Design and Implementation Conventions
in the C+- Subset of C++
6th DRAFT
28 January 2011

1. The following C features are not to be used, except as necessary to interface with existing libraries or other open-source code:
   a. multiple inheritance
   b. private inheritance
   c. private or protected derivation
   d. friend, except in testing classes
   e. static allocation of non-atomic data (more below)
   f. lexically nested classes
   g. virtual base classes
   h. pure virtual functions
   i. conversion by constructor
   j. ´&' as an expression operator (OK for call-by-ref parameter), except in lowest-level memory-management classes
   k. anonymous union or enum
   l. top-level variable decls (i.e., var decls outside of a class)
   m. struct
   n. register
   o. goto
   p. auto
   q. volatile
   r. extern, except in ‘‘extern "C"’’
   s. malloc
   t. bit fields

2. Define each class in one .h/.C file pair. The only exception to this rule is for homogeneous collection or container classes. For such classes, a single .h/.C pair may (but is not required to) contain the definitions for both the container class as well as the elements that it contains. For example, a database class and the type of its elements may be defined in a single .h/.C pair. If the element type of the container has data members that are defined as other classes, all of those other class must be defined in separate .h/.C file pairs.

3. Format and document .h files *exactly* as follows, with strict adherence to the format of the comments, including the number and placement of *'s and the amount of indentation.

```
////
//
// The comment at the top of a .h file is a high-level description of the class
// defined in the file.  Start the description with the words "Class XXX" and
// then describe the purpose of the class and the major functions it provides.
```

```
// The function descriptions in this header comment are generally brief, e.g.,
// "Class XXX provides functions to add, delete, change, and find its
// elements."  Do not list all of the function details in the header comment,
// since full comments for each function appear below in the body of the class,
// at the cite of each function declaration.  The header comment can describe
// the data representation used in the class in high-level terms if it's
// germane to explaining what the class is for.  The header comment does not
// describe low-level details of the data representation or any details of
// member function implementation.
//
//
// Author: Full name and current email address of file's author; name is at
//         least first and last, with middle name or initial if necessary or
//         commonly used by author; email address appears in parentheses
//         following full name; email address may be abbreviated to a local
//         address if the full address can be expected to be known to.  E.g.,
//         Gene Fisher (gfisher@calpoly.edu); John H. Smith (john_smith@...).
//
// Create Date: Date file was originally created (NOT last modified), in the
//              format ddmmmyy, where dd is a one or two-digit month date, mmm
//              is the three-character all lowercase abbreviation for the
//              month, consisting of the first three characters of the month
//              name, yy is a two-digit year.  E.g., 31jan98.
//
////

#ifndef xxxIncluded
#define xxxIncluded

#include ...


class X : public Y {

  public:

    T1 FuncName(T2 t2, ..., Tn tn);
    //
    // Prose description of the function, describing what the function does,
    // not how it is implemented.  Describe the use of each parameter by name,
    // refer to the instance object as "this", and use the word "return" to
    // describe the return value if there is one.  Also describe each data
    // member used as an input and each data member modified as an output.
    // Modification to a parameter or data member include indirect modification
    // to the value of reference or pointer-valued parameter or data member.
    // Stylistically, use complete sentences, avoid passive voice.
    //
    // pre: formal precondition
    //
    // post: formal postcondition
    //
    // catch: list of exception objects caught
    //
```

```
    // throw: list of objects thrown
    //

     ... other public functions ...

    Note no public data members


  protected:

    Protected functions in same format as public functions.

    T1 varname;                    // Comment describing data member
     ... other protected data members ...
};
```

4. Format and document .C files as follows.

```
////
//
// Implementation of XXX.
//
////

T1 XXX::FuncName(2 t2, ..., Tn tn) {

    Note that function signature is identical to that declared in .h file,
    including spellings of parameter names

    //
    // Include comments for each local variable and comments above each line or
    // group of lines that describe how the function works.  All but the
    // "totally obvious" lines of code should be commented.  Comments for
    // variables should be descriptive noun phrases.   Comments for code lines
    // should be in complete sentences.
    //
    // All code comments should be formatted exactly as as this one is: (1)
    // start with "//" on a separate line, indented to the current level of
    // code indentation; (2) start each comment line with "//", indented to
    // current indentation + 1; (3) end with "//" on a separate line, indented
    // to current indentation + 1.
    //
}
```

   See below under "indentation and spacing conventions" for further discussion of the format of code within a function definition.

