USGS
science for a changing world

Coding Conventions and Principles for a National Land-Change Modeling Framework

Chapter 1 of
Section F, Land-Change Modeling and Analysis
Book 6, Modeling Techniques

Techniques and Methods 6–F1

U.S. Department of the Interior
U.S. Geological Survey

Coding Conventions and Principles for a National Land-Change Modeling Framework

By David I. Donato

Chapter 1 of
Section F, Land-Change Modeling and Analysis
Book 6, Modeling Techniques

Techniques and Methods 6–F1

U.S. Department of the Interior
U.S. Geological Survey

U.S. Department of the Interior
RYAN K. ZINKE, Secretary

U.S. Geological Survey
William H. Werkheiser, Acting Director

U.S. Geological Survey, Reston, Virginia: 2017

Suggested citation:
Donato, D.I., 2017, Coding conventions and principles for a National Land-Change Modeling Framework: U.S. Geological Survey Techniques and Methods, book 6, chap. F1, 30 p., https://doi.org/10.3133/tm6F1.

ISSN 2328-7055 (online)

Contents

Abstract 1
Introduction 1
General Coding Principles and Conventions 2
- Purposes 2
- Programming Style Contrasted with Programming Practice 2
- General Coding Principles 3
- General Coding Conventions 4
- Practical Considerations 4
- Summary Reference Sheet for Coding Conventions and Principles 5

Conventions for Achieving Modularity 5
Naming Conventions 6
- General Naming Conventions 6
- Naming Conventions for Identifiers 8
- Naming Conventions for Files 12
- Naming Conventions for Programs and Modules 13
- Naming Convention for Pure Functions 13
- Summary Reference Sheets for Naming Conventions 13

Ongoing Development of Conventions 13
References Cited 14
Appendix 1. Basis for Limited–`extern` Coding for Modularity 16
Appendix 2. Discussion of the Use of Global Variables 17
Appendix 3. Template for a Module's Base C++ Code 19
Appendix 4. Template for a Module's C++ Header 23
Appendix 5. Summary of National Land-Change Modeling Framework Coding Principles and Conventions 26
Appendix 6. Summary of Naming Conventions 27

Tables

1. Naming conventions applicable to identifiers, files, programs, namespaces, and modules in any programming language....6
2. Naming conventions for C++ identifiers. 9
3. Naming conventions for C identifiers....12
4. Naming conventions for filenames used within operating-system supported file systems....12
5–1. Summary of National Land-Change Modeling Framework Coding Principles....26
5–2. Summary of National Land-Change Modeling Framework Coding Conventions 26
5–3. Summary of National Land-Change Modeling Framework Conventions for Modularity....26
6–1. Summary of naming conventions applicable to identifiers, files, programs, namespaces, and modules in any programming language....27
6–2. Summary of naming conventions for C++ identifiers 27
6–3. Summary of naming conventions for C identifiers....29
6–4. Summary of naming conventions for filenames used within operating-system supported file systems....30

Abbreviations

CPU	central processing unit
FAT32	file allocation table (32-bit)
IDE	integrated development environment
HFS+	Hierarchical File System Plus
ISO	International Organization for Standardization
NLCMF	National Land-Change Modeling Framework
NTFS	New Technology File System
OO	object-oriented
PImpl	pointer-to-implementation
RAII	resource acquisition is initialization
RAM	random-access memory
STL	standard template library
TDS	top-down structured

Coding Conventions and Principles for a National Land-Change Modeling Framework

By David I. Donato

Abstract

This report establishes specific rules for writing computer source code for use with the National Land-Change Modeling Framework (NLCMF). These specific rules consist of conventions and principles for writing code primarily in the C and C++ programming languages. Collectively, these coding conventions and coding principles create an NLCMF programming style. In addition to detailed naming conventions, this report provides general coding conventions and principles intended to facilitate the development of high-performance software implemented with code that is extensible, flexible, and interoperable. Conventions for developing modular code are explained in general terms and also enabled and demonstrated through the appended templates for C++ base source-code and header files. The NLCMF limited-`extern` approach to module structure, code inclusion, and cross-module access to data is both explained in the text and then illustrated through the module templates. Advice on the use of global variables is provided.

Introduction

The U.S. Geological Survey is incrementally developing a software framework for land-change modeling tentatively designated as the National Land-Change Modeling Framework, or NLCMF. This framework will provide reusable, modular computer software in source code, and it will implement software features that enable and facilitate faster development of the next generation of land-change models with a greatly enhanced capability for ready interoperation with other computer models of environmental, social, and economic processes.

Like any software framework (or any software system or complex computer program), the NLCMF requires clear and specific **coding**[1] conventions and principles. Coding conventions and principles are necessary for all but the simplest computer source code in order to make code more consistent, easier for programmers to read and understand, and therefore easier to maintain and modify. Programmers need conventions and principles for coding because code that is acceptable to a machine is not necessarily meaningful to a programmer. The base syntax and rules of computer programming languages permit the development of code that is technically correct and functionally flawless but also nearly incomprehensible to human readers. Experienced programmers almost universally understand this and agree that in order to produce readable, understandable, and maintainable code they must adhere to coding conventions and principles that serve human understanding—even though these conventions and principles are in no way mandated by the syntax or rules of computer programming languages.

This report establishes a set of specific rules for writing computer source code for use with the NLCMF. These rules may be called "coding conventions and principles," or alternatively they may be collectively called a "programming style." Both forms of reference are used in this report. The programming style presented in this report has been developed specifically for use with code written in the C and C++ programming languages. These two languages are the primary programming languages of the NLCMF, though other programming languages may play less prominent roles in the framework.

The publication of these coding conventions and principles for land-change modelers confers two major benefits on the national land-change modeling community. First, it saves the time of the programmers who work in land-change modeling by providing them with a detailed programming style, saving them the time and effort required to develop their own style. Second, it enables different programmers to write code in accordance with a common programming style that supports and promotes sharing and reuse of computer source code. The first of these benefits allows programmers to begin work more quickly and to feel greater confidence in their ability to produce code that potential future collaborators will be able to

[1]The term "coding" as used in this report means developing software by writing computer source code in a formal programming language. The term "programming" is a near synonym that could have been used in this report, but the term "coding" was chosen because it correctly suggests the detailed writing of functions and statements within an overarching system or framework. The connotation of "coding" as drudge work for entry-level programmers is not intended in this report.

understand, use, and modify as required. The availability of a set of coding conventions and principles allows programmers to begin work immediately on the programming tasks at hand rather than first having to put time and thought into the preliminaries of developing a programming style. The second of these benefits allows programmers to read one another's code more easily; and it makes it easier for a programmer to go back and read and understand his or her own code weeks or months after originally writing it. Reading and understanding computer code, especially code written by someone else, is often a necessary task, but also an unusually difficult and time-consuming one, so any expedient that promotes substantially faster and easier comprehension of computer code is essential in promoting collaborative development of land-change models. Coding conventions and principles aid programmers in developing and later reading their own code, and they also facilitate collaborative software development by allowing programmers more easily to read and understand a growing body of code written by various colleagues, collaborators, and contributors.

The distinction between "conventions" and "principles" in this report is consistent with general use of these terms. Though the meanings of the terms "conventions" and "principles" overlap somewhat, it is clear that coding conventions are narrower in scope and more arbitrary than coding principles. For example, the convention that constants in computer code are given relatively short names consisting of all capital letters, such as `PI` and `VOLUME`, is a rule that applies to only a few coding decisions and has no obvious relation to physical reality or logical necessity. By contrast, the principle that parameter values should generally be passed by reference rather than by value applies to far more coding decisions and is derived from the underlying goal of producing faster code. Hence, the first example is a convention and the second is a principle. Not all cases, however, are so clear. Since conventions and principles coexist on a spectrum of rules for coding style, the distinction between coding conventions and coding principles is sometimes artificial or arbitrary. Consequently, this report does not dwell on classification. Of course, in cases in which it is clear that a rule is a convention rather than a principle, the rule will be explicitly called a convention. In particular, naming rules are consistently called conventions in this report.

Programming style places very few constraints on software design because programming style–the set of conventions and principles for coding–is almost entirely independent of the structure and functioning of individual programs, systems of programs, and programming frameworks. This is important because programmers tend to be independent and resistant to limitations on their creativity. The NLCMF programming style has been developed specifically to give programmers as much control as possible over the creative and problem-solving aspects of software development. Further, the near orthogonality of programming style and substantive software design allows style to be prescribed independently of design. Thus the programming style for the NLCMF can be presented in this report even while documentation for substantive NLCMF design is in preparation. This report presents coding conventions and principles for the NLCMF.

General Coding Principles and Conventions

Purposes

The general coding principles for the NLCMF (and to a lesser extent, the general coding conventions) have been chosen to serve two main purposes:

1. The software of the NLCMF must support fast processing and high throughput for computation-intensive land-change models built within the NLCM Framework.

