NATIONAL WEATHER SERVICE
OFFICE of HYDROLOGIC DEVELOPMENT

Science Infusion Software Engineering Process Group (SISEPG)

C++ Programming Standards and Guidelines

Version 1.11

1. Introduction

The Office of Hydrologic Development (OHD) develops and maintains software which the National Weather Service (NWS) Weather Forecast Offices (WFOs) and River Forecast Centers (RFCs) use to generate hydrologic forecasts and warnings for rivers and streams across the country. OHD also develops and maintains software which runs centrally to acquire and distribute critical data to the WFOs and the RFCs. Software development and maintenance has become a critical component supporting the operations of NWS forecast offices and it is essential that it be well written and maintained.

Well written software offers many advantages. It will contain fewer bugs and will run more efficiently than poorly written programs. Since software has a life cycle and much of which revolves around maintenance, it will be easier for the original developer(s) and future keepers of the code to maintain and modify the software as needed. This will lead to increased productivity of the developer(s).

The OHD Science Infusion Software Engineering Process Group (SISEPG) has developed standards and guidelines to ensure that developers follow good and widely accepted software development practices when coding. It is believed this will lead to well written and better structured programs, which must be simple, intuitive, and uniform. The overall cost of the software is greatly reduced when the code is developed and maintained according to software standards.

This document will present standards and guidelines for the C++ Programming Language. The C++ standards are programming techniques which OHD developers are expected to follow to help to them write high quality software. The C++ standards will be enforced through peer reviews and code walkthroughs. The C++ guidelines are good programming practices which developers are encouraged to adopt.

The developer should also read the OHD General Software Development Standards and Guidelines document to become familiar with the standards and guidelines deemed by the SISEPG to be applicable to all programming languages.

It is important to note that standards are not fixed, but will evolve over time. Developers are encouraged to provide feedback to the SISEPG (sisepg@gateway2.nws.noaa.gov). Also each project area may derive its own standards and guidelines if special requirements are desired. Finally, the developers are to follow the "OHD General Software Development Standards and Guidelines" except where specified in this document.

2. Standards

2.1 File Names

- Header files and namespace files must use suffixes: `.h, .H, .hh, .hpp, or .hxx`
- Source files must use suffixes: `.C, .cc, .cpp, or .cxx`
- File name suffixes shall be consistent within a project
- If the file is a `class` or a `struct` declaration and definition, or namespace, the file name shall be `UpperMixedCase` plus one of the suffixes listed above, for example, the header file and source file for class `MyClass` shall be `MyClass.h` and `MyClass.cpp` respectively. Otherwise, the `lowerMixedCase` should be used for file name plus a suffix. For example, the header file and source file for function `myFunction` shall be `myFunction.h` and `myFunction.cpp` respectively.

2.2 File Organization

- Each file contains only one class declaration or definition except functors and static classes (see Section 2.7 for details on functors and static classes).
- Must contain the *comment block* specified in the *"OHD General Software Development Standards and Guidelines"* at the beginning.
- Include a brief description about the content of the file after the *comment block*.
- The content of the file shall be in the following order:

1. A header file should include the following preprocessor directives at the beginning and the end of the file to prevent multiple inclusions. The convention is an all `UPPERCASE` construction of the file name and the `H` suffix connected by an underscore `'_'`. This applies to namespace files too. For example,

```
#ifndef XX_H
#define XX_H
  ...
#endif//#ifndef XX_H
```

2. *Comment block* described in the *"OHD General Software Development Standards and Guidelines"*
3. A brief description about the content of the file
4. Include header files
5. `#defines` and `Macros`
6. The `"use"` directive should not be used in header files. To do so will likely cause name confliction and include unnecessary files to retrieve the directive

in the header file(s) and because the header file may be included in another header file or other header files.

7. Class or function declaration or definition.

2.3 Include Files

- Always use the C++ standard library headers that have no extension, the `<xxx.h>` versions are deprecated. For example:

```
#include <iostream>
#include <string>
```

- Use the new prefix `c` instead of the old extension `.h` for C standard header files. For example:

```
#include <cstdlib> //was:<stdlib.h>
#include <cstring> //was:<string.h>
```

- Use the `< >` pair for library and system headers, for example:

```
#include <filename.h>
```

- Use the " " pair for non-system (user defined) headers, for example:

```
#include "filename.h"
```

- Do not use absolute or relative paths to point to your header files. It is mandatory that you use the –I<dir> directive of the C compiler to instruct the compiler where your header files are located..
- List the system header files first in alphabetical order. Then list the non system include files (including COTS includes) also in alphabetical order.

