**Version History**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ***Ver. No.*** | ***Authors*** | ***Date*** | ***Reviewers*** | ***Review Date*** | ***Release Date*** |
| 1.0 | Application Development Team | 27-Aug-2018 | QMF | 31-Aug-2018 | 03-Sep-2018 |
| 2.0 | Application Development Team | 10-Dec-2019 | QMF | 13-Dec-2019 | 16-Dec-2019 |
| 3.0 | Application Development Team | 02-Nov-2020 | QMF | 06-Nov-2020 | 10-Nov-2020 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**Change History**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Ver. No.*** | ***Section*** | ***Date*** | ***Change Information*** | ***RFC No.*** |
| 1.0 | All | 03-Sep-2018 | New Release | - |
| 2.0 | All | 03-Sep-2018 | New Release | - |
| 3.0 | All | 02-Nov-2020 | Annual Review | - |
|  |  |  |  |  |
|  |  |  |  |  |

# STANDARDS FOR CODING IN C++

1. **Objective**

The objective of this document is to define coding standards for programs developed in ‘C++’.

1. **Scope**

This standard applies to programs developed in ‘C++ ’.

1. **References to (checklists, forms, guidelines, lists, standards, Templates, other processes)**

| ***Item*** | ***Description*** | ***ID*** |
| --- | --- | --- |
| **Checklists** | *--* | *--* |

1. **Document Organization**

This document is organized as follows:

* Class Declarations
* Format the Source Code Files
* Rules for Indentation
* Function Size
* Globals
* Naming Conventions
* Conventions for Comments in the Code
* Source Code Compatibility
* Error Messages & Exception Handling
* Debug Messages
* Additional Conventions

1. **Class Declarations**

* A class should be declared in a .h file and the definition should be in the .cpp file, both with the same name as that of the class. Therefore creating a class should result in a pair of <class name>.h and <class name>.cpp files
* A file containing a class declaration has a file name of the form <class name> + extension. Use uppercase and lowercase letters in the same way as in the source code
* Valid file extensions are ‘.h’ and ‘.cxx’
* Each class declaration has its own header file with the standard elements
* Each class consists of the following elements that have to be listed in the order given below:
* constructors
* destructor
* assignment operator
* new methods of the class
* overridden methods of the base classes; the order is given by visiting the base classes in breadth first order (if multiple inheritance is used)
* data member of the class
* Each group may have their own public, protected and private sections separated by special comments

*Example:*

class NewClass : public BaseClass   
 {  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
// Constructors, destructor, assignment  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
public:  
 // NewClass(); // Use default version.  
private:  
 NewClass(const NewClass&); // No copies.  
  
public:  
 // ~NewClass(); // Use default version.  
  
private:  
 NewClass& operator=(const NewClass&); // No assignments.  
  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
// New methods of class NewClass  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
public:  
  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
// From class BaseClass  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
public:  
  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
// My data members  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
private:  
  
};

* Class declaration follows the canonical class form. The canonical class form (see Coplien, Advanced C++) is defined as the minimal set of management methods in a class, which need to be provided by the programmer such that a class works properly. The canonical class form has to be applied, whenever a class has pointer data members and allocates its own memory!
* It is good style, however, to always follow this form, indicating that the developer is very well aware of the pitfalls of C++ programming
* Each file (.h/.cpp) should contain the declaration/definition of only one class (primary class). May also contain declaration/definition of any other class/classes used only by the primary class and not by any other class. If the primary class has another class as its member, then the other class definition should precede the definition of the primary class in the file
* All the class member variables should be declared before the member functions
* A member variable and a member function cannot share the same name
* Provide or exclude the default and copy constructor, the destructor and the assignment operator. Use commented versions to indicate intenional use of the default versions
* The C++ compiler provides default implementations for default and copy constructor, destructor and assignment operator. This may result in severe errors hard to locate if the methods are forgotten. A related problem is that for some classes (like those used in the singleton pattern) no copies or assignments should be allowed. Not providing the copy constructor and the assignment operator does not mean they are not there! Another often-used approach is to provide the methods, but print out a warning (or worse: using assert(0)). The drawback of this variant is that it may only accidentally be discovered during run time. Following this rule assures that there are errors at compile time or at link time!
* If the programmer really wants the compiler-generated versions of these methods, he has to document this decision and use the standard comment
* / Use default version to show that the compiler generated versions is intentionally accepted.

