



**C++ coding conventions**  
(Compatible with code generated from "Skilltree" design)  
written by Avner Ben  
22-Mar-2005

**Contents**

1 .....	Contents
2 .....	Purpose
2 .....	Products
2 .....	Naming conventions
3 .....	Other guidelines
4 .....	C++ header file contents
6 .....	C++ implementation file contents

## Purpose

The purpose of this document is to define C++ code products – content and writing conventions - for use in the context of "Skill-Driven" design. *Note:* Skill-driven design specific items are annotated by "(SDD):".

## Products

C++ Code is designed and written in software components. Each component defines a namespace and is made of one or more software modules. Each module features one or more classes and related material and consists of at most two files : a header file (file extension ".h") and an implementation file (file extension ".cpp"). Both files share the same name before the extension. In addition, the software product may also contain source files (".cpp") that are not related to any module and feature the main routine of a process that uses the other files for the purpose of unit test or the production package.

## Disclaimer

The following guidelines apply to the common case. Exceptions are possible, at the designer's discretion, provided there is an obvious, self-explaining cause. Under no condition should the documentation dedicate space to explaining itself!

## Naming conventions

- Names that are composed of a number of words:
  - In **mixed case** names: embedded words are capitalized. All other letters are lower case (even for words that are originally all capitals, such as acronyms!)
  - In **upper case** names, embedded words are separated by underscore .
- **Types** (classes, structures, enumerations, typedefs, unions and generic parameters) are mixed-case, capitalized .
- **Variables** (local, global, arguments, members) are mixed case, non-capitalized .
  - Member data are suffixed by underscore.
- **Method** and function names are mixed case, non-capitalized .
  - Prefer polymorphic method names:
    - Method name should not be qualified by the entity which it serves. It is already in its namespace!
    - The names of methods that serve a particular argument type (as in the left side of double dispatch) should not be qualified by the argument type – it is already part of the method signature!
    - (SDD): Method names should consist of an imperative verb followed by the minimum needed to make it unique. The skill itself (that the method implements) is quoted above anyway!
- Preprocessor directives ("**macros**") are all upper case .
- **Enumerated** type **values** are all upper case .
- Arguments are (optionally) suffixed by **pass-mode** – Ptr or Ref.
- Arguments are (optionally) suffixed by **usage** - Out, InOut .

## Other guidelines

- **Generic parameters :**
  - Use contract-defining names (e.g. "*forward\_iterator*"), rather than the generic "T".
  - Use "*typename*" when any type (including basic types) is valid.
  - Use "*class*" when only user-defined types are valid.
- **Spacing:**
  - Separate method implementations by blank lines.
  - Separate classes by blank lines.
  - Separate semantic groups in a class (e.g. public methods, protected methods, member data) by blank lines.
  - After comma (as in argument list) always comes a space.
  - Operators are separated from their operands by spaces, except for the following cases:
    - The "=0" of pure virtual function declaration.
    - Default argument value.
- **Documentation :**
  - Avoid over-documentation. You must be able to keep every descriptive word of you write true to the current state of the code!
  - (*SDD*): In component and class, prefer the definitive skill over the short description. You describe what the entity *can do* for us, no more and no less!
  - (*SDD*): Use the "Stereotype" custom item to define the entity's role in the pattern, where applies (e.g. "Singleton", "Observer", "Façade", "Decorator", "Decorator subject").
  - (*SDD*): In method and skill, prefer usage contract rules ("Premise", "Input", "Output", "Cause", "New State" and "Exit") to custom item.
  - (*SDD*): Do not describe the algorithmic logic inside the function. A complete usage contract should eliminate the need!
- **Alignment:**
  - Right braces should be aligned on the column of the item to which they refer (rather than the left brace).
  - Left braces for long items (classes and long methods) should start on the next line, in the same column.
  - Left braces for short items (structures and methods less than 10 lines) may start on the same line.
  - Labels (e.g., "public:", "private:", "protected:", case in switch statement) are aligned on the same column as the item which they divide (i.e. column one for most classes). Do not indent labels!
  - Comments should be above their item and on the same column.
  - Comments for multiple items (e.g. "associations:") should be above the first item and dedented (i.e., on the same column as the containing item).
  - Continuation lines (for comments and long expressions) start on the same column. There is no need for a special mark! Avoid indenting continuation lines as they will be mistaken for control structures.
  - Break statements on the nearest operator (e.g. "&&", "||", "+"). The operator starts the continuation line.
  - Break statements and comments on the nearest punctuation (e.g. comma). The punctuation mark closes the broken line.