2. The software of the NLCMF should be extensible, flexible, and interoperable to the greatest extent possible without substantially reducing software speed.

The first of these two purposes is important primarily for computation-intensive models, but not for the land-change models that require only modest amounts of computation. The second purpose, however, applies to almost all of the land-change models that might be built with the NLCMF. These two goals are partially contradictory because in some cases programming for fast processing requires introducing a level of complexity that reduces the readability and extensibility of source code. As is true in almost any design activity, designing a programming style requires finding a balance among competing interests. In designing the coding principles for the NLCMF, primacy has been given to fast processing and high throughput, but even so, much emphasis has been placed on producing readable, understandable, and reusable code whenever the cost in software speed is minimal. It is essential that the NLCMF promote and facilitate the development of extensible, flexible, and interoperable software since an effective software framework by definition provides software structure and reusable code that together provide each new programming project undertaken within the framework with a substantial head start.

Programming Style Contrasted with Programming Practice

The primary purpose of this report is to prescribe a programming style for the NLCMF. Programming style addresses the appearance and form of the program code, whereas programming practices are concerned with the substance of data definition and processing. A maximum line length for source code, for example, is almost a purely stylistic prescription. By contrast, the preference for defining class member functions outside of the declaration of the class is mostly in the nature of a programming practice, but because of the effect on

the appearance and form of source code it is also partially a feature of programming style. Because there is overlap among programming style specifications and programming practices, many of the coding principles and conventions presented in this section do imply some features of programming practices. However, as mentioned at the end of the "Introduction," the coding conventions and principles presented in this report do not substantially limit programmers' creativity or range of options for solving programming problems.

Since programming practices are not directly the subject of this report, readers interested in learning more about recommended programming practices should consult other sources such as the following books:

- *"The C++ Programming Language"* (4th ed.) by Stroustrup (2013)
- *"Code Complete; A Practical Handbook of Software Construction"* (2d ed.) by McConnell (2004)
- *"Effective C++; 55 Specific Ways to Improve Your Programs and Designs"* (3d ed.) by Meyers (2005)
- *"Effective Modern C++; 42 Specific Ways to Improve Your Use of C++11 and C++14"* (1st ed.) by Meyers (2014)

A few words of caution regarding general recommendations for programming practices are in order. Practices that may be helpful in developing software with limited requirements for high computational performance may prove harmful in developing computation-intensive software for the NLCMF. In some cases, this is because recommended standards of programming practice emphasize the development of maintainable code and presume that development is performed by a large team working on a large-scale software system. In these cases, recommended practices may promote the use of techniques, such as object-oriented polymorphism, that may reduce code development and maintenance costs at the expense of run-time efficiency. Of course, for many types of software, a modest loss of run-time efficiency is quite acceptable and is well justified by the resulting reductions in development and maintenance costs. In land-change modeling, however, losses of efficiency in storage usage or processing may be acceptable within the ancillary code (such as code for pre-processing data, writing reports, and organizing datasets), but not in data-intensive or computation-intensive core modeling code. Inefficient storage usage may prevent a model from running at all because of limits on available computer memory. Inefficient processing may extend model run durations by hours, days, or even weeks. The goal of achieving the specific performance requirements of the NLCMF should not be undermined by applying programming practices suggested for general use.

General Coding Principles

The following principles should be applied to the NLCMF and to the models built under it:

1. Code should be logically organized into coherent modules. The later section of this report, "Conventions for Achieving Modularity," presents the approach to modularity that has been chosen for the NLCMF. Appendix 1 provides a detailed rationale for sharing of variables and other interaction among independently compiled C and C++ modules.

2. Global *variables* should almost always be avoided in header files (`.hpp`) for modules other than the **main module**[2] for a model.

3. Global *constants* (including enumerators) and inline member and non-member functions may, however, be declared and defined in module header files, although only with care taken not to use too much memory since such constants have local linkage in each module in which they are included and are stored duplicatively.

4. Global constants and variables required in a non-main module, but not used directly by any **client**[3] of this module (such as a model's main module), should be declared and defined in the module's base source-code (`.cpp`) file but not in its header (`.hpp`) file. Appendix 2 explains the issues raised by the use of global variables and provides guidance to programmers for balancing concerns about maintainability with concerns about programming efficiency and software performance.

5. In most cases variables, objects, and classes that need to be available for use in multiple functions within a module can be, and generally should be, declared in a named namespace unique to the module.

6. The specifier "`static`" should not be used in declaring global or module-level variables. It should only be used within functions or classes. In contrast, the specifier "`const`" should be used for most or all immutable variables to inform the compiler to flag unwanted attempts to change values that should not be changed.

Appendixes 3 and 4 provide source-code module templates that implement and illustrate these six general coding principles.

[2]The term "main module" refers to the module of a C++ or C program containing the "main" function as the entry point for the program. A non-main module (or auxiliary module) is any module that does not contain a "main" function.

[3]In these notes, a "client" module is a module which calls a function from a service module or makes use of data defined in the service module. This usage should not be construed to be too confining. Modules may sometimes be peers, with each making use of functions or data in the other.

General Coding Conventions

The following coding conventions should be applied in all NLCMF code and in code for models written to run under the NLCMF:

1. Use inline functions in preference to preprocessor macros.
2. Organize code into modules to enhance readability and reusability of code, and to facilitate independent development and maintenance of the various parts of the NLCMF.
3. Generally, define a class's member functions outside of its declaration.
4. Organize resources and data into classes whenever a class is conceptually appropriate.
5. Avoid cluttering the global namespace.
6. Avoid excessive use of global variables, but do use global variables as necessary to avoid "tramp" variables and to improve computational efficiency. Appendix 2 discusses the appropriate use of global variables.
7. Use namespaces to avoid name collisions and to enable independent development.
8. Use the "`public`," "`private`," and "`protected`" keywords in class declarations as appropriate to restrict changes to class and class-instance data members.
9. Avoid memory leaks by insuring that destructors cannot fail.
10. Use virtual destructors in classes with virtual function members.
11. Pass by reference rather than by value when possible, but use the "`const`" specifier to protect actual parameters in function calls from being changed by the called function.
12. Use object-oriented programming appropriately, but do not use classes or objects where they are conceptually ill-fitting and have no compensating advantage.
13. Unless there is a specific reason to do otherwise, keep source-code lines at 80 characters or fewer.
14. Use white space, indenting, and comments consistently in order to emphasize and enhance the logical structure of source code.
15. Make code easy to understand at a high- and middle-level of detail through rapid visual inspection.
16. Adopt methods (such as aligning parallel elements of code, and using consistent visual spacers) that make source code visually and esthetically appealing; that is, do pretty-print.
17. Carefully structure all code with stylistic consistency.
18. Prefer functions that are written with 50 or fewer lines of code.
19. Emphasize readability and understandability over efficiency when efficiency gains would amount to 5 minutes or less of CPU time per day under expected use.
20. Initialize all variables.
21. Use variable and other names consistently across all functions and modules.
22. Make documentation sufficiently complete to allow for continuing maintenance and management following the departure of all persons originally involved in development.
23. In each module or major function include a top section of comments that provides identifying information, dates of creation and modification, a description of purpose and usage, explanation of any parameters passed to and used by the function or module, and a record of changes.
24. Provide comment lines within each function or module body to explain any aspects of the code, which might be confusing to a maintenance programmer.
25. Avoid excessive commenting within function bodies, but comment freely in header comments for functions.
26. Supplement within-code documentation with an external document that provides a technical programmer's overview of the application and shows how the parts of the application work together. This document should explain any significant design choices.
27. Refer to appendixes 3 and 4 for samples of the structure of within-code documentation for NLCMF modules.

Practical Considerations

The following points of practical advice may save programmers a great deal of time:

1. **Do not optimize prematurely.** The admirable desire of programmers to produce efficient and bug-free code at every step of development can be a serious impediment to progress. Like any other form of perfectionism, premature optimization turns a tractable task into a frustrating and intractable one. It will be helpful to think of a programming project as an evolutionary progression. Programmers should heed the advice of Frederick Brooks[4] and build the first version of a system to throw it away.

[4]See "*The Mythical Man Month—Essays on Software Engineering*" by Frederick Brooks (1995).

2. **Too much advice is worse than none.** This is a corollary of point 1 above. An attempt to apply the voluminous advice available for programming, especially in C++, can result in a cycle of unproductive stasis. Programmers who feel confident of their proficiency in developing effective, efficient working code should not allow themselves to be hobbled by putative "best practices" that are too often phrased as generic criticisms of coding techniques that are both natural and effective for the purpose of producing the initial versions of working code that are essential for progress. Programmers need to recognize and to ignore worst-case advice when the programming task at hand calls for average-case advice. Once code is working, the minority of the code that requires correction or improvement can be revised in accordance with a small selection from the large supply of advice available for handling problem cases.