Example

```
#include <iostream>
#include <string>
#include <vector>

#include “MyClass.h”
#include “MyFunc.h”
```

2.4 Comments

- Use the JavaDoc convention format for the documentation comment and the *comment blocks*[1], such that the comment can be extracted by a documentation tool, such as `'ccdoc'`. For example,

```
/**
  * Documentation goes here
  */
  class MyClass
  {
     ...
  }
```

- Use the C++ comment "//" style or the C style (/* ... */) for inline comments

2.5 Naming Schemes

In general, names shall be mnemoic and meaningful. All variables should be initialized before use.

- `namespace, class, struct,` template argument and parameter names (variables that are visible in more than one linkage unit)
    - Use uppercase letters as word separators, `Lowercase` for the rest of a word
    - First character in a name is uppercase
    - No underscores('_')

  For example, `class MyClass{ ... }`
- Macro and `#defined` constant, enum, union, class static data member, and global variable names
    - Use all uppercase letters with underscore '_' as separators

  For example, `#define MY_DEFINE  1`
- Class method and variable names
    - Use uppercase letters as word separators, lowercase for the rest of a word
    - First character in a name is lowercase
    - No underscores('_')

  For example, `MyClass::myMethod(){ ... }`
- Class data member names
    - Private data member names shall be prepended with the underscore '_'
    - Following the '_', use the same rules as for method names (where the first character is an underscore and the next characters are mixed lowercase), for example, `int _myDataMember`
    - `static const` data member shall be all uppercase, for example, `public static const PI=3.14159`
- `typedef` reflect the style appropriate to the underlying type. For example, use method name scheme for `typedef`-name for a method or a built-in type and class name scheme for `typedef`-name for a `class, struct`, or `union`. If the purpose of the `typedef` is to transparently switch between underlying types, choose any appropriate naming style.
- Do not use class, struct, variable, method names that differ by case only.
- Do not use class, struct, variable, method names that conflict with those in the standard library.
- C function names

- Avoid creating C functions when programming in C++;
- See the *OHD C Programming Standards and Guidelines* for C function name schemes if you have to create C functions.

2.6 Readability and Maintainability

See the *OHD Software Development Standards and Guidelines* for guidance on how to make your files more readable and maintainable. These are standards which apply to programming languages in general not just to the C++ language.

2.6.1 Indentation

The *OHD Software Development Standards and Guidelines* document states that consistent indentation shall be used when distinguishing conditional or control blocks of code.

Use 3 or 4 spaces to indent code and be consistent with whatever you choose. **Do not use tabs.**

2.6.2 Braces

- Place braces under and inline with keywords, like this:

```
if (condition)              while (condition)
{                           {
   ...                         ...
}                           }
```

Or use the old c style like this

```
if (condition){             while (condition){
   ...                         ...
}                           }
```

- "`for loop`", "`if then else`" and "`try catch`" statements, always uses braces form, even if there is only a single statement within the braces, for example, use

```
for (initialization; condition; update )
{
   statement;
}
```

Instead of

```
for ( initialization; condition; update )
   statement;
```

Or use the following style (Kernighan and Ritchie) as an alternative.

```
for (initialization; condition; update ) {
   statement;
```

```
}
```

- If a loop statement is empty, use the following form

```
for (initialization; condition; update)
;
```

or

```
 for (initialization; condition; update)
{}
```

Do not do this

```
 for (initialization; condition; update);
```

It is hard to read or add more codes to the loop.

2.6.3 "return" statements

Do not use parenthesis in return statements when it's not necessary.

2.6.4 "if then else" statments

- Use the following style for `if then else` statement

```
if (condition)
{
}
else if (condition)
{
}
else
{
}
```

Or use the following style (Kernighan and Ritchie) as an alternative.

```
if (condition){
}
else if (condition){
}
else{
}
```

2.6.5 "switch" statments

- `switch` formatting
  - Falling through a case statement into the next case statement shall be permitted as long as a comment is included.