5. On static allocation
   a. The data declaration of any user-defined class type should always be a pointer to that type.
   b. Rationale: we are using a SmallTalk-like or Java-like approach to object allocation.

6. Types aliased with `typedef`

a. The `typedef` keyword should be used to provide mnemonic names for atomic types. E.g.,

```
typedef int Coordinate
```

b. When `typedef` is used to provide a mnemonic name for a non-atomic type, a pointer should not appear within the *typedef*. E.g.,

```
typedef String Name
```

is conventionally correct

```
typedef String* Name
```

is not correct.

c. The preceding `typedef` rule, together with the earlier rule for pointer-only allocation of class types, ensure referential transparency for type atomnicity. I.e., whether a type is atomic or non-atomic can always be determined at the cite of type reference, since:
   i. Non-atomic types are always referenced as pointers, with '*'
   ii. Atomic types are are always referenced without '*'

7. Encapsulating non-conventional external libraries
   a. Resources provided by any external C++ library that violates any of the preceding conventions must be encapsulated in *exactly one* class.
   b. That is, a "wrapper" must be placed around all non-conventional library services.

8. Indentation and spacing conventions
   a. Except for the specific indentation shown within a class definition above, indentation should be every 4 spaces.
   b. If tabs are used, assume tab width = 8 characters, so that spacing is physically 4-spaces-then-tab, ...
   c. Do not set tabwidth to any value other than 8 characters in any editor.
   d. The following is a template for indentation and spacing of C++ functions; blank lines are significant.

```
T1* FuncName(T2* t2, ..., Tn* tn) {
    Tv1* tv1;           // Comment ...
    ...
    Tvm* tvm;           // Comment ...

    //
    / Comment ... .
    //
    for (...) {

        //
        // Comment ... .
        //
        if (...) {
            ...
        }
        //
        // Comment, if necessary.
        //
```

```
    else if {
        ...
    }
    ...
    //
    // Comment, if necessary.
    //
    else {
        ...
    }
}

//
// Comment ... .
//
while (...) {
    ...
}

//
// Comment ... .
//
for (start-expression; while-expression; end-expression) {
    ...
}

//
// Comment ... .
//
for (long-start-expression ... ;
        long-while-expression ... ;
            long-end-expression) {
    ...
}


//
// Comment ... .
//
switch (...) {
    //
    // Comment, if necessary.
    //
    case c1:
        ...
        break;
    ...
    //
    // Comment, if necessary.
    //
    case ck:
        ...
        break;
```

```
}
```

9. Class and function naming conventions
   a. The names of MVP model types (class and typedef names) should be identical to the names of corresponding UML objects. If the UML object name does not obey the capitalization conventions for type names given below, then the UML name should be changed accordingly.
   b. The names of model functions should be identical to the names of corresponding UML operations. If the UML operation name does not obey the capitalization rules for function names given below, then the UML name should be changed accordingly.
   c. The names of view types (class and typedef names) should be the same as corresponding model names, with the suffix "UI" added. If a single design defines two or more alternative views, the view class names should be disambiguated with the type of UI for each. E.g., for model class PersonDatabase, view classes PersonDatabaseButtonUI and PersonDatabaseMenuUI are button-style and menu-style UI's, respectively.
   d. The names of view functions should be full-word mnemonic, including multi-word where appropriate, with the same capitalization conventions as for model type and function names. See "capitalization conventions" below for further details.
   e. The names of process types and functions should be full-word mnemonic, including multi-word where appropriate, with the same capitalization conventions as for model type and function names. See "capitalization conventions" below for further details.
   f. The names of constants should be mnemonic, typically one word or two words, all uppercase, with multiple words separated by underscores. E.g., LENGTH, MAX_SIZE.
   g. The names of data member access functions should start with "Get".
   h. The names of data member setting functions should start with "Set".
   i. The names of boolean-valued query functions should start with "Is".
   j. The names of searching functions should start with "Find".