3. **Methodologies do not always help.** Some development methodologies are useful, especially for development teams, but none is a substitute for imagination and proficiency in converting problems into working code. Form should not be exalted over substance. Low-ceremony methods reduce the amount of effort that goes into creating charts, lists, diagrams, and other artifacts that are discarded when development is complete. In contrast, rich documentation aids development and helps to make a system maintainable when initial development is complete. For the NLCMF, a low-ceremony/rich-documentation style of development is recommended.

4. **Use an integrated development environment (IDE) or programmer's editor.** The use of a programmer's editor (software) is recommended for writing and changing computer source code. Appropriate software delineates C and C++ syntactical categories with varied colors and fonts. Such editor programs are often included in IDEs, but acceptable stand-alone editors are also available.

Summary Reference Sheet for Coding Conventions and Principles

Appendix 5 provides a summary of the NLCMF coding conventions and principles presented in this section. Programmers and other software developers can print appendix 5 on a single sheet of paper for ready reference.

Conventions for Achieving Modularity

There are many ways to achieve modularity in computer code. In order to achieve unity and consistency in the NLCMF, it is therefore necessary to choose an approach and to express it in the form of a set of conventions. The NLCMF conventions for modularity are most clearly presented through the module templates provided in appendixes 3 and 4. The reader is encouraged to study these templates.

The classification of functions within NLCMF modules is largely derived from the ideas of top-down structured (TDS) programming, in which functions are stratified into various levels during the successive refinement of complex procedures. The code structure in the modules is not, however, simply a series of levels. There are groups for initialization functions, specialized processing, accessor functions, and non-standard processing. This structure has been developed with the intention of accommodating all of the functions required in each module within a logical set of subdivisions that programmers should find intuitive. Of course, as the NLCMF is developed, experience may require changes to this function-classification structure.

Of particular importance in determining the modular structure of complex software in C and C++ is the convention for sharing declarations and definitions among multiple translation units, and for enabling access to data or classes declared or defined in other modules. The limited-`extern` convention has been selected for the first generation of the NLCMF. This convention, and the reasons for choosing it, are explained and described in appendix 1.

The use of a modular structure for complex software entails the potential for compilation dependencies among the modules. Compilation dependencies may become a problem as the NLCMF grows and requires longer and longer amounts of time to re-compile models following changes to code. The following techniques may be used to ameliorate the problems due to compilation dependencies:

- Provide scripts to enable the selection of classes and routines into minimal, special-purpose modules.
- Organize selected code into header files.
- Separate class declarations from definitions in header files.
- Use "`#ifndef`" to prevent re-declarations or re-definitions due to multiple references to the same header file by applying the preprocessor technique of the "guarded include," which is shown in the source code of appendix 4.
- Use precompiled headers when possible.
- Use forward declarations judiciously.
- Use object references and pointers when it is not necessary to use objects themselves.
- Use pointers to implementations judiciously.

The identification of the complete set of NLCMF modules, and the determination of the purpose and membership of each module, is an effort that is still underway. This partitioning of code into modules is part of the NLCMF design activity.

Naming Conventions

This section presents rules for creating programmer-defined names in four tables. Table 1 presents rules (conventions) applicable to identifiers in general. Table 2 provides rules specifically for C++ identifiers. Table 3 addresses C identifiers. Finally, Table 4 provides rules especially for file names.

Although the rules presented in this section borrow ideas from prior sets of naming conventions, collectively they are a new set developed specifically for the National Land-Change Modeling Framework. In accordance with standard usage among programmers and others involved in software development, these rules are called naming conventions here. These conventions are necessary (though, by themselves, not sufficient) for the development of human-eye readable, understandable, and maintainable software code. This section presents naming conventions for all of the entities for which programmers must invent names, including variables, collections, functions, structures, classes, files, namespaces, and modules. Since C and C++ are the primary programming languages used with the National Land-Change Modeling Framework, there is a subsection devoted to conventions specific to each language. There are also subsections for general naming conventions, conventions for files, and conventions for programs and modules. An additional subsection discusses the naming of pure functions.

General Naming Conventions

Table 1. Naming conventions applicable to identifiers, files, programs, namespaces, and modules in any programming language.

[The table presents eight general conventions that set bounds on programmer discretion when inventing and forming names, and that also articulate the software qualities to be served by the use of naming conventions. Convention G–3 explains that programmers should occasionally break from conventions when the conventions work more strongly against software quality than in favor of it. The naming conventions apply to all of the other, more specific conventions and are applicable under Windows, UNIX, Linux, or any other UNIX-like operating system. "G" in the convention identifier is an abbreviation for "general"]

Convention Identifier	Convention	Examples

G–1	**Abbreviations:** Avoid the use of abbreviations in names.	**Correct usage:**
		`shortValue`
		`projectTotal`
		
		Incorrect usage:
		`shrtVal`
		`prjTot`

G–2	**Acronyms:** Only use acronyms if they will be readily recognized by all programmers working with the code or the data. An **acronym** is a set of letters extracted from a name to represent it briefly and pronounced as a word rather than letter-by-letter. When an acronym is used within a name, only the initial letter should be capitalized.	**Correct usage:**
		`temporaryRamAddress`
		`regionalLidarData`
		
		Incorrect usage:
		`temporaryRAMAddress`
		`regionalLIDARData`

G–3	**Breaking from convention:** Programmers may break from any convention in order to improve consistency, readability, maintainability, or program functionality. Each instance of a break from convention, however, should be clearly documented through comments in the affected code. This does not mean that conventions may be disregarded without good reasons. Naming conventions and other conventions should be applied as consistently as possible throughout software and system development in order to improve the quality and maintainability of software systems. Programmers should break from conventions only when the conventions would significantly detract from software quality or maintainability.	

Table 1. Naming conventions applicable to identifiers, files, programs, namespaces, and modules in any programming language.—Continued

[The table presents eight general conventions that set bounds on programmer discretion when inventing and forming names, and that also articulate the software qualities to be served by the use of naming conventions. Convention G–3 explains that programmers should occasionally break from conventions when the conventions work more strongly against software quality than in favor of it. The naming conventions apply to all of the other, more specific conventions and are applicable under Windows, UNIX, Linux, or any other UNIX-like operating system. "G" in the convention identifier is an abbreviation for "general"]

Convention Identifier	Convention	Examples

G–4	**Initialisms:** Initialisms in names are treated the same as acronyms. An **initialism** is a set of letters extracted from a name to represent it briefly and pronounced letter by letter; whereas, an acronym is pronounced as if it were a word. Especially in writing, initialisms and acronyms are similar in purpose and effect, so in popular usage many writers and speakers simply use the term "acronym" to include initialisms.	**Correct usage:**
		`firstHtmlTag`
		`nextDvdFile`
		
		Incorrect usage:
		`firstHTMLTag`
		`nextDVDFile`

G–5	**Length:** The ideal length for identifiers is 12 to 24 characters. The ideal length for program and module names is 8 to 16 characters. The ideal length for filenames (excluding any filename extension, such as "`.dat`" or "`.cpp`") is also 8 to 16 characters. *This convention is subject to many exceptions.* It is more important for names to be clear and unambiguous than to be of ideal length. Names longer than 60 characters, however, should almost never be used.	**Correct usage:**
		`FindDerivative`
		`ReportFileG4`
		`ReportModule.cpp`
		`Utilities.h`
		
		Incorrect usage:
		`ComputeDensityFunctionDerivative`

G–6	**Mathematical and statistical values:** Whenever possible, names should correspond as closely as is reasonable to standard mathematical and statistical symbols and terms; and when software implements a computational procedure or algorithm documented outside of code in mathematical symbols, the variable names used in the software should have clear and obvious correspondences to the mathematical variables and symbols used in the outside documentation. In all other respects, mathematical and statistical constants (for example, "`PI`") should follow the general conventions for constants; and mathematical and statistical variables should follow the general conventions for variables, except that brevity is preferred in names for mathematical and statistical values.	**Correct usage:**
		`X`
		`y`
		`a`
		`mean`
		`sigma`
		`correlationMatrix[]`
		`rms`
		`i`
		`BesselFunction()`
		`BOLTZMANN`
		`E`
		`PI`
		`Rsquared`
		`cosine`
		`log16`
		
		Incorrect usage:
		`variable_x`
		`valueOfPi`

Table 1. Naming conventions applicable to identifiers, files, programs, namespaces, and modules in any programming language.—Continued

[The table presents eight general conventions that set bounds on programmer discretion when inventing and forming names, and that also articulate the software qualities to be served by the use of naming conventions. Convention G–3 explains that programmers should occasionally break from conventions when the conventions work more strongly against software quality than in favor of it. The naming conventions apply to all of the other, more specific conventions and are applicable under Windows, UNIX, Linux, or any other UNIX-like operating system. "G" in the convention identifier is an abbreviation for "general"]

Convention Identifier	Convention	Examples

G–7	**Names and meaning (semantics):** A name should be descriptive of the content, use, or meaning of the entity named. A variable name should indicate what the variable contains or how it is used. A function name should indicate what a function does or what value it returns. A file name should indicate the file's contents if possible, or at least be distinctive enough to avoid mistaken file access.
	