- The default case shall always be present and trigger an error if it shall not be reached, yet is reached.
- If you need to create variables put all the code in a block.

  For example

```
switch ( ... )
{
   case 1:
      ...
   // FALL THROUGH
   case 2:
   {
      int v;
      ...
   }
   break;
   default:
}
```

2.6.6 Split lines

The incompleteness of split lines must be made obvious (see the *"OHD General Software Development Standards and Guidelines"* for details)

2.7 Class Design

Class members are declared in the following order:

1. public members
2. protected members
3. private members

Classes must have a default constructor, a virtual destructor, a copy constructor, and an overloaded assignment operator, except functors, and static classes. Functors (function objects) normally have only one overloaded `( )` operator as member. Static classes contain only static member functions and are therefore never instantiated. Instead, they must be declared as a private default constructor (without any implementation) to prevent instantiation.

2.7.1 Default Constructor

Always provide a default constructor for your class. If the default constructor is sufficient, a comment should be added indicating that the compiler-generated version will be used. If your constructor has one or more optional arguments, a comment should be added indicating that it will function as a default constructor.

2.7.2 Virtual Destructor:

If your class is intended to be derived from by other classes then make the destructor virtual. You shall always make a destructor virtual for the sake of future extensibility. Only make it non-virtual if you have a real good reason to do so.

2.7.3 Copy Constructor:

If your class is copyable, either define a copy constructor and assignment operator or add a comment indicating that the compiler-generated versions will be used. If your class objects should not be copied make the copy constructor and assignment operator private and do not define bodies for them. If you do not know whether the class objects should be copyable, then assume not until the copy operations are needed. The copy constructor shall have `"const"` in the parameter.

2.7.4 Assignment Operator:

If your class is assignable, either define an assignment operator or add a comment indicating that the compiler-generated versions will be used. If your object shall not be assigned, make the assignment operator private and do not define a body for it. If you don't know whether the class objects shall be assignable, then assume not. The overloaded assignment `operator=` shall return `*this` as a `XXX&` (allows chaining of assignments). The assignment operator shall have `"const"` in the parameter.

2.7.5 Data Members:

Data Members shall be private; doing this will promote information hiding and restricted access, which are very important in objected oriented programming.

2.7.6 Example:

See Appendix A for a class declaration layout

2.8 Safety and Performance

2.8.1 Type Conversions

Type conversions must always be done explicitly. Never rely on implicit type conversion. Use the C++ set of casting operators: `static_cast`, `reinterpret_cast`, `const_cast` and `dynamic_cast` instead of C-style casting.

2.8.2 Use of Pointers

A pointer which is declared but not initialized is known as a dangling pointer. Un-initialized pointers are dangerous because subsequent operations on the pointer will not know if the pointer references a valid memory address. When a pointer variable is declared always initialize it either with the address of an existing object or the value NULL. This usually sets the pointer's value to zero. The operating system is very strict about memory operations in the reserved zero page address space. So initializing pointers to NULL will quickly find code problems where they are used before being set to point to valid memory addresses. Any place the pointer is subsequently used the C++ code shall first check the pointer to see if it is NULL before trying to dereference it.

Always test pointers for NULL values before trying to dereference them.

2.8.3 Do Not Hardcode Values

Do not hardcode numerical constants. Instead, declare a constant using either a `const` directive or a `enum` directive. In both cases, the constant name should be all uppercase.

2.8.4 Create Large Objects on the Heap

Use dynamically allocated memory for large arrays, as opposed to declaring local variables. Allocating objects dynamically will create them on the heap. The heap has more space than the stack.

2.8.5 Function Prototype Variables

The arguments specified in a function prototype shall be associated with variable names. These variable names must match the variable names in the function definition. Doing this makes function prototypes more meaningful. A programmer can tell more about the arguments which need to be supplied to a function if the arguments in the prototype have meaningful names associated with them.

2.9 C++ Standard Template Library (STL)

Try to use the C++ STL. The C++ STL contains a set of containers, iterators, function objects, algorithms and utility classes. They are better than the C++ native arrays and pointers. By using the STL, you don't need to reinvent the wheel. Also, codes created by STL are more readable and maintainable than classic C++ codes.

2.10 Global Variables

Use of global variables shall be minimized. Global variable values are very hard to trace while debugging a program.