*Example:*

class NewClass  
{  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
// Constructors, destructor, assignment  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
public:  
 **// NewClass(); // Use default version.  
 // NewClass(const NewClass&); // Use default version.**public:  
 **// ~NewClass(); // Use default version.**public:  
 **// NewClass& operator=(const NewClass&); // Use default version.**...  
};

* The default copy constructor and assignment operator can often not be accepted and should therefore be provided by the programmer. But if we do not want these methods, we forbid them **by declaring them private and without providing an implementation**, marking it with the special comments:

// No copies  
// No assignments

*Example:*

class NewClass  
{  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
// Constructors, destructor, assignment  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
public:  
 NewClass();  
**private:  
 NewClass(const NewClass&); // No copies.**public:  
 // ~NewClass(); // Use default version.  
  
**private:  
 NewClass& operator=(const NewClass&); // No assignments.**...  
};

* Since there has a copy constructor been defined, the default constructor has to be defined and implemented. The compiler or the linker will complain if the methods are used in the application and give an error message at compile/link time instead of at runtime
* Member variables cannot be initialized in the declaration but needs to be initialized in the constructor because whenever an instance of an object comes into existence, the constructor is executed first
* Group the methods of a class according to the different base classes. Using inheritance in C++, it is often hard to remember where a method was originally defined (and therefore documented) when a class hierarchy gets deeper. To make it simpler for reading a class declaration, all overridden methods should go into a specially commented section, one for each base class from bottom to the top of the inheritance tree

*Example:*

class Base   
{  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
// Constructors, destructor, assignment  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
public:  
  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
// New methods of class Base  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
public:  
  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
// My data members  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
private:  
  
};  
  
class Derived A : public Base   
{  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
// Constructors, destructor, assignment  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
public:  
  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
// New methods of class Derived A  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
public:  
  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
// From class Base  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
public:  
  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
// My data members  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
private:  
  
};  
  
class DerivedB : public Derived A   
{  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
// Constructors, destructor, assignment  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
public:  
  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
// New methods of class DerivedB  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
public:  
  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
// From class Derived A  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
public:  
  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
// From class Base  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
public:  
  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
// My data members  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
private:  
  
};

* As far as possible all member variables should be declared as private and member functions should be written to access them. This is to ensure data encapsulation. May not always be followed but is a good programming practice.
* For each group of methods list first public then protected and finally private elements.
* Declare a virtual destructor in every base class that has a virtual function. Using virtual destructors, objects can be destroyed without knowing their type.
* If a class has pointer member, then it is essential that class definition should have the constructor for allocating, destructor for deallocating memory. The class should also have an assignment operator and copy constructor in such case.
* Don’t write inlines inside the class declarations (implicit inline). Use explicit inline declarations instead.
* All the inline method definitions are listed below the class declaration, separated by the special comment:

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ in lines

* The declaration itself will be hard to read if there are inline methods implemented inside. The class declaration should always be as clean as possible.
* The following code segment shows how inline methods should be formatted:

class Demo  
{  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
// Constructors, destructor, assignment  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
 ...  
  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
// New methods of class Demo  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
public:  
 int getValue();  
 ...  
  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
// My data members  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
private:  
 int myValue;  
}  
  
//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_-\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ in lines  
  
inline int Demo::getValue()  
{  
 return myValue;  
}

* It is not necessary to put the inline classifier in front of the method declaration. This is only needed for the following implementation:
* Any global variable used by a particular class only should be defined as static variable of that class. By doing this all instances of this class share only one copy of this variable
* Repeat the virtual keyword. If a method of a base class that was declared virtual is overridden in a derived class, the virtual keyword does not have to be repeated. However, always repeat the virtual declaration

*Example:*

class Conduit  
{  
 ...  
protected:  
 virtual void invoke(Visitor\* aVisitor);  
};  
  
class Mux : public Conduit  
{  
 ...  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
// From class Conduit  
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
protected:  
 **virtual** void invoke(Visitor\* aVisitor);