	Names should express the ideas and entities in the scientific or business activities that prompted the software to be written, such as a mathematical problem to be solved, a landscape metric to be computed, or a change process to be modeled. Names generally should not derive from programming-language constructs and ideas, although variable types can be indicated by suffixes when such suffixes are helpful in making the code more readable and maintainable. A list of suffixes for C++ code is listed under "Variables and suffixes" in table 2.
		Correct usage:
		`priceCovariance_fd`
		`PopulationProfile_cls`
		`FitRegressionModel()`
		`numCounties`
		`polygon_idnum`
		`expectedValue`
		`DataInput.cpp`
		`landCoverType`
		`ComputeFragmentation()`
		
		Incorrect usage:
		`Class42`
		`myDoubleFloat`
		`CFrag12()`
		`pgmB174_12_14_2011.Jqf.v3r2.cpp`

G–8	**Space characters in names:** Spaces should not be used in identifiers or in filenames, even when the operating system allows spaces in filenames.	**Correct usage:**
		`ReportFileA2`
		`TemporaryReportOutput`
		
		Incorrect usage:
		`Report File A2`
		`Temporary Report Output`

Naming Conventions for Identifiers

The C++ programming language provides native syntactical support for object orientation and generic programming. Accordingly, the naming conventions for C++ computer code must accommodate these particular and important features of C++, and C++ naming conventions should be expected to differ from the conventions developed especially for other programming languages. Because of the complexity and richness of the C++ programming language, C++ computer code is more difficult to read and understand than code written in some other programming languages. Fortunately, however, naming conventions can help to make the meaning and syntactical types of programmer-defined names more readily and immediately discernable, even on brief visual inspection. Thus, naming conventions are more important for developing readable and maintainable code in C++ than for some of the other programming languages. Table 2 presents 23 conventions for C++ computer code, including a set of suffixes that serve to identify variable type.

The C programming language shares much of its syntax with C++, but there are a few differences. Table 3 provides four conventions that apply to C code but not to C++ code. Otherwise, the C++ conventions in table 2 also apply to C code when they are applicable. For example, CPP–2 and CPP–3 do apply to C code because C code can use Boolean variables; but CPP–1 and CPP–4 do not apply to C code because C does not have syntactical support for classes.

Table 2. Naming conventions for C++ identifiers.

["CPP" in the convention identifier means C++ code]

Convention Identifier	Convention	Examples
CPP–1	**Accessor functions:** Member functions in classes used to set values for class members or to retrieve values from class members should be named with "`Get`" or "`Set`" as the first word in the function name.	`GetCountyArea()` `SetCellPopulation()`

CPP–2	**Boolean constants:** Identifiers for Boolean constants should be set in all uppercase letters and should be short.	`TRUE`
		`FALSE`
		`ON`
		`OFF`

CPP–3	**Boolean values:** Identifiers for variables holding Boolean values should be of moderate length and easy to understand and remember. The name itself should clearly indicate the meaning of a true or false value assigned to the variable. These identifiers should be set in all uppercase letters and may include numerals. Break characters may be inserted for readability.	**`Correct usage:`**
		`IS_AVAILABLE`
		`FILE_EXISTS`
		`NOT_FLOAT32`
		
		`Incorrect usage:`
		`Avail`
		`FEXIST_YN`
		`NF32`

CPP–4	**Class variables:** A variable declared within a class with storage class "`static`" is a class variable. Unlike an instance variable, a class variable holds a single value, so all instances of the class which access a class variable are accessing the same variable. The identifier for a class variable should be suffixed with "`_cv`".	`printerStatus_cv`
		`modelInstanceCount_cv`
		`unitOfLength_cv`

CPP–5	**Classes:** Identifiers for classes are in mixed-case beginning with an uppercase letter. Optionally these identifiers may be suffixed with "`_c`".	`Month`
		`ExtendedPrecisionInt`
		`AgentProfile`
		or
		`AgentProfile_c`

CPP–6	**Constants:** Identifiers for most constants should be in all uppercase letters with underscores separating words.	`MAX_CELL_INDEX`

CPP–7	**Enumerators:** Identifiers for enumerators (named integer constants defined by an enumeration) should be in all uppercase letters with underscores separating words.	`RED`
		`JANUARY`
		`BACKWARDS`
		`COMMERCIAL_GROWTH`

CPP–8	**Functions:** Function names should be in mixed case beginning with a capital letter. The name should be a verb phrase (or some form of verbal construction) describing what the function does. Since some functions have side effects and others do not, the verb phrase may emphasize either the side effects (processing) or the value returned by the function.	`FindMaximumTestScore()`
		`ComputeSecondDerivative()`
		`PrintSummary()`

CPP–9	**Mathematical and statistical values (other than constants):** In addition to complying with the general conventions applying to mathematical and statistical values, C++ identifiers for such values should be kept simple by grouping them within namespaces or classes.	`Namespace basic_statistics`
		`{`
		`  double mean, variance;`
		`  int    N;`
		`}`

Table 2. Naming conventions for C++ identifiers. —Continued

["CPP" in the convention identifier means C++ code]

Convention Identifier	Convention	Examples
CPP–10	**Member functions:** The functions which are members of a class are named according to the general conventions for functions but their names may optionally be suffixed with "_cm".	`FindAreaOfPolygon_cm()`
CPP–11	**Namespace aliases:** Namespace aliases should each consist of a string of numerals and lowercase letters not exceeding six characters.	`math1` `gisft`

CPP–12	**Namespaces:** C++ namespace names should be in lowercase letters with words separated by underscores. Optionally, a namespace name may be suffixed by "_ns". Generally, a namespace name should be a single word if possible; a namespace name should be relatively short, preferably fewer than 12 characters.	`math_functions_set_1`
		`gis_file_types`
		`basic_statistics_ns`

CPP–13	**Objects:** Objects generally follow the conventions for variables, but in addition, an object name should include the class name along with other words or characters to distinguish the object from other instantiations of the same class.	`monthFyBegin`
		or
		`monthFyBegin_obj`
		`agentProfile`
		or
		`agentProfile_obj`

CPP–14	**Functions, procedures, and subroutines:** In C++ the term "function" refers to functions, procedures, and subroutines. The terms "procedure" and "subroutine" are not part of the formal syntax of C++. Currently, no naming convention is specified to distinguish pure functions from other subroutines.	

CPP–15	**Pointer:** The identifier for a pointer generally should be suffixed by "_ptr", but optionally may begin with "ptr".	`interimCount_ptr`
		or
		`ptrInterimCount`

CPP–16	**Types:** Identifiers for user-defined types (not built-in types) are in mixed case beginning with a lowercase letter and suffixed with "_t".	`int12_t`
		`cell_t`
		`smallRaster_t`

CPP–17	**Variables for a fixed number of entities:** A variable indicating a fixed number of items or entities should begin with "num" followed by the name of the set of items or entities. An identifier prefixed with "num" names a variable with a value not known at compile time but unlikely to change value during the course of processing after initialization. By contrast, an identifier suffixed with "_count" names a variable with a value subject to change during processing; a variable suffixed with "_idnum" contains an identifying number that is without numerical significance; and a variable with a fixed value known at compile time should be coded as a constant.	`numCounties`
		`numCells`
		`state_idnum`
		`current_population_count`
		`NUMBER_STATES_IN_US`

CPP–18	**Variables in general**: A variable name should be a noun or noun phrase naming the value represented by the variable.	`xCoordinate` `maximumEmployment`
CPP–19	**Variables in mixed case:** A variable name should be in mixed case beginning with a lowercase letter. Underscores should only be used when required for clarity or readability.	`countyBoundaryType` `currentAC_Amperes`

Table 2. Naming conventions for C++ identifiers. —Continued

["CPP" in the convention identifier means C++ code]

Convention Identifier	Convention	Examples
CPP–20	**Variables pluralized:** An identifier for an array, list, or other set of multiple values should generally be pluralized.	`productCodes[200]` `pixels[400][5000]`
CPP–21	**Variables in global scope:** Global variables may be identified either by the scope resolution operator (`::`) or a suffix (`_g`).	`::category` `category_g`
CPP–22	**Variables for indexes and algebraic variables:** The names i, j, k, l, m, n, o, p should generally be reserved for designating integer indexes for arrays or loops; or for subscripts for indexed mathematical constructs. The names a, b, c, d, e, f, g and s, t, u, v, w, x, y, and z should generally be used for floating-point (real) variables in computational code.	
CPP–23	**Variables and suffixes:** The type of variable may optionally be indicated by a suffix.	