3. Guidelines

General coding guidelines provide the programmer with a set of best practices which can be used to make programs easier to read and maintain. Unlike the coding standards, the use of these guidelines is not mandatory. However, the programmer is encouraged to review them and attempt to incorporate them into his/her programming style where appropriate.

- Use `static const` members instead of `#defined` constants
- Use `enum` to define a collection of integral constants
- Use inline functions instead of Macros
- Do not use templates directly. Instead use proper `typedefs`
- Practice `const` correctness. All "variables' and parameters that should not be changed have to be declared `const`. Furthermore all methods that do not alter the object have to be `const`. Especially `getXxx` methods should be `const` in all cases.
- Initialize all variables
- Avoid pointer arithmetic. Use the STL containers and iterators instead. The `operators ==` and `!=` are defined for all pointers of the same type, while the use of the `operators <, >, <=, >=` are portable only if they are used between pointers which point into the same array.
- Parts-of relation inheritance should be avoided. Recognize inheritance for implementation; use a private base class or (preferable) a member object [as an alternative]. [4]
- Local variables should be declared near their first use.
- Do not use assignment `operator=` when constructing an object, use the copy constructor instead
- For increment and decrement, the prefix form (++i or --i) is preferred, because it is more efficient (especially for the STL iterators).
- Do not put parenthesis next to keywords or function names. Put a space between the parenthesis and the keyword. For example,

  ```
  while (condition)
  ```

  Do not use the following form (without space)

  ```
  while(condition)
  ```

  Do put parenthesis next to function names. For example,

  ```
  myFunction ( arg1, ... )
  ```

Do not use the following form (without space)

```
myFunction(arg1, ...)
```

- To increase readability, put spaces between variables, key words and operators, for example,

```
average = ( x + y + z ) / total
```

do not do this

```
average=(x+y+z)/total
```

- Portability means that a source code file can be compiled and executed on different machines with the only changes being the possible inclusion of different header files and the use of different compiler flags. The header files contain `#define` and `typedef` constructs that may vary from machine to machine (a different machine may mean different hardware, a different operating system, a different compiler, or any combination of these).
- The C++ programmer needs to have a good grasp of the namespace of a class definition, and lifetime and scope of a variable
- Pointers should be named in some fashion that distinguishes them from other "ordinary" variables. One possibility is that pointer variable names start with a lowercase "p" followed by the rest of the name with the first character capitalized, e.g. pValue. This makes recognition of pointer variables in source code easier.

  Another possible naming convention for pointers is to append "ptr" to the end of pointer variable names. The name of the pointer should either imply the type of the data object it is referencing or the name of the pointer should describe the data it represents.

  The programmer should be consistent in naming pointers. Do not mix different naming conventions.
- Loop index variable names may be short and do not need to be descriptive.
- When using macros, it is essential to use parentheses to ensure correct evaluation of the macro.
  Example:

```
/* The following macro definition could result in an error. */
#define PI 3.14159
#define CIRCLE_AREA(x)   ( PI * x * x )

area = CIRCLE_AREA ( c + 2 ) ;

/* The preprocessor will expand this macro to: */
```

```
area = PI * c + 2 * c + 2 ;

/* Given operator precedence in C, this will be incorrectly
   evaluated as: */

area = ( PI * c) + ( 2 * c ) + 2 ;

/* The following macro definition uses parentheses to ensure that
   it is evaluated as the programmer intended it to be. */

#define CIRCLE_AREA(x) ( PI * ( x ) * ( x ) )
area = CIRCLE_AREA ( c + 2 ) ;

/* The preprocessor will expand this macro to: */

area = PI * ( c + 2 ) * ( c + 2 ) ;
```

This will evaluate in the correct order.

- Limit use of library functions, especially in loops. For example, consider using the form `x*x` rather than the math library function `pow ( x, 2 )`.
- Use the `goto` statement very sparingly. Structured programming techniques have practically eliminated the need for the `goto` statement. In the opinion of many programmers, the `goto` statement should never be used.
- The `goto` has few uses in general high-level programming, but it can be very useful when C++ code in generated by a program rather than written directly by a person; for example, `gotos` can be used in a parser generated from a grammar by a parser generator. The `goto` can also be important in the rare case in which optimal efficiency is essential, for example, in the inner loop of some real-time application.
- Reduce repetitive computations by only doing them once and saving the result in a temporary variable for future access.
- Reference Scott Meyer's books, *Effective C++: 50 Specific Ways to Improve Your Programs and Design* and *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, on improving C++ program.

