Phy. Functions and methods are often named based on verbs and adjectives: GetX, DoDispose, ClearArray, etc.

A long descriptive name which requires a lot of typing is always better than a short name which is hard to decipher. Do not use abbreviations in names unless the abbreviation is really unambiguous and obvious to everyone (e.g., use “size” over “sz”). Do not use short inappropriate names such as foo, bar, or baz. The name of an item should always match its purpose. As such, names such as “tmp” to identify a temporary variable or such as “i” to identify a loop index are ok.

If you use predicates (that is, functions, variables or methods which return a single boolean value), prefix the name with “is” or “has”.

File layout and code organization

A class named MyClass should be declared in a header named my-class.h and implemented in a source file named my-class.cc. The goal of this naming pattern is to allow a reader to quickly navigate through the ns-3 codebase to locate the source file relevant to a specific type.

Each my-class.h header should start with the following comments: the first line ensures that developers who use the emacs editor will be able to indent your code correctly. The following lines ensure that your code is licensed under the GPL, that the copyright holders are properly identified (typically, you or your employer), and that the actual author of the code is identified. The latter is purely informational and we use it to try to track the most appropriate person to review a patch or fix a bug. Please do not add the “All Rights Reserved” phrase after the copyright statement.

```
/* -*- Mode: C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) YEAR COPYRIGHTHOLDER
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 * Author: MyName <myemail@example.com>
 */
```

Below these C-style comments, always include the following which defines a set of header guards (MY_CLASS_H) used to avoid multiple header includes, which ensures that your code is included in the “ns3” namespace and which provides a set of doxygen comments for the public part of your class API. Detailed information on the set of tags available for doxygen documentation is described in the doxygen website (http://www.doxygen.org).

```
#ifndef MY_CLASS_H
#define MY_CLASS_H

namespace n3 {

/**
 * \brief short one-line description of the purpose of your class
 *
 * A longer description of the purpose of your class after a blank
 * empty line.
 */
class MyClass
{
public:
 MyClass ();
 /**
  * \param firstParam a short description of the purpose of this parameter
  * \returns a short description of what is returned from this function.
  *
  * A detailed description of the purpose of the method.
  */
 int DoSomething (int firstParam);
private:
 void MyPrivateMethod (void);
 int m_myPrivateMemberVariable;
};

} // namespace ns3

#endif /* MY_CLASS_H */
```

The my-class.cc file is structured similarly:

```
/* -*- Mode: C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) YEAR COPYRIGHTHOLDER
 *
 * 3-paragraph GPL blurb
 *
 * Author: MyName <myemail@foo.com>
 */
#include "my-class.h"

namespace ns3 {

MyClass::MyClass ()
{}

...

} // namespace ns3
```

Language features

As of ns-3.25, ns-3 had avoided C++-11 (and C++-14) features until compiler support was mature. Starting with ns-3.26, we will permit the use of C++-11 features supported (https://gcc.gnu.org/gcc-4.8/cxx0x_status.html) by our minimum compiler supported (gcc-4.8, clang-3.3). Starting with ns-3.27, we require at least gcc-4.9 instead of gcc-4.8.

If a developer would like to propose to raise this bar to include more features than this, please email the developers list. We will move this language support forward as our minimally supported compiler moves forward.

Comments

The project uses Doxygen (http://www.doxygen.org) to document the interfaces, and uses comments for improving the clarity of the code internally. All public methods should have Doxygen comments. Doxygen comments should use the C comment style. For non-public methods, the @internal tag should be used. Please use the @see tag for cross-referencing. All parameters and return

values should be documented.

As for comments within the code, comments should be used to describe intention or algorithmic overview where is it not immediately obvious from reading the code alone. There are no minimum comment

requirements and small routines probably need no commenting at all, but it is hoped that many larger routines will have commenting to aid future maintainers. Please write complete English sentences and capitalize the first word unless a lower-case identifier begins the sentence. Two spaces after each sentence helps to make emacs sentence commands work.