Suffix	Type of variable	Examples
`_c or _cls`	class	`Point_c` or `Point_cls`
`_count`	the number of items or entities currently found in the data; how many of an item exist at a particular time	`Counties_count`
`_f`	float (single precision)	`average_f`
`_fd`	double (double precision)	`average_fd`
`_ld`	long double (long double precision)	`derivativeOfX_ld`
`_g`	global	`year_g`
`_i`	int	`sampleSize_i`
`_il`	long int	
`_index`	an index variable used as a subscript for an array	`year_index`
`_cm`	member of class	`AdjustRetailPrice_cm()`
`_cv`	class variable	`retailPrice_cv`
`_ns`	namespace	`phase_I_ns`
`_idnum`	the identifying number associated with a value or entity	`County_idnum`
`_obj`	object	`employerRetail_obj`
`_ptr`	pointer	`NextAction_ptr`
`_str`	string	`changeName_str`
`_t`	type (not built in)	`myPartRecord_t`
`_tf`	Boolean value	`WIN_tf`

Table 3. Naming conventions for C identifiers.

["C" in the convention identifier means C code]

Convention Identifier	Convention	Examples
C–1	**C++ conventions:** The naming conventions for C code are the same as those for C++ code except as described in this table or where C++ constructs (such as classes and namespaces) are not available in C.	
C–2	**Struct instance:** Identifiers for specific instances of struct types should be in mixed case with the initial letter in lowercase (just like objects in C++). Such identifiers may optionally be suffixed by "`_struct`".	`imageProfileMain` or `imageProfileMain_struct`
C–3	**Struct type name:** Identifiers for a struct type should be in mixed case with the initial letter capitalized (just like identifiers for classes in C++). Such identifiers may optionally be suffixed by "`_struct_t`".	`ImageProfile` or `ImageProfile_struct_t`
C–4	**Variables in global scope:** The identifier for a global variable should be suffixed by "`_g`".	`totalSumOfSquares_g`

Naming Conventions for Files

During the early stages of the development of the NLCMF, the conventions for naming files should not be too limiting. Developers should enjoy relative freedom in naming files until code is ready to be merged into the National Land-Change Modeling Framework, so for the present, the conventions related to file naming remain broad and liberal. As the NLCMF matures, a distinction will be made between the filenames used within operating-system supported file systems and more general types of filenames. As it is developed, the NLCMF will provide a catalog of files and other named entities in order to prevent name collisions and to help both developers and users to keep track of named entities. Therefore, developers should not worry much about making the names of files within file systems meaningful because the NLCMF catalog will supplement operating-system supported file system filenames with full descriptive information.

Table 4. Naming conventions for filenames used within operating-system supported file systems.

["F" in the convention identifier is an abbreviation for "filename"]

Convention Identifier	Convention	Examples
F–1	**Aliases:** Filename aliases are not used because they are not supported on all mainstream operating systems.	

F–2	**Characters used in filenames:** Each file name must begin with a letter, which may be either lowercase or uppercase. Names may include lowercase and uppercase letters, numbers 0 to 9, and the two special characters "`.`" and "`_`". The space character may not be used in filenames.	**Correct usage:**
		`Region04_rasterA12.hsf.aux`
		`CatalogC`
		`UTIL_05b.gif`
		
		Incorrect usage:
		`_UtilityData.txt`
		`42Temp.cpp`
		`TextRes$V12.res`

F–3	**Directories and folders:** The names used for directories and folders should follow the same conventions as names for ordinary files. In addition, directory and folder names should not exceed 30 characters.	

Table 4. Naming conventions for filenames used within operating-system supported file systems.—Continued

["F" in the convention identifier is an abbreviation for "filename"]

Convention Identifier	Convention	Examples
F–4	**Length:** Filenames should generally be in the ideal range of 8 to 16 characters (excluding extensions such as `.dat` or `.cpp`). Although this convention is subject to many exceptions, filenames including extensions longer than 48 characters are strictly prohibited.	
F–5	**Operating system independence:** Filenames should be usable on any current mainstream operating system for personal computers, workstations, or servers released or significantly upgraded during the past decade, including Windows, OS-X, Linux, and various releases of UNIX. In particular, filenames should be usable on FAT32, NTFS, ext3, ISO 9660, and HFS+ file systems.	
F–6	**Similarity of filenames:** The ideal of meaningful names that clearly identify the contents of files is difficult to achieve and conflicts with the goal of imposing machine-processable structure on a potentially large collection of files. Therefore, in general, it is preferable that names of related files reflect the similarity through structure in names.	

Naming Conventions for Programs and Modules

In this edition of the NLCMF coding principles and conventions, the naming conventions for programs and modules are the same as those for other files, except that names for programs must follow any operating-system specific requirements, such as having a filename extension like "`.exe`" appended.

Naming Convention for Pure Functions

A pure function has no side effects and must return a value. Some non-pure functions return values, and some do not, but all have side effects. The C and C++ languages do not make a clear syntactic distinction between pure and non-pure functions except in the case of functions declared to return no value. Distinguishing in source code between pure and non-pure functions may or may not prove beneficial. On one hand, a distinction might aid programmers in reading and understanding source code; however, the distinction will also make code more difficult to read, and it may lead to errors and confusion because the convention that is used to make the distinction would not be checked by the compiler. It is difficult to know *a priori* whether the distinction would be helpful or harmful on balance.

A pure function could be identified with a suffix, as in

```
ComputeDerivative_pf( )
```

or, it might be identified by a prefix, as in

```
PUREcomputeDerivative( ).
```

For this version of the NLCMF coding principles and conventions, no convention is recommended for distinguishing pure from non-pure functions. It is recommended but not required, however, that within-code documentary comments for all functions identify a function as pure or non-pure, and that external documentation for any reusable C or C++ function identify it as pure or non-pure.

Summary Reference Sheets for Naming Conventions

Appendix 6 provides a summary of NLCMF naming conventions. Programmers and other software developers may print out this appendix for ready reference.

Ongoing Development of Conventions

This initial programming style for the NLCMF will change as the NLCMF evolves. The following kinds of changes are expected:

- As specialized classes are developed to serve the specific purposes of the NLCMF, additional naming conventions will be developed to show relationships among related classes and objects.

- The generic-programming capabilities of C++ will be used to develop a specialized template library for the NLCMF. Additional conventions for names and parameters in this template library will be required.
- The use of collection types from the standard template library (STL), such as vectors and lists, may become extensive enough to require conventions to help organize and manage collections of these specialized data types.

As a data-management facility is developed, standards and conventions will be needed for managing data and controlling information for numerous modeling runs.

Updated conventions may be provided in subsequent updates.

References Cited

Brooks, F.P., Jr., 1995, The mythical man-month; Essays on software engineering (1st ed.): Reading, Mass., Addison-Wesley Professional, 322 p.

McConnell, S.C., 2004, Code complete; A practical handbook of software construction (2d ed.): Redmond, Wash., Microsoft Press, 914 p.

Meyers, Scott, 2005, Effective C++; 55 specific ways to improve your programs and designs (3d ed.): Boston, Mass., Addison-Wesley Professional, 297 p.

Meyers, Scott, 2014, Effective modern C++; 42 specific ways to improve your use of C++11 and C++14 (1st ed.): Sebastopol, Calif., O'Reilly Media, Inc., 336 p.

Stroustrup, Bjarne, 2013, The C++ programming language (4th ed.): Boston, Mass., Addison-Wesley Professional, 1,346 p.

Appendixes

Appendix 1. Basis for Limited-`extern` Coding for Modularity

Two different conventions for module structure that were considered for the NLCMF and its client models are as follows:

1. **Explicit-`extern` approach**—The header (`.hpp`) file for a particular module can be included only in the base source-code (`.cpp`) file for this module itself, not in the header or base source-code files for any other modules, such as the main module. For simplicity in the description of this approach, a module that needs to use data from another module is called the "client," and the module that defines and holds the data is called the "server." Then in this approach, if a client module needs to use a variable defined in a server module, the variable must be declared with the "`extern`" specifier in the client module, and the variable must be initialized in the server module. A disadvantage of this approach is that it forces the programmer of the client module to choose variables and to declare them explicitly in the client-module code. This disadvantage can be partially overcome by providing a supplementary header file (perhaps with filename extension `.hps`) for the particular module in order to package and provide the `extern` declarations typically used by client modules.