4. Appendix A Class Declaration Layout

```
/**
 * Filename: Temperature.h
 * Original Author: somebody
 * Date created: Aug. 1st, 2400
 * Development Org: OHD HSMB
 *
 * Library required: C++ standard library
 *
 * Compiler requirements:
 *
 * This class maintains a temperature value.
 * Reference: convert_temperature.cpp from Jane Doe
 *
 */

#ifndef TEMPERATURE_H
#define TEMPERATURE_H

// SYSTEM INCLUDES
//

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//

// FORWARD REFERENCES
//

class Temperature
{//class Temperature

   public:

/**-------------------------------------------------------
 * Default constructor.
 *-------------------------------------------------------
 */
  Temperature( void );

/**-------------------------------------------------------
 * The standard constructor.  The argument is a value
 * of temperature in C
 *
 * throw an out_of_range exception if the result is below
 * absolute zero
 *-------------------------------------------------------
 */
  Temperature( float const& temperatureC );

/**-------------------------------------------------------
```

```
 * Copy constructor.
 *
 * Make a copy of the Temperature object
 *-----------------------------------------------------------
 */
   Temperature( Temperature const& from );

/**-----------------------------------------------------------
 * Destructor.
 *-----------------------------------------------------------
 */
   ~Temperature( void );

 // OPERATIONS

/**-----------------------------------------------------------
 * Add a Celsius temperature
 *
 * throw an out_of_range exception if the result is below
 * absolute zero
 *-----------------------------------------------------------
 */
   void addC( float const& temperatureC );

/**-----------------------------------------------------------
 * Subtract a Celsius temperature
 *
 * throw an out_of_range exception if the result is below
 * absolute zero
 *-----------------------------------------------------------
 */
   void subtractC( float const& temperatureC );

/**-----------------------------------------------------------
 * Add a Fahrenheit temperature
 *
 * throw an out_of_range exception if the result is below
 * absolute zero
 *-----------------------------------------------------------
 */
   void addF( float const& temperatureF );

/**-----------------------------------------------------------
 * Subtract a Fahrenheit temperature
 *
 * throw an out_of_range exception if the result is below
 * absolute zero
 *-----------------------------------------------------------
 */
   void subtractF( float const& temperatureF );

/**-----------------------------------------------------------
 * Test if the temperature is valid, that is if the
 * temperature is below absolute zero
 *
 * return false if the temperature is below absolute zero
```

```
 *-------------------------------------------------------
 */
   bool isValid( float const& tempK );

 // ACCESS

/**-------------------------------------------------------
 * Set the temperature in Celsius
 *
 * throw an out_of_range exception if the result is below
 * absolute zero
 *-------------------------------------------------------
 */
   void setC( float const& temperatureC );

/**-------------------------------------------------------
 * Set the temperature in Fahrenheit
 *
 * throw an out_of_range exception if the result is below
 * absolute zero
 *-------------------------------------------------------
 */
   void setF( float const& temperatureF );

/**-------------------------------------------------------
 * Set the temperature in Kelvin
 *
 * throw an out_of_range exception if the result is below
 * absolute zero
 *-------------------------------------------------------
 */
  void setK( float const& temperatureK );


// INQUIRY

/**-------------------------------------------------------
 * Get the temperature in Fahrenheit
 *
 *-------------------------------------------------------
 */
  float getF() const;

/**-------------------------------------------------------
 * Get the temperature in Celsius
 *
  *-------------------------------------------------------
  */
  float getC() const;

/**-------------------------------------------------------
 * Get the temperature in Kelvin
 *
 */
  float getK() const;
```

```
// OPERATORS

/* Assignment operator.
 *
 * Assign another temperature object to this object
 *
 * returns a reference to this object.
 *-------------------------------------------------------
 */
  Temperature& operator=( Temperature const& from );

/**-------------------------------------------------------
 * Overloaded '+' operator.
 *
 * add two Temperature objects
 *
 * returns a Temperature object
 *-------------------------------------------------------
 */
   Temperature  operator+( Temperature const& from );

/**-------------------------------------------------------
 * Overloaded '-' operator.
 *
 * Subtract the a Temperature object from this Temperature
 * object
 * returns a Temperature object
 *-------------------------------------------------------
 */
   Temperature operator+( Temperature const& from );

 //static public members

/**-------------------------------------------------------
 * The Celsius temperature zero in Kelvin
 *-------------------------------------------------------
 */
   static const float ABSOLUTE_ZERO_IN_C;

    protected:

    private:

   float _temperatureK; //the temperature value in Kelvin

/**-------------------------------------------------------
 * throw an out_of_range exception if invalid
 *-------------------------------------------------------
 */
   void testValidity( float const& temp )
                                 throw( std::out_of_range);

/**-------------------------------------------------------
 * Converts a Celsius temperature to Fahrenheit
 *
```

```
 * Throw an out_of_range exception if the conversion fails
 * provided for convenience
 *-------------------------------------------------------
 */
   static float convertCtoF( float const& temperatureC )
                               throw( std::out_of_range);

/**-------------------------------------------------------
 * Converts a Fahrenheit temperature to Celsius
 *
 * Throw an out_of_range exception if the conversion fails
 * provided for convenience
 *-------------------------------------------------------
 */
   static float convertFtoC( float const& temperatureF )
                                throw( std::out_of_range);

 };//class Temperature

 #endif  // TEMPERATURE_H
```