## C++ header file contents

1. **Software product** identification – title (domain name - between quotes).
2. **software component** identification
  - 2.1. **Title** (domain name - between quotes) followed by hyphen followed by programmatic name.
  - 2.2. **(SDD): Description:**
    - 2.2.1. The definitive skill (optional).
    - 2.2.2. Short description – "Function:" followed by text.
    - 2.2.3. Long description (optional) – "Description:" followed by text.
    - 2.2.4. Custom descriptor items (optional) – item keyword followed by colon followed by text.
  - 2.3. **Author.**
  - 2.4. **Copyright notice** and legal stuff (optional).
  - 2.5. Creation and revision **dates.**
3. Preprocessor directive ("macro") to prevent **double inclusion** – begin. Use a unique name made of the module name and "H".
4. **Include** preprocessor directives.
 

*Notes: (A) include only what you need. (B) Do not include files that are already included indirectly. (C) Do not use absolute pathname. (D) Use relative pathname only where necessary (for other components in the product). Still, prefer not to use pathnames at all. E.g., configure your IDE to look for include files in a series of predefined paths. Include directives are ordered as follows:*

  - 4.1. The standard C library.
  - 4.2. The standard C++ library.
  - 4.3. Infrastructure components.
  - 4.4. Other components in the software product.
  - 4.5. Other components in the project.
5. **Using namespace** commands.
 

*Note: Using the "std" namespace is not recommended. Prefer fully qualifying standard-library types.*
6. **Begin namespace** command. The namespace of the component or subsystem to which the module belongs.
7. **Global material** (optional):
  - 7.1. **Preprocessor directives** ("Macros").
  - 7.2. **Global constants.** *Note:* Prefer defining constants in the related class as static members.
  - 7.3. **Global variable** declarations ("extern"). *Note:* Use global variables sparingly. Prefer static members and singletons.
8. **Class** definitions
 

*Guidelines: (A) Order the class declarations by containment and inheritance dependency – i.e. bottom up (to avoid forward declarations). (B) The working implementations of an abstract base class should be on separate header files.*

  - 8.1. **Description (SDD):**
    - 8.1.1. **Title** (domain name - between quotes) followed by hyphen followed by either the definitive skill (if available) or the short description or "To Be Defined".
    - 8.1.2. **Short description** (if not already appearing above) – "Function:" followed by text.
    - 8.1.3. **Long description** (optional) – "Description:" followed by text.
    - 8.1.4. **Custom descriptor** items (optional) – item keyword followed by colon followed by text.
    - 8.1.5. **Stereotype** (e.g. "Singleton", "Facade" - optional).
  - 8.2. **Forward declarations** (for types used in the class). *Note:* Minimize the use of forward declarations! Besides representing unfinished design (let alone the case of recursive association in the problem domain), forward declarations also reduce performance, by inhibiting inline code!