2. **Limited-`extern` approach**—The header (`.hpp`) file for a particular server module is intended to be included in the base source-code (`.cpp`) file for this module itself and also in the base source-code file for any of its client modules. Any global variables used within the particular server module but not required by client modules must be declared and defined in the base source-code (`.cpp`) file for the server module rather than in its header (`.hpp`) file. The header file for the server module should use the preprocessor directives `#ifndef`, `#define`, and `#endif` to prevent re-inclusion of its contents when the contents have been previously included in a translation unit. The use of these three preprocessor directives in this way in a header file is termed a "guarded include," and this technique is illustrated in the header-file template in appendix 4. Under the limited-`extern` approach:

    a. Header files contain "declarations" for variables, functions, and classes; and contain "**definitions**"[1] of inline functions (both member and non-member);

    b. Header files contain "`consts`" and "`typedefs`" referenced by client modules on the understanding that this will result in duplicative storage for the consts;

    c. Base source-code files contain definitions of non-inline member and non-member functions, and contain declarations and definitions of each module's internal global variables;

    d. Clients may access data internal to another module through the module's accessor functions, generally avoiding direct access; and

    e. Base source-code files for client modules may contain declarations and definitions for the client module's global variables, which may be made available to the functions in service modules through pass-by-reference (avoiding the use of `extern` declarations within headers or base source-code for service modules).

This approach simplifies the programmer's effort and may prevent some programming errors. In its simplest form, however, it may require the use of additional random access memory (RAM) because of the inclusion of inline function definitions in client modules that do not require them (depending on the ability of the build environment to identify and then to eliminate unreferenced functions). If this becomes a problem, the amount of RAM required can be reduced by creating specialized header files.

These two approaches to module structure are based on the following features of the C++ programming language:

1. The linkage of "`consts`" and "`typedefs`" is internal by default. This means that the same identifier may be used for otherwise unrelated `consts` or `typedefs` in two or more modules.

2. In older C++ compiler implementations (and possibly in some current implementations), variables declared at module level with the specifier "`static`" are given internal linkage.

3. In order to have the possibility of being incorporated into machine code without function calls, an "`inline`" function must be defined identically in each module (translation unit) in which it is used.

4. The linker treats the global scope (or global namespace) of all modules linked together as the same scope and namespace.

The second of these approaches, the "limited-`extern`" convention, has been chosen for the first version of the NLCMF. The application of this convention has been incorporated into the module templates in appendixes 3 and 4. These templates provide a compact illustration of the practical meaning and implementation of the limited-`extern` approach.

[1] A "definition" may take the form of assignment of values within a function to which the variable (or array) is passed by reference.

Appendix 2. Discussion of the Use of Global Variables

Dogmatic, blanket proscriptions of global variables are endemic in the literature of programming principles and practices. The undiscriminating global deprecation of the use of global variables in software written in a high-level procedural programming language, however, is a disservice to programmers because it too often subjects them to criticism for their valid and necessary situational exercise of professional judgment. When a programmer faces a programming problem or task, the question to be answered *is not* how to program within the confines of rules and putative best practices imposed on top of the functional and syntactic restrictions of the development and build environment. Rather, it is how best to meet the requirements of the problem or task efficiently and effectively. If the requirement is to write a single-use program as quickly as possible, then any arguments about software maintainability are inapplicable. Even if a program will be used repeatedly, the effort expended to make it maintainable will be wasted if there is no requirement ever to change the program, or if it is a simple program that will be modified only once or twice over several years. There certainly are good arguments for programmers to use global variables cautiously in developing software that needs to be packaged for widespread source-code use, but if software is packaged for distribution as precompiled modules, then the syntactically global variables within the translation unit for each module are effectively local variables—local to their respective modules. Even when the task at hand requires that maintainable source code be developed for frequent update by multiple developers, global variables may be easier to understand than chains of **tramp**[2] parameters traversing multiple levels of function calls.

There is no dispute that the use of global variables can reduce the understandability and maintainability of some programs. There should, however, also be no dispute that a program design intended to achieve an appropriate and acceptable balance of performance, understandability, and maintainability depends both on the character of the program and on the real-world context of its development and use. The widely asserted rule discouraging the use of global variables needs to be qualified through two broad observations, which include:

1. The general deprecation of the use of global variables may be appropriate for relatively pure object-oriented programming, but it is not appropriate for other programming paradigms (including mixed paradigms), especially not for top-down structured (TDS) programming. TDS is an appropriate way of programming process-simulation land-change models, and it is the presumptive choice for model structure in the NLCMF.

2. The effects and implications of the use of global variables differ according to software features other than just the visibility and scope of variables within a translation unit.

If almost all of the processing in an object-oriented program is carried out through the member functions of classes, and if as a result there are relatively few levels of function calls within the body of this object-oriented (OO) program, then global variables should be avoided. In this case, global variables indicate that at least one class is missing, and that the missing class or classes should define objects used for data intermediation among the other classes. However, in other design patterns or paradigms such as the TDS approach that is favored in the NLCMF, the argument against the use of global variables promotes unnatural alternatives such as lengthy parameter lists. The argument against global variables depends in large part on the assumption that all functions should achieve low coupling with high cohesion. In TDS software, however, this assumption is invalid because TDS software must allow successive refinement of processes that are specific and often unique to the TDS program. Each logical unit of processing in TDS software is generally composed of multiple levels or layers of C/C++ functions, and the desirable low coupling and high cohesion applies at the logical level among logical units, not at the syntactic level within each logical unit. Other than a few primitive functions called from other functions within a logical unit of TDS software, most of the functions in a TDS logical processing unit are containers for process-specific control-flow code and program-specific data processing or computation, and these functions are poor candidates for code reuse or generalization. Consequently, the emphasis when writing a TDS logical processing unit is to create correct, maintainable, but probably not reusable code. In order for a TDS design to be understandable and therefore maintainable, it should clearly identify and group the data used by each logical processing unit, and this generally is best done in C or C++ through syntactically global variables.

While there are legitimate concerns about naming confusion and naming collisions caused by the use of global variables in software that consists of multiple modules, in C++ programs these concerns can be effectively remedied through the use of the following:

- Suitable naming conventions;
- The scope-resolution operator ("`::`");
- Namespaces;
- Use of the "`const`" specifier for immutable variables;
- Routine declaration of class members as private or protected;
- Pointers;
- Modular structure;
- Careful construction of header files; and
- Standard-library containers.

[2] A "tramp**"** parameter is a parameter that is passed through several levels of functions but not used in all levels through which it is passed.

For many C++ programs, the first resort, and the simplest and most understandable and maintainable way to handle global variables, is through declarations that group global variables in one or more namespaces. Typically, each module will have its own dedicated namespace. There will also be a few namespaces that are used in more than one module.

To summarize for the NLCMF, the conventions and principles for use of global variables are as follows:

- Except in the main module for a model, the declaration of global variables should be avoided in header files;
- Global variables should be grouped within namespaces and used as necessary to prevent unnaturally long parameter lists and tramp parameters; and
- Global variables should be used sparingly in base source-code files (`.cpp` files) but as much as necessary to improve space and time efficiency.