5. Appendix B Class Definition Layout

```
/**-------------------------------------------------------
 * Filename: Temperature.cpp
 * Original Author: somebody
 * Date created: Aug. 1st, 2400
 * Development Org: OHD HSMB
 *
 * Library required: C++ standard library
 *
 * Compiler requirements:
 *
 * This class maintains a temperature value.
 * Reference: convert_temperature.h, onvert_temperature.cpp
 *            from Jane Doe
 *
 * The Temperature class is implemented here
 * Filename: Temperature.cpp
 *-------------------------------------------------------
 */

// SYSTEM INCLUDES
//
#include <exception>
#include <string>

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//
#include "Temperature.h"

// FORWARD REFERENCES
//

// LIFECYCLE

//-------------------------------------------------------
Temperature::Temperature( void ):_temperatureK( 0.f ) {};

//-------------------------------------------------------
Temperature::Temperature( float const& temperatureC )
{                                                 //Temperature
   float tempK = temperatureC - ABSOLUTE_ZERO_IN_C;
  testValidty( temp );
  _temperatureK = temp;

}                                                 //Temperature

//-------------------------------------------------------
Temperature::Temperature( Temperature const& from )
{                                                 //Temperature
```

```
  testValidty( from._temperatureK );
  _temperatureK = from._temperatureK;
}                                                  //Temperature

//---------------------------------------------------------

Temperature::~Temperature( void ) {};

//---------------------------------------------------------

void Temperature::addC( float const& temperatureC )
{                                                  //addC
   _temperatureK += temperatureC;
}                                                  //addC

//---------------------------------------------------------

void Temperature::subtractC( float const& temperatureC )
{                                                   //subtractC
   float tempK = _temperatureK - temperatureC;
   testValidty( tempK );
   _temperatureK -= temperatureC;
}                                                   //subtractC

//---------------------------------------------------------

void Temperature::addF( float const& temperatureF )
{                                                  //addF
   _temperatureK += convertFtoC( temperatureF );
}                                                  //addF


//---------------------------------------------------------

void Temperature::subtractF( float const& temperatureF );
{                                                  //subtractF
   float tempK = _temperatureK
                 - convertFtoC( temperatureF );

   testValidity( tempK );
   _temperatureK = tempK;
}                                                  //subtractF

//---------------------------------------------------------

bool Temperature::isValid( float const& temperatureK )
{                                                  //isValid
   return !( temperatureK < 0.0 );
}                                                  //isValid

// ACCESS

//---------------------------------------------------------

void Temperature::setC( float temperatureC )
{                                                  //setC
   float tempK = temperatureC - ABSOLUTE_ZERO_IN_C;
   testValidity( tempK );
```

```
   _temperatureK = tempK;
}                                                    //setC

//-------------------------------------------------------

void Temperature::setF( float const& temperatureF )
{                                                    //setF
   float tempK = convertFtoC( temperatureF )
                 - ABSOLUTE_ZERO_IN_C;
   testValidity( tempK );
   _temperatureK = tempK;
}                                                    //setF

//-------------------------------------------------------

void Temperature::setK( float const& temperatureK )
{                                                    //setK
   testValidity( temperatureK );
   _temperatureK = temperatureK;
}                                                    //setK

// INQUIRY

//-------------------------------------------------------

float Temperature::getF() const
{                                                      //getF
   return convertCtoF( _temperatureK + ABSOLUTE_ZERO_IN_C
                     );
}                                                      //getF


//-------------------------------------------------------

float Temperature::getC() const