}

1. To prevent repeated declaration of the same class, each header file should contain #ifndef \_HEADER\_H/#define \_HEADER\_H /#endif macro. All the required #includes for a class declaration should be in the header files
2. Don’t use macros. C++ offers language elements, which (almost) eliminate the need for preprocessor commands. In C, macros are often used for (1) constants and (2) untyped functions. For both use cases, C++ offers a much better (type-safe) solution. Constants are now a built in Language feature of C++ (and of ANSI C). Untyped functions are covered by C++ template functions. There are, however, reasons to use macros, which still make sense in C++: Definition of Symbols for conditional compilation and macros, which generate Code
3. Use parenthesis around the parameters of macros

*Example:*

#define TRACE\_ANY (point, message)  
 TRACE (tracehandler::TR\_ANY, cClassName, cMethodName, **(**point**)**, **(**message**)**)

1. **Format the Source Code Files**

* The format for files containing source code and for headers will be as follows:
* Header
* #include Statements
* Literal Definitions
* MACRO Definitions.
* Class Definitions
* It is recommended that conditional compilation statement be put around the #include statements in the source code files

## Format of the Source Code File Header

1. A source code file should contain the following information at the start of the file as Program Header. The information appears as a comment with following:

* Copyright Notice
* The name of the file
* The initials of the programmer who wrote the file and the group to which he/she belongs
* Date of creation
* The development platform
* The target platform(s)
* Names of the functions in the file that are entry points
* List of the macros that are used in this file and where they are defined
* Dates of updations with a brief description of the changes made and the initials of the person making the changes. Entries in this section should be numbered (see Commenting Changes to the Code) and ordered chronologically
* List of functions used from a third party library and their library names

## Format of the Copyright Notice

1. The copyright notice should contain the following text:

Copyright (c) 1994 <Company Name>

All Rights Reserved.

THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF

<Company Name>

The copyright notice above does not evidence any actual or

intended publication of such source code

## Additional Header File Conventions

1. The header file should define a pre-processor symbol when compiled. The header file code should be compiled only if that symbol is not defined previously

#ifndef \_MYHEADER\_H

#define \_MYHEADER\_H

*the contents of myheader.h*

#endif

*The header file should not contain any executable C/C++ statements*

1. **Rules for Indentation**

* Each block of code should be indented with respect to the outer block. The indentation should be 4 spaces. Do not use tabs for indentation. Large program blocks should end with endblock comment. The switch statement should be indented the following way:

switch(<expression>)

{

case <value1> :

<statement>

.

case <value2> :

<statement>

} /\* endswitch \*/

* Try to avoid using one switch statement within another
* The opening brace is set always below the corresponding keyword or function/method name and the closing brace has the same indent as the opening one

1. **Function Size**

* The average function size should be sixty source code lines. Under no circumstances should the function size be more than hundred source code lines
* Additionally following things should be kept in mind to decide the optimum function size
* The level of indentation should not exceed six levels
* A single block of code should not exceed twenty to thirty lines

1. **Globals**

* All global variable definitions must be centralized in to a single compiled source file, or atleast a set of source files that contain only global variables and a function that initializes them
* Comments should accompany declaration of every global variable stating the usage of the variable. Try to use as few global variables as possible. If a global is used by functions inside a single file, restrict its use by making it static

1. **Naming Conventions**

* The name of the variable should reflect the data stored/referred by the variable. Hungarian Notation should be used

## Format Variables and User Defined Data Types

1. The format of a C variable or a C function should be:

* type format name
* The type component gives the data-type of the value represented by the variable or the value returned by the function. The format component should be used if the value represented or returned is an array or a pointer. The name component is the user-defined name of the variable. It is recommended to be as meaningful as possible.

|  |  |
| --- | --- |
| ***Data Type*** | ***Notation*** |
| char | c |
| float | f |
| double | d |
| int | i |
| Long int | l |
| short int | s |
| void | v |
| structure | t |
| class | o |
| widget | w |
| swidget | sw |
| x-type | x |

|  |  |
| --- | --- |
| ***Format*** | ***Notation*** |
| pointer | p |
| array | a |

* Avoid using underscores in the variable name. Instead make first letter of each word in the name capital
* If a variable is unsigned add the prefix u to it
* The names of the global variables should start with g
* Use the variables i, j ... n as loop counters
* Do not use similar looking names in the same scope
* Do not use similar sounding names in the same scope
* The length of the name of the variable should not exceed 30
* Structure names should have the prefix tag
* Types begin with a capital letter
* Enums have the prefix E and enum tags start with the prefix e