10. Capitalization conventions
   a. Type names (i.e., class and typedef names) should begin with a capital letter, and each distinct word in the name should be capitalized, e.g., `PersonRecord`. If an abbreviation is used in the name, all letters in the abbreviation should be capital, e.g. `PersonDB`.
   b. Function names should follow the same conventions as type names.
   c. Filenames for .h/.C pairs should be all lowercase, and the same as the class name, with words separated by dashes. Where file names would exceed 20 characters (or some other platform-specific filename limitation), they may be abbreviated, but should be an understandable mnemonic abbreviation. For example, for a class name `GeneralArtifactRepository`, `gen-art-repository.h` is a reasonable abbreviation, `gar.h` is not. In all cases, the root name of .h and .C files should be identical.

11. Variable naming conventions (including function parameters and class data members).
   a. For variable names of user-defined types:
      i. Variable names (including parameter and data member names) should be all lower case and a "proper" abbreviation of the type name, e.g., PersonRecord* person_rec, PersonDB* person_db.
      ii. When a variable name abbreviates two or more words, the words should be separated by

underscore characters

iii. A "proper" abbreviation is an abbreviation the characters of which are a proper subset of the characters in the words of the variable's type. E.g., PersonDB* person_database is not a proper abbreviation.

iv. In all cases, variable names should be as short as possible while still being mnemonically significant.

v. This rule is one of the few places in these standards open to subjective interpretation. E.g., in a context where few other variables start with the letter "p", the name "p_rec" could be sufficiently mnemonically significant to be used instead of the longer "person_rec".

vi. In all cases, variable naming should be consistent throughout a project. E.g., PersonRecord *p_rec, PersonDB* person_db is not consistent naming.

vii. In the case where two or more variables of the same type are declared in a given scope, the names should be suffixed with unique integers starting with 1, e.g., `PersonRecord* person_rec_1, *person_rec_2` or suffixed with short mnemonic identifiers, e.g., StringEditor* str_ed_name, *str_ed_id, *str_ed_addr.

viii. For variables with two or more words, numeric suffixes should be separated with an underscore character, e.g., PersonDB* person_db_1, *person_db_2. For variable names of one word, numeric suffixes should be appended directly with no underscore, e.g., Name* name1, *name2;

b. For variable names of atomic types or external library types that follow some other type naming conventions:

i. Variable names (including parameter and data member names) should be all lower case, short, and mnemonic, e.g.,

```
bool status
int counter
```

ii. Where very short names are sufficiently mnemonic, such as with loop counter variables, single-character variables names are allowed, e.g., int i; for (i=1, ...) ... ;

iii. The same numeric suffix and disambiguation rules above should be used.

12. Size limits

a. No function may be longer than 25 lines, excluding comments and blank lines. This rule may not be achieved by condensing lines in such a way as to violate any of the preceding formatting conventions. E.g., the following is legal

```
for (i=1; i<=n; i++) {
    if (i % 2) {
        Foo1(i);
        Foo2(i);
    }
}
```

the following is NOT legal

```
for (i=1; i<=n; i++) { if (i % 2) { Foo1(i); Foo2(i); } }
```

nor is even the following

```
for (i=1; i<=n; i++) {
    if (i % 2) {
        Foo1(i);
        Foo2(i);}
}
```

b. No class may have more than 25 public functions plus 25 protected functions, i.e., no more than 50 functions total. If a class has fewer that 25 public functions, it may have up to 50 functions total, however it may never have more than 25 public functions. Typically, classes should have far fewer than 50 functions total.

c. No class may have more than 25 protected data members.

d. Note that combining the 25-line rule with the 50-function rule means that no .C file can be longer the 1250 lines of code. Typically, .C files should have far fewer than 1250 lines.