Appendix 3. Template for a Module's Base C++ Code

```
/**************************************************************************
 THIS IS a C++ source-code template. It may be copied into a programmer's
    ASCII editor as a template for a new module.
**************************************************************************

  System Name:        National Land-Change Modeling Framework (NLCMF)

  Version:            1.0

  Module Name:        Module_Name

  File Name:          Module_Name.cpp

  Lead Author:        David I. Donato, Research Computer Scientist
                      Eastern Geographic Science Center
                      U.S. Geological Survey

  Other Authors:      --

  Date First Used:    N/A

  Modification Dates: May 8, 2015


  Description:

   This is the Base Source-Code for NLCMF module "Module_Name". It provides
   functions, classes, and data used to [text to be supplied].

   This module is used by [text to be supplied].


  Notes:
    [1] As much as is possible without unduly delaying or complicating
        development, the techniques of RAII (Resource Acquisition Is
        Initialization) should be used in allocating and initializing
        resources such as file pointers. The goal of efficient and
        comprehensible process simulation, of course, takes precedence
        over protection against memory leaks or unexpected occasional data
        loss during abnormal termination.
    [2] The Pointer-to-Implementation (or "PImpl") design technique should
        be used in the event that the inclusion of classes and their
        member functions use memory that must be freed for other uses in
        order to allow the software to run on the target hardware. In
        most cases, however, the extra RAM required to include classes and
        their member functions are justified by the gain in clarity and
        simplicity of the source code when the indirection and layering
        associated with Pointer-to-Implementation are avoided.

  Modifications:


**************************************************************************/
```

```
/*****************************************************************************/
/* [1] PRE-PROCESSOR DIRECTIVES AND GLOBAL "USING" DIRECTIVES                */
/*****************************************************************************/

#include <algorithm>
#include <fstream>
#include <iomanip>
#include <ios>
#include <iostream>
#include <map>
#include <memory>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

#include "ModuleName.hpp"

using namespace modalias;

/*****************************************************************************/
/* [2] DEFINITIONS FOR NON-INLINE STRUCT AND CLASS MEMBER FUNCTIONS          */
/*****************************************************************************/

namespace mod_name_ns
{

}

/*****************************************************************************/
/* [3] DECLARATIONS OR DEFINITIONS FOR FILES AND GLOBAL FILE-RELATED VARIABLES*/
/*****************************************************************************/
/* Clients may access data internal to this module through accessor functions.*/
/* Data internal to this module may be made available to service modules by   */
/* passing by reference.                                                     */
/*****************************************************************************/

namespace module_name_ns
{
 /* USE GLOBAL VARIABLES SPARINGLY FOR SPACE OR TIME EFFICIENCY. */
}

/*****************************************************************************/
/* [4] DECLARATIONS OR DEFINITIONS FOR GLOBAL VARIABLES, ARRAYS, CONTAINERS,  */
/*     AND OBJECTS.                                                          */
/*****************************************************************************/
/* Clients may access data internal to this module through accessor functions.*/
/* Data internal to this module may be made available to service modules by   */
/* passing by reference.                                                     */
/*****************************************************************************/

namespace module_name_ns
{
/* USE GLOBAL VARIABLES SPARINGLY FOR SPACE OR TIME EFFICIENCY. */
}
```

```
/*****************************************************************************/
/* [5] DEFINITIONS FOR OTHER FUNCTIONS (NOT CLASS OR STRUCT MEMBERS) BY       */
/*     PROCESSING LEVEL (LEVELS A THROUGH E)                                 */
/*****************************************************************************/
namespace mod_name_ns
{

/*****************************************************************************/
/* Level A - Initialization                                                  */
/*****************************************************************************/



/*****************************************************************************/
/* Level B[1-5] - Primary Processing Capabilities                            */
/*****************************************************************************/



/*****************************************************************************/
/*  Level B1 - Top-Level Executive Control and Supervision                   */
/*****************************************************************************/



/*****************************************************************************/
/*  Level B2 - High-Level Processing                                         */
/*****************************************************************************/



/*****************************************************************************/
/*  Level B3 - Mid-Level Processing                                          */
/*****************************************************************************/



/*****************************************************************************/
/*  Level B4 - Low-Level Processing                                          */
/*****************************************************************************/



/*****************************************************************************/
/*  Level B5 - Primitive Functions                                           */
/*****************************************************************************/
```

```
/*****************************************************************************/
/* Level C - Specialized Processing                                          */
/*****************************************************************************/

////////////////////////////////////////////////////////////
// C: DoSampleActions01                                   //
//--------------------------------------------------------//
// Perform Version 1 of some sample actions in order to   //
// illustrate the convention for function coding within   //
// a code Level.                                          //
////////////////////////////////////////////////////////////

int DoSampleActions01(string str2)
{
 // This is a dummy function illustrating coding style.
 //
 cout << "((((( Sample Action 01 " << str2 << ")))))" << endl;
 return 0;
}


/*****************************************************************************/
/* Level D - Accessor Functions                                              */
/*****************************************************************************/



/*****************************************************************************/
/* Level E - Non-Standard Processing                                         */
/*****************************************************************************/


}

/*****************************************************************************/
/*****************************************************************************/
/*****************************************************************************/
```

Appendix 4. Template for a Module's C++ Header

```
/*****************************************************************************
 THIS IS a C++ source-code template. It may be copied into a programmer's
    ASCII editor as a template for a new module.
*****************************************************************************

  System Name:        National Land-Change Modeling Framework (NLCMF)

  Version:            1.0

  Module Name:        Module_Name

  File Name:          Module_Name.hpp

  Lead Author:        David I. Donato, Research Computer Scientist
                      Eastern Geographic Science Center
                      U.S. Geological Survey

  Other Authors:      --

  Date First Used:    N/A

  Modification Dates:  May 8, 2015


  Description:

   This is the header file for the NLCMF module "Module-Name".
   [More text to be supplied]

   This header file provides declarations for the classes, functions,
   and other entities of module "Module_Name.cpp". This header may also
   include definitions for constants, enumerations, and inline functions
   (both member and non-member).


  Notes:
    [1] TBD
    [2] ...


  Modifications:


*****************************************************************************/

// The following guarded include prevents more than one inclusion of this code
// within a single module.

#ifndef MODULENAME
#define  MODULENAME
```

```
/*****************************************************************************/
/* [1] GLOBAL CONSTANTS, TYPEDEFS, ENUMERATIONS, and NAMESPACE ALIASES       */
/*****************************************************************************/
/* Variables declared "const" have local scope and will require separate     */
/* storage space in each module that includes this header file.              */
/*****************************************************************************/

namespace module_name_ns
{
}
namespace modalias = module_name_ns;

/*****************************************************************************/
/* [2] DECLARATIONS FOR STRUCTS AND CLASSES                                  */
/*****************************************************************************/

namespace module_name_ns
{

}

/*****************************************************************************/
   /***************************************************************/
   /* Except in the main module for a model, the declaration of global */
   /* variables should be avoided in header files.                  */
   /***************************************************************/
   /*****************************************************************************/
   /* [3] DECLARATIONS FOR FILES AND GLOBAL FILE-RELATED VARIABLES               */
   /*****************************************************************************/
//  namespace module_name_ns
//  {
//  /* AVOID DECLARING OR DEFINING GLOBAL VARIABLES IN HEADER FILES. */
//  }
   /*****************************************************************************/
   /* [4] DECLARATIONS FOR GLOBAL VARIABLES, ARRAYS, CONTAINERS, AND OBJECTS  */
   /*****************************************************************************/
//  namespace module_name_ns
//  {
//  /* AVOID DECLARING OR DEFINING GLOBAL VARIABLES IN HEADER FILES. */
//  /* RARELY, LARGE DATA ENTITIES MAY BE DECLARED IN THE HEADER FOR */
//  /* DIRECT USE BY CLIENT MODULES.                                 */
//  }
   /***************************************************************/
/*****************************************************************************/



/*****************************************************************************/
/* [5] DECLARATIONS AND DEFINITIONS OF MEMBER AND NON-MEMBER INLINE FUNCTIONS */
/*****************************************************************************/

namespace module_name_ns
{

}
```

```
/*****************************************************************************/
/* [6] FORWARD DECLARATIONS FOR OTHER FUNCTIONS (FUNCTION PROTOTYPES)      */
/*****************************************************************************/

namespace module_name_ns
{

// Level A - Initialization           //
//-------------------------------------


// Level B[1-5] - Primary Processing  //
//                Capabilities        //
//-------------------------------------//


// Level B1 - Top-Level Executive     //
//            Control and Supervision //
//-------------------------------------


// Level B2 - High-Level Processing   //
//-------------------------------------


// Level B3 - Mid-Level Processing    //
//-------------------------------------


// Level B4 - Low-Level Processing    //
//-------------------------------------


// Level B5 - Primitive Processing    //
//-------------------------------------


// Level C - Specialized Processing   //
//-------------------------------------


// Level D - Accessor Functions       //
//-------------------------------------


// Level E - Non-Standard Processing  //
//-------------------------------------


}
#endif // End of guarded include.
```

Appendix 5. Summary of National Land-Change Modeling Framework Coding Principles and Conventions

Table 5–1. Summary of National Land-Change Modeling Framework (NLCMF) Coding Principles.

Place inline functions and global constants in `.hpp` header files.	**Avoid** global variables in `.hpp` header files except in the main module of a model.
Declare and define a module's global variables in its `.cpp` base source-code file.	**Do not use** the specifier "`static`" anywhere except within functions and classes.
Declare variables, objects, and classes used in more than one function in a unique, named namespace.	The *main idea* is to share declarations of functions, classes, and structs among all modules through `.hpp` headers but to encapsulate each module's data in its `.cpp` base source-code file.

Table 5–2. Summary of National Land-Change Modeling Framework (NLCMF) Coding Conventions.

Define most class member functions outside of the class declaration.	**Avoid** pre-processor macros; prefer inline functions.
Use namespaces to enable independent development and to avoid name collisions; **declare** most class data members "private" or "protected."	**Avoid** cluttering the global namespace but do use global variables as needed to avoid "tramps."
Use classes to organize and encapsulate data, file handles, and other resources when appropriate.	**Avoid** conceptually ill-fitting classes; encapsulate with modules and namespaces to avoid confusion.
Use virtual destructors in classes with virtual member functions.	**Do not write** destructors that can fail.
Use variable and other names consistently across all modules.	**Avoid** pass-by-value in favor of pass-by-reference.
Format source code for esthetic appeal and readability.	**Avoid** lines over 80 characters.
Prefer functions with 50 or fewer lines of code.	**Do not postpone** preparing external documentation.
Richly annotate functions with prefatory comments.	**Avoid** excessive comments within function bodies.
Initialize all variables and **declare** immutable variables "`const`".	**Avoid** complexity that achieves only minor performance gains.