*Example:*

enum EDispatchMode  
{  
 eBroadcast,  
 eBypass,  
 eDispatch,  
 eDummyMode  
};

1. General rule for all Names: In names, built out of more than one word, the words are delimited by a capital first letter of the following word
2. Template parameters have the prefix T

*Example:*

template <class **TKey**, class **TValue**>  
class Map   
{  
 ..  
};

1. Constants have the prefix c.

*Example:*

const Octet **cSetupReq** = 0x81;

1. Defines consist of capital letters and underscores only

*Example:*

#define COMLIB\_VXWORKS

1. Method and function names start with a lowercase letter
2. Exceptions to this rule are, of course, constructors and destructors, which have to have the same name as the class and therefore start with a capital letter

*Example:*

virtual void disconnectPeerA();

1. Data members start with a lower case letter
2. Data members have the prefix my or our. In order to distinguish data members from other variables and method or function arguments the prefixes my (regular data members) or our (static data members) are used
3. Do not use these prefixes anywhere else!

*Examples:*

class Conduit  
{  
 ...  
private:  
 Conduit\* **myPeerA**; // I know my PeerA Conduit   
 Conduit\* **myPeerB**; // I know my PeerB Conduit  
};  
  
class StateU0 : public SigCtrlState  
{  
public:  
 static BaseState\* getInstance();  
private:  
 static BaseState\* **ourInstance**; // Singleton  
};

## Macros

1. All macro names should be in capital letters. Words inside the name should be separated by an underscore
2. **Conventions for Comments in the Code**

* All comments should be in plain English and written as you write code, or even before you write code. Make sure that the comment and the code agree
* The comment should be indented at the same level at which the corresponding code is indented. Each comment will start with a code indicating the type of the comment

|  |  |
| --- | --- |
| **Comment Type** | ***Code*** |
| Module Header | MH |
| Globals Description | GL |
| Function Header | FH |
| Code Comment | CC |
| Change Comment | NC |

## Commenting a Function

1. Each function will have the following information associated with it:

* Prototype
* Purpose
* Input arguments and their semantics
* Modifiable arguments (try to use as few of them as possible) and their semantics
* Return value and its semantics
* Algorithm (optional) is provided if the function is
* Big (say more than 10 lines)
* The logic is complicated
* The logic is not apparent (by just looking at the purpose and the code)
* Side effects (optional) Side effects should also list the global variables that are modified by this function
* The names of the functions that are called from this function
* The names of the functions that call this function (the list may not be exhaustive)

1. Items 2 to 9 are collectively called as function header

**Example:**

/\*

prototype

\*/

int iProcessData(int iCommand, char \*cpData, int \*ipDatalen);

/\*

Purpose : This function performs compresses/uncompresses the data supplied.

Inputs : Command denotes the operation to be performed on the data. Legal values are

CRUNCH\_IT

EXPAND\_IT

Modifiable Values Supplied :

cpData pointer to the buffer where the data is stored. will contain the processed data on return.

ipDatalen points to a location where length of the data is stored. will contain the length of the processed data on return.

Return Value:

The function will return an integer indicating the result of the operation.

It returns:

SUCCESS the operation succeeded

INVALID\_COMMAND the command parameter is not correct

FAILED could not carry out the operation

Algorithm:

SWITCH command

CASE CRUNCH\_IT

compress the spaces in the data

FOR every word in the line

IF the word is not in the hash table

add the word to the hash table

IF unsuccessful

RETURN FAILED

ENDIF

ENDIF

replace the word by its table entry number

ENDFOR

update the size of the compressed text

RETURN SUCCESS

CASE EXPAND\_IT

WHILE there is more data to uncompress

get the next hash entry number from the data

IF entry does exists in the hash table

RETURN FAILED

ENDIF

replace the entry number in the data with the word

ENDWHILE

RETURN SUCCESS

DEFAULT

RETURN INVALID\_COMMAND

ENDSWITCH

Side effects :

If the function fails to perform the specified operation the data buffer and the data lengths remain in inconsistent state. The hash table gets updated with every new word found in the data.