- 8.3. **Left brace** (on a line of its own).
  - 8.4. **Internal types**, exported: typedefs, enumerations, inner classes and structures.  
*Notes:* (A) Prefer internal structures and classes to external ones when they are created exclusively by this class (e.g., iterator). (B) Prefer internal typedefs and enumerations over external ones when they are normally used to set the state of such objects. (C) Prefer inner *structure* over *class* when the entire contents are in the public.
  - 8.5. **Interface methods** (public).
    - 8.5.1. (*SDD*): Methods are documented by:
      - 8.5.1.1. The **skill** (that the method implements).
      - 8.5.1.2. **Short description** – "Function:" followed by text.
      - 8.5.1.3. **Usage contract** rules – rule keyword rules ("Premise", "Input", "Output", "Cause", "New State" and "Exit") followed by colon followed by text.
      - 8.5.1.4. **Custom descriptor** items (optional) – the item keyword followed by colon followed by text.
      - 8.5.1.5. The **facility** of which the skill is root (optional).
      - 8.5.1.6. *Note:* method declarations do not have to quote the same parameter names as in the implementation, or to name parameters at all. Use meaningful names to minimize the need for description above.
    - 8.5.2. *Order the methods as follows:*
      - 8.5.2.1. **Instantiation** (constructor, destructor, copy constructor, copy assignment).
      - 8.5.2.2. **Operations** (methods that do substantial work).
      - 8.5.2.3. **Accessors** (get/set methods).
      - 8.5.2.4. **Friend** function declarations (e.g. symmetrical binary operators) . *Note:* Friend functions may also be defined inline in the class.
  - 8.6. **Implementation methods** and data (protected). Methods and data to be accessible to derived classes.
  - 8.7. **Member data** (private).
    - 8.7.1. **Associations**. Containment of objects of a substantial user-defined type (rather than "basic types that C forgot", like "Complex", "Point" or "UnicodeString") . Descriptor contains (a) the role of contained object (optional), the cardinality on both sides (as in the class diagram) separated by colon (e.g. "0-1:1-N") and (c) containment semantics – either "by value" (black diamond in the class diagram) or "by reference".
    - 8.7.2. **State**. Variables of basic type and trivial user -defined types that are used by methods to control object state.
    - 8.7.3. **Cached data**. Variables of basic type and trivial user -defined types that are used by the constructor and other methods to store data for reference.
  - 8.8. **Implementation methods** (private). Methods that are used internally by other methods. "Primitives".
  - 8.9. **Prevented copy** (private). In classes with no copy semantics – declaration (only) of the copy constructor and/or the copy assignment.
  - 8.10. **Friend**. *Note:* the position of the friend declaration (private/public) is insignificant. The bottom of the class is where they are the least distracting.
  - 8.11. **Right brace** (on a line of its own).
  - 8.12. A class definition is (optionally) followed by **inline friend** and helper **function** definitions (e.g. symmetric binary operators and the stream insert operator) .
9. Definitions for **inline functions** only declared inside the classes above (optional). These are all grouped at the bottom of the header file
  10. **End namespace**. Append the namespace in comment to the right brace.
  11. Preprocessor directive ("macro") to prevent **double inclusion** – end. Append the macro name as comment to the "endif" directive.

## C++ implementation file contents

1. **Software product** identification – as in header file.
2. **Software component** identification (description optional – same as in header file).
3. **Include** directives:
  - 3.1. To the module's own header file.
  - 3.2. To header files used exclusively by the implementation (e.g., *use* of types have only been forward-declared in the header file of the module).
4. **Begin namespace** – same namespace as in the header file.
5. **Global variable** allocation, initialized. Extern variables declared in the module header file.
6. **class** implementations
  - 6.1. Separator line.
  - 6.2. **Class description** (as in the header file - optional).
  - 6.3. **Static member data** allocation, initialized (e.g. the singleton pointer).
  - 6.4. **Method** implementations (for methods not implemented inline in the header file).
    - 6.4.1. **Description** (optional - same as in header file).
    - 6.4.2. **Method header**, followed by left brace (usually in the same line).
    - 6.4.3. (*SDD*): Statements and blocks, each (optionally) preceded by the **reliance** implemented and – where in the one place implementing (rather than just relying on) a skill – the full **skill** description.
      - 6.4.3.1. (*SDD*): Control statements (while, for, if, case, do) require **reliance** descriptor above.
      - 6.4.3.2. (*SDD*): Reliance descriptors are not applicable to the switch statement.
  - 6.5. **Friend function** implementations (for friend functions not implemented inline in the header file).
  - 6.6. **Inner class** implementations (Methods etc. not implemented inline in the header file).
7. **End namespace**. Append the namespace in comment to the right brace.