Short one-line comments and long comments can use the C++ comment style; that is, '//', but longer comments may use C-style comments:

```
/*
 * A longer comment,
 * with multiple lines.
 */
```

Variable declaration should have a short, one or two line comment describing the purpose of the variable, unless it is a local variable whose use is obvious from the context. The short comment should be on the same line as the variable declaration, unless it is too long, in which case it should be on the preceding lines.

Every function should be preceded by a detailed (Doxygen) comment block describing what the function does, what the formal parameters are, and what the return value is (if any). Either Doxygen style "\" or "@"

is permitted. Every class declaration should be preceded by a (Doxygen) comment block describing what the class is to be used for.

Casts

Where casts are necessary, use the Google C++ guidance: "Use C++-style casts like static_cast(double_value), or brace initialization for conversion of arithmetic types like int64 y = int64{1} << 42."

Try to avoid (and remove current instances of) casting of uint8_t type to larger integers in our logging output by overriding these types within the logging system itself. Also, the unary '+' operator can be used to print the numeric value of any variable, such as:

```
uint8_t flags = 5;
std::cout << "Flags numeric value: " << +flags << std::endl;
```

Avoid unnecessary casts if minor changes to variable declarations can solve the issue.

Miscellaneous items

- The following emacs mode line should be the first line in a file:

```
* -*- Mode: C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
```

- ```
  NS_LOG_COMPONENT_DEFINE("log-component-name");
  ```

  statements should be placed within namespace ns3 (for module code) and after the

  ```
  using namespace ns3;
  ```

  in examples. NS_OBJECT_ENSURE_REGISTERED() should also be placed within namespace ns3.

  - Const reference syntax:

    ```
    void MySub (const T&);    // Method 1  (prefer this syntax)
    void MySub (T const&);    // Method 2  (avoid this syntax)
    ```

  - Use a space between the function name and the parentheses, e.g.:

    ```
    void MySub(const T&);     // avoid this
    void MySub (const T&);    // use this instead
    ```

    This spacing rule applies both to function declarations and invocations.

    - Do not include inline implementations in header files; put all implementation in a .cc file (unless implementation in the header file brings demonstrable and significant performance improvement).
    - Do not use “nil” or “NULL” constants; use “0” (improves portability)
    - Consider whether you want the default copy constructor and assignment operator in your class, and if not, make them private such as follows:

      ```
      private:
      ClassName (const Classname&);
      ClassName& operator= (const ClassName&)
      ```

    - Avoid returning a reference to an internal or local member of an object:

      ```
      a_type& foo (void);  // should be avoided, return a pointer or an object.
      const a_type& foo (void); // same as above
      ```

      This guidance does not apply to the use of references to implement operators.

      - Expose class members through access functions, rather than direct access to a public object. The access functions are typically named “Get” and “Set”. For example, a member m_delayTime might have accessor functions GetDelayTime () and SetDelayTime ().
      - For standard headers, use the C++ style of inclusion, such as
      ```

```
#include <cheader>
```

instead of

```
#include <header.h>
```

- Do not bring the C++ standard library namespace into ns-3 source files by using the “using” directive; i.e. avoid “using namespace std;”.
- When including ns-3 headers in other ns-3 files, use <> when you expect the header to be found in build/ (or to be installed) and use “” when you know the header is in the same directory as the implementation.
  - inside .h files, always use

    ```
    #include <ns3/header.h>
    ```

  - inside .cc files, use

    ```
    #include "header.h"
    ```

    if file is in same directory, otherwise use

    ```
    #include <ns3/header.h>
    ```

- When writing library code, try to avoid the use of unused function parameters; use the NS_UNUSED () macro to explicitly note that a parameter is intentionally ignored. However, when writing client code (tests and examples), NS_UNUSED() is not required and may be avoided especially if the author feels that it clutters the code (example: hooking a trace source to gather program output, but not using all of the parameters that the trace source provides).

---

ns-3

Wiki (https://www.nsnam.org/wiki/)
Bugzilla (https://www.nsnam.org/bugzilla/)
webmaster@nsnam.org (mailto:webmaster@nsnam.org)

a discrete-event network simulator for internet systems

© 2011 — 2019 nsnam • Unless otherwise noted, license for website content: CC BY-SA 4.0
(https://creativecommons.org/licenses/by-sa/4.0/)

(/)