This routine calls the following routines :

strtok available in the library

gather\_spaces from the file utils.c

find\_word

add\_word from the file hash.c

This function is called by:

compress\_file to compress a file line by line in file

compress.c

expand\_file to expand a compressed file line by

line in file expand.c

\*/

## Commenting Iterative Statements

1. Each iterative statement may contain a comment header at the start of the looping statement. This comment header is optional and should be included for complex loops (large loop body, complex or non-apparent looping condition)
2. The comment header should contain:

* The precondition
* What each iteration does
* The post condition
* If it is not clear from the code how the post condition is reached from the precondition, it should be explained

## Commenting the IF Statement

1. Each conditional statement may also be commented. The commenting is necessary for conditional statements with complex or non-apparent conditions

* The comment should include:
* The condition
* What happens if it is true
* What happens if it is false

## Commenting Changes to the Code

1. All changes made to the code should be properly commented using the change comment. In the change comment, a number should follow NC. This number should be the number under which this change is recorded in the module header
2. If code is to be added, then a comment should precede the added code stating the date of the change and brief explanation
3. If code is to be removed, then it should not be physically deleted from the file. Instead it should be commented out with date and explanation
4. If code is to be changed, then the previous code should first be duplicated and then one copy is commented out with date stamp and the explanation and the other copy should be modified
5. **Source Code Compatibility**

* The source code is expected to be portable across operating systems and their flavors, compilers and machine architectures. The following points need to note in this regard.
* Path names of non-standard header files
* Size of an integer variable
* Use a safe subset of C++ language features
* Default character type (signed or unsigned)
* Non-standard library functions
* The #ifdef pre-processor statement should be used to include/exclude non-portable part of the source code.
* Enabling the compiler switch for ANSI Compatibility helps make code more portable

1. **Error Message and Exception Handling**

* All the error messages should be written to stderr. If a system call fails, the function perror (provided by the standard C library) should be called. The name of the function that failed should be passed to perror. Functions throwing exceptions has to have the exception handling code.
* Following information should accompany messages displayed when software aborts.
* Name of the source code file (given by the macro \_\_FILE\_\_)
* The line number inside the source code file where it is aborting (given by the macro \_\_LINE\_\_)
* The name of the function in which it is aborting

1. **Debug Messages**

* All debug messages should be written to stdout. The debugging statements should be conditionally compiled if DEBUG is defined. The debugging statements should not be physically removed from the code

1. **Additional Conventions**

* **General Rule**: Every time a rule or convention is broken, this must be clearly documented
* Every statement in the code should start on a new line
* All compiler errors and warnings should be enabled. Always compile with highest warning level such that the compiler reports all warnings. Change your code to cause no warnings!
* The final version of the source code should not generate any errors or warnings when compiled
* No errors should be reported by lint on the source code. Try to remove warnings sounded by lint to the best extent possible
* Delete unnecessary code instead of commenting it out. Do not misuse #ifdef preprocessor commands to comment out code
* Always use the normal form for assignment operator implementation. Test for self assignment
* All keywords should be preceded and followed by a space
* All operators except structure member dot and arrow operators, unary minus operator, the increment and decrement operators, array member index operator, & and \* operators for address of and contents pointed by, the round brackets should be preceded and followed by a space
* An explicit typecast should be followed by a space
* Commenting for every typecast is must.
* Include comments after #include statements.
* Use <...> for system includes or foreign libraries, “...“ for own includes
* Use as few include statements in a header file as possible
* System header files should be included first. User created header files will follow them. There should be a blank line between these two blocks
* Order of include files:
* header file of the class
* header files of the same module / library / project
* header files of the other module / library / project of the component
* header files of the other layers in the descending order (your platform, Component Libraries)
* system header files (<window.h>, <string>, …)
* Space should be inserted after a comma, in parameter list for a function call for example.
* Routines that make a new object, which the caller must delete, begin with create.