{                                                      //getC
   return _temperatureK + ABSOLUTE_ZERO_IN_C;
}                                                      //getC

//-------------------------------------------------------

float Temperature::getK() const
{                                                      //getK
   return _temperatureK;
}                                                      //getK


// OPERATORS

//-------------------------------------------------------

Temperature& Temperature::operator=( Temperature const&
                                                        from )
{                                                    //operator=
  if ( this != &from ) //prevent self assignment
  {                                                     // if
      testValidity( from._temperatureK );
```

```
      _temperatureK = from._temperatureK;
   }                                                    // if
   return *this;
}                                                     //operator=

//---------------------------------------------------------

Temperature Temperature::operator+( Temperature const&
                                                        right )
{                                                     //operator+
   Temperature t;
   t._temperatureK = _temperatureK + right._temperatureK;
   return t;
}                                                     //operator+

//---------------------------------------------------------

Temperature Temperature::operator-( Temperature const&
                                                        right )
{                                                     //operator-
   float tempK = _temperatureK - right._temperatureK;
   testValidity( tempK );
   Temperature t;
   t._temperatureK = tempK;
   return t;
}                                                     //operator-

//---------------------------------------------------------

//static public members

const float Temperature::ABSOLUTE_ZERO_IN_C = -273.16;

//private methods

//---------------------------------------------------------
void Temperature::testValidity( float const& tempK )
                               throw( std::out_of_range )
{                                       //testValidity
   if ( !isValid( tempK ) )
   {                                    //if
      throw std::out_of_range(
             "Temperature is below absolute zero!" );
   }                                   //if
}                                       //testValidity


//---------------------------------------------------------

float Temperature::convertCtoF( float const& temperatureC )
                                  throw( std::out_of_range )
{                                              //convertCtoF
   if ( temperatureC < ABSOLUTE_ZERO_IN_C )
   {                                           //if
     throw std::out_of_range(
                  "Temperature is below absolute zero!" );
```

```
   }                                                  //if
   return ( 9.0 * temperatureC / 5.0 ) + 32.0;
}                                                     //convertCtoF

//-------------------------------------------------------------

float Temperature::convertFtoC( float const& temperatureF )
                                  throw( std::out_of_range )
{                                                     //convertFtoC
   float tC = ( temperatureC - 32.0 ) * 5.0 / 9.0;
   if( tC < ABSOLUTE_ZERO_IN_C )
   {                                  //if
      throw std::out_of_range(
                  "Temperature is below absolute zero!" );
   }                                  //if
   return tC;
}                                                     //convertFtoC
```

References

[1] OHD General Software Development Standards and Guidelines

[2] Bjarne Stroustrup, *The C++ Programming Language*, Special Edition, ISBN 0-201-70073-5, Addison-Wesley

[3] Java Code Conventions

[4] MDL C++ Coding Standards

[5] C++ Coding Convention, http://courses.iicm.edu/programmierpraktikum/skriptum/cplusplus_coding_convention_20040406.pdf

[6] UWYN C++ Coding Standard, http://www.uwyn.com/resources/uwyn_cpp_coding_standard/

[7] C++ Coding Standard, http://www.possibility.com/Cpp/CppCodingStandard.html

[8] C++ Programming Style Guidelines, http://geosoft.no/development/cppstyle.html

[9] FX-ALPHA C and C++ Coding Conventions, http://www-sdd.fsl.noaa.gov/ fxa/manuals/codingGuidelines.html

[10] C++ FAQ LITE Frequently Asked Questions, http://www.parashift.com/c++-faq-lite/

[11] The C Programming Language, Second Edition, Brian Kernighan and Dennis Ritchie