Table 5–3. Summary of National Land-Change Modeling Framework (NLCMF) Conventions for Modularity.

Use the module templates.	**Avoid** unnecessary forward declarations.
Use precompiled headers when possible.	**Avoid** re-declarations by using guarded includes.
Use pointers to implementations judiciously.	

Appendix 6. Summary of Naming Conventions

Table 6–1. Summary of naming conventions applicable to identifiers, files, programs, namespaces, and modules in any programming language.

[The naming conventions are applicable under Windows, UNIX, Linux, or any other UNIX-like operating system. "G" in the convention identifier is an abbreviation for "general"]

Convention Identifier	Convention	Examples
G–1	**Abbreviations:** Avoid using in names.	`shortValue (not shrtVal)`
G–2	**Acronyms:** Use only if well known; capitalize the first letter only.	`temporaryRamAddress` `regionalLidarData`
G–3	**Breaking from convention:** Do for consistency, clarity, maintainability, or program functionality.	
G–4	**Initialisms:** Make name the same as for acronyms (see G–2).	`firstHtmlTag`
G–5	**Length:** Ideally is 12 to 24 characters for identifiers; 8 to 16 for program, module, and filenames; ≤60 total characters.	`FindDerivative` `ReportsModule.cpp`

G–6	**Mathematical and statistical values:** Make as close as possible to standard mathematical terms, symbols, and variables.	`X`
		`sigma`
		`PI`

G–7	**Names and meaning (semantics):** Use meaningful and descriptive names from the subject domain.	`FitRegressionModel()` `expectedValue`
G–8	**Space characters in names:** Do not use spaces in identifiers or file names.	`ReportFileA2` `TemporaryReportOutput`

Table 6–2. Summary of naming conventions for C++ identifiers.

["CPP" in the convention identifier means C++ code]

Convention Identifier	Convention	Examples
CPP–1	**Accessor functions:** Name with "`Get`" or "`Set`" as the first word.	`GetCountyArea()` `SetName()`

CPP–2	**Boolean constants:** Make name short and all letters are capitalized.	`TRUE`
		`FALSE`
		`ON`
		`OFF`

CPP–3	**Boolean values:** Make moderate length and easy to understand; name should indicate meaning of true or false value of variable assigned; make all letters uppercase; may include numerals; character breaks may be inserted.	`IS_AVAILABLE`
		`FILE_EXISTS`
		`NOT_FLOAT32`

CPP–4	**Class variables:** Suffix with "`_cv`".	`printerStatus_cv`

CPP–5	**Classes:** Use mixed case, first letter capitalized; with optional suffix "`_c`".	`Month`
		`AgentProfile`
		`AgentProfile_c`

CPP–6	**Constants:** Use all upper-case letters with underscores separating words.	`MAX_CELL_INDEX`

CPP–7	**Enumerators:** Make name the same as for constants (see CP–6).	`RED`
		`JANUARY`
		`COMMERCIAL_GROWTH`

Table 6–2. Summary of naming conventions for C++ identifiers.—Continued

["CPP" in the convention identifier means C++ code]

Convention Identifier	Convention	Examples

CPP–8	**Functions:** Use mixed-case with first letter capitalized; name should be a verb phrase or construction describing what the function does.	`FindMaximumTestScore()`
		`ComputeSecondDerivative()`
		`PrintSummary()`

CPP–9	**Mathematical and statistical values (other than constants):** Keep name simple by grouping within namespaces or classes.	
CPP–10	**Member functions:** Name these like functions (see CP–8) with optional suffix "`_cm`".	`FindAreaOfPolygon_cm()`
CPP–11	**Namespace aliases:** Use ≤6 lowercase alphanumeric letters.	`math1` `gisft`

CPP–12	**Namespaces:** Prefer ≤12 lowercase alphanumeric letters with underscores and optional suffix "`_ns`".	`math_functions_set_1`
		`gis_file_types`
		`statistics_ns`

CPP–13	**Objects:** Name these like variables (see below); generally include class name.	`monthFyBegin` `monthFyBegin_obj`
CPP–14	**Functions, procedures, and subroutines:** Make no naming distinction.	
CPP–15	**Pointers:** Begin name with "`ptr`" or suffix with "`_ptr`".	`interimCount_ptr` `ptrInterimCount`
CPP–16	**Types:** Use mixed case, first letter lowercase; suffix with "`_t`".	`int12_t smallRaster_t`
CPP–17	**Variables for a fixed number of entities:** Begin with "`num`" followed by the name with the first letter capitalized.	`numCounties` `numCells`
CPP–18	**Variables in general:** Use a descriptive noun or noun phrase naming the value represented by the variable.	`xCoordinate` `maximumEmployment`
CPP–19	**Variables in mixed case:** Use mixed case with the first letter lowercase; use underscores only when needed for clarity.	`countyBoundaryType` `currentAC_Amperes`
CPP–20	**Variables pluralized:** Use for an array, list, or a set of multiple values.	`productCodes[200]` `pixels[400][5000]`
CPP–21	**Variables in global scope:** Use "`::`" or suffix "`_g`".	`::category` `category_g`
CPP–22	**Variables for indexes and algebraic variables:** Generally use i, j, k, l, m, n, o, p as integer indexes or subscripts; generally use a, b, c, d, e, f, g and s, t, u, v, w, x, y, z as floating-point (real) variables.	

Table 6–2. Summary of naming conventions for C++ identifiers.—Continued

["CPP" in the convention identifier means C++ code]

Convention Identifier	Convention	Examples
CPP–23	**Variables and suffixes:** Optionally use suffixes.	

Suffix	Type	Examples
_c or _cls	class	Point_c or Point_cls
_count	the number of items or entities currently found in the data; how many of an item exist at a particular time	Counties_count
_f	float (single precision)	average_f
_fd	double (double precision)	average_fd
_ld	long double (long double precision)	derivativeOfX_ld
_g	global	year_g
_i	int	sampleSize_i
_il	long int	
_index	an index variable used as a subscript for an array	year_index
_cm	member of class	AdjustRetailPrice_cm()
_cv	class variable	retailPrice_cv
_ns	namespace	phase_I_ns
_idnum	the identifying number associated with a value or entity	County_idnum
_obj	object	employerRetail_obj
_ptr	pointer	NextAction_ptr
_str	string	changeName_str
_t	type (not built in)	myPartRecord_t
_tf	Boolean value	WIN_tf

Table 6–3. Summary of naming conventions for C identifiers.

["C" in the convention identifier means C code]

Convention Identifier	Convention	Examples
C–1	**C++ conventions:** Use conventions for C code, if applicable.	
C–2	**Struct instances:** Make like C++ objects (mixed case, first letter lowercase) with optional suffix "_struct".	imageProfileMain imageProfileMain_struct
C–3	**Struct type name:** Use mixed case with the first letter capitalized and optional suffix "_struct_t".	ImageProfile ImageProfile_struct_t
C–4	**Variables in global scope:** Use suffix "_g".	totalSumOfSquares_g

Table 6–4. Summary of naming conventions for filenames used within operating-system supported file systems.

["F" in the convention identifier is an abbreviation for "filename"]

Convention Identifier	Convention	Examples
F–1	**Aliases:** Do not use.	

F–2	**Characters used in filenames:** Use alphanumerics; may include the special characters "`.`", and "`_`"; start name with either an uppercase or lowercase letter.	`Region04_rasterA12.hsf.aux`
		`CatalogC`
		`UTIL_05b.gif`

F–3	**Directories and folders:** Use names following conventions for ordinary files; use ≤30 characters.	
F–4	**Length**: Use ≤48 characters, preferably between 8 and 16 characters.	
F–5	**Operating-system independence:** Filenames should be usable on any current mainstream operating system; should be usable on FAT32, NTFS, ext3, ISO 9660, and HFS+ file systems.	
F–6	**Similarity of filenames:** Use structurally similar names for similar files when possible.	

Manuscript approved May 9, 2017
For additional information regarding this publication, contact:
Director, Eastern Geographic Science Center
U.S. Geological Survey
12201 Sunrise Valley Drive, MS 521
Reston, VA 20192
(703) 648-4230
Or visit the Eastern Geographic Science Center at
https://egsc.usgs.gov/

Prepared by the USGS Science Publishing Network
Reston Publishing Service Center
Edited by David A. Shields
Layout by Cathy Knutson

ISSN 2328-7055 (online)
https://doi.org/10.3133/tm6F1