*Example:*

virtual Boolean ConduitFactory::createConduitStackToA(Conduit\*& aTopConduit,  
 Conduit\*& aBottomConduit,  
 Visitor\* aVisitor) = 0;

* Routines that copy an existing object, where the caller must delete the copy, begin with copy
* Routines that abandon an object and pass deletion responsibility to the caller begin with orphan
* Getters with boolean return value begin with is

*Example:*

template <class TKey, class TValue>  
class Map   
{  
 ..  
 bool **isEmpty**() const;  
};

* Getters begin with get if none of the Rule 48 to Rule 51 is applicable
* When the caller takes no responsibility for deletion (therefore depends on the internal state of the called object) or the return is of a value type, the prefix get is used

*Example:*

class Protocol : public Conduit  
{  
 ..  
 const BaseState\* **getNextState**() const;  
};

* Setters that accept an object (the caller has allocated) and take responsibility for deleting it begins with adopt

*Example:*

class String  
{  
public:  
 ...  
 **adoptRep**(const char\* newRep);  
};

* Adopting methods that cannot follow the previous rule (such as constructors) prefix the name of the argument with adopt

*Example:*

class MsgTransporter : public Visitor  
{  
public:  
 MsgTransporter(unsigned long dispatchKey,  
 DispatchMode dispatchMode,  
 Messenger\* **adoptMessenger**);  
};

* All names (e.g. name of the author) should be initials of the person in the upper case.
* Compound statements are always enclosed with braces ({ })
* Flow control statements consist of blocks that are executed based on a condition. If the block has just one statement, braces enclosing the block are optional (by the syntax of C++). This can be error prone if an additional statement is added. Therefore the consequent use of braces eliminates a potential source of errors

*Example:*

for (int i=0; i<100; i++) for (int i=0; i<100; i++)  
{ if (i % 2)   
 if (i % 2) cerr << i << “ even” << endl;  
 {  
 cerr << i << “ even” << endl;  
 }  
}

* If statement :

if (condition)  
{  
 statements on true.  
}  
else  
{  
 statements on false.  
}

* for statement:

for (int i=0; i<100; i++)  
{  
 loop statements.  
}

* while statement:

while (condition)  
{  
 statements.  
}

* do..while statement:

do  
{  
 statements.  
}   
while (condition);

* switch statement:

switch (expression)  
{  
 case tag1:  
 statements;  
 break;  
 case tag2:  
 ...  
  
 default:  
 break;  
}

* Class declaration:

class ClassName  
{  
 //...  
};

* Functions/methods:

returnValue foo(arguments)  
{  
 statements;  
}

* Templates:

template <class TObject>  
returnValue ClassName<TObject>::methodName(arguments)  
{  
 statements;  
}

* The dereference operator ‘\*’ and the address-of operator ‘&’ should be directly connected with the type names in declarations and definitions.
* The characters ‘\*’ and ‘&’ should be written together with the type of the variable, not with its name, in order to emphasize that they are part of the type definition. Instead of saying that \*i is an int, say that i is an int\*

*Example:*

**char\*** aString;  
**int&** aIntRef = ..;  
  
bool isValidName(const **String&** name);

* Magic numbers should be avoided as far as possible. They should always be defined as macro and then used
* Enumerated variables should be used when integer variables can take only certain predefined values only
* Try to use as few exit points from a function as possible
* Try to keep the code as simple as possible
* Use library functions to perform a task, if such a function is available
* Try to minimize use of temporary variables
* Use parenthesis to avoid precedence ambiguity
* Unlike C, in C++ no testing is necessary after memory allocation using new.
* Do not mix new/delete and malloc/free
* Do not mix signed and unsigned types
* Do not mix int and long
* Organize the functions inside a file hierarchically. The rule is, if you are calling another function, the called function should appear later in the file
* Make sure that you always initialize a variable before using it
* Watch out for rounding off errors, especially if you are doing floating point calculations
* Do not try to optimize at the code level. If you have any suggestions regarding algorithmic efficiency, talk to the project manager so that it can be included in the design after due deliberations
* Do **not** use nested comments
* Do **not** use goto statement
* Do **not** use the ternary operator
* Do **not** initialize variables in the header files
* Each line in the source code should be less than or equal to 80 characters
* Avoid using **register** variables
* Avoid using pointers to unknown types (void \* variety)
* Some linkers are not case sensitive
* Always trap defaults in the switch statement. Place error sounding/aborting code there if the control is not expected to come there
* Always provide a break statement after each case or default clause
* Avoid using fall through in the switch statement
* Avoid complicated conditional statements. Use explicit comparisons in the conditional statements.
* Always parenthesize parameters to the macro in the expansion