Physical Design Rules

1 Introduction

These rules are concerned with the organization of code into files. They define standards to avoid name collisions, and ensure that compile and link-time dependencies between files are correctly established.

The book *Large Scale C++ Software Design* by John Lakos gives more details for these rules and describes techniques to minimize the complexity of dependencies between files in order to make linking, testing, and maintenance less costly. John Lakos has also condensed this book into three articles in *C++ Report* in the June, September, and November-December 1996 issues.

Before applying any specific organizational rules, some thought should be given to overall organization of components, files, and libraries. The issue of libraries and link modules is of particular importance. The organization into files and libraries satisfy the following goals:

1. Grouping into libraries should be arranged with linkage dependencies in mind, such that *you never need to include the same library twice* to fully link the program. For example, if library `L.a` contains classes `c1()` and `c3()`, and library `M.a` contains class `c2()`, then if a member function of `c1()` calls a member fucntion of `c2()`, which in turn uses anything in calls `c3()`, you have a cyclic linkage dependancy. On many systems the executable would have to be made via an ugly statement like:

   ```
   CC -oexecut mymain.c -lL.a -lM.a -lL.a
   ```

   Avoid this situation; if two groups of routines necessarily depend upon each other, that grouping is not a good one for separating them into two libraries.

2. An individual file should contain a group of related routines (classes and functions)— related in the sense that if a program needs one of them, it normally needs most or all of them. Many compilers threat one

file as one compilation unit; that is, they pull in the code for everything in that file if anything in the file is used.

Some style guides promulgate a rule of "one class in one file." That is too inflexible (see Rule 12 below, for example) and proliferation of small individual files can itself be a headache. But you should *consider* the one-class/one-file ideal when planning your file organization.

2 Rules

1. Place a unique and predictable internal include guard around the contents of each header file.

   - Style suggestion: For Trybos_AbstractRecord.hh use:

     ```
     #ifndef _Trybos_AbstractRecord_hh_
     #define _Trybos_AbstractRecord_hh_
     // All of Trybos_AbstractRecord.hh here
     #endif
     ```

2. The following rule may be useful if preprocessor efficiency is a concern: Place a redundant external include guard around each preprocessor include directive in each header file.

   - This will stop the preprocessor from even opening a file if the file has already been included.
   - Example:

     ```
     #ifndef  _Trybos_Package_hh_
     #include "Trybos_Package.hh"
     #endif
     ```

3. The .cc file of every component should include the corresponding .hh file as the first substantive line of code.

- Ensures that the .hh file parses by itself, and another user would not have to figure out what other header files to include before the .hh file (the .hh file should include them itself).

4. Prepend every global identifier with its package prefix.

   - For example: `class Cal_Tower;`
   - This helps avoid name collisions.
   - This may be a good idea even when namespaces are available, you may want packages to have finer granularity than namespaces.

5. Avoid data with external linkage at file scope.

   - If the statement `int i;` occurs outside any blocks in a file, it is at file scope, and is therefore "global data," and can conflict with names elsewhere in the program.
   - Global data tends to reduce modularity.
   - The reduced "locality of reference" from the use of global data reduces compiler optimization.
   - If global data is nonetheless needed, it should be implemented as private static data in a class with static accessor functions.

6. For similar reasons as for rule 5, also avoid (global) functions, enumerations, and typedefs at file scope. In general allow only classes, structs, unions, and definitions for overloaded operators at file scope.

7. If an entity is defined in a .cc file, it should be declared in the corresponding .hh file.

   - For example, if the following appears in a .cc file:

```
void f(int x, int y){
  // function body here
}
```

then the following should appear in the corresponding .hh file:
`void f(int x, int y);`
(A point of style: C++ permits the names for arguments—x and y in this case—in the declaration to differ from the names of the corresponding arguments in the definition. It is better style, however, to use the same names in both cases, and to try to make those names meaningful.)

- Following this rule makes it possible for compile dependencies to be obtained from the list of included header files.

8. Obtain the declaration for a function by including the .hh file of the component in which the function is defined.

    - If the argument list of a function changes, this needs to trigger a compilation of all components that use the function. The compilation dependencies are built from the #include statements.

9. When using a class, declare the class locally and only include the header file for that class if the compiler requires it.

    - The statement `class Cal_Tower;` is an example of a local class declaration.
    - Often, the compiler does not need to know the details of a particular class, and including the header file would introduce an unnecessary compile dependency between components.
    - Trying the compilation without the header file is the only reliable way to find out.
    - Examples of when the header file is needed:
      A class derives from another class, or contains an instance of another class as a data member.
    - Example of when a local class declaration is adequate:
      A member function returns an instance of that class.

10. Put inline functions into a separate .icc file which is then #included into the .hh file.

11. Large components can be broken up into many .cc files to improve compiler and linker efficiency. (On many systems, code residing in a library originating in one file is linked in on an "all-or-none" basis. So a single large file containing unrelated routines can cause code bloat.)

12. Put classes with their friend classes and functions in the same component.

    - Prevents other components from violating encapsulation by defining local versions of friends.
    - Friends are useful for iterators and global operator functions.
    - In general, minimize friend declarations, for maintenance reasons.

13. Avoid cyclic dependencies between components.
    Especially avoid cyclic dependencies between packages.

    - If two classes are interdependent, either break the dependency or put them in the same component.

3 Definitions

1. File scope: The scope outside of all blocks and classes in a file.

2. Translation unit: A .cc file after all .hh files have been included by the preprocessor.

3. Internal linkage: A name local to its translation unit has "internal linkage," and cannot collide with names in other translation units.

4. External linkage: A name with "external linkage" can interact with other translation units at link time.

5. Global identifier: A name with external linkage or a name at file scope in a .hh file. (Potentially useable anywhere in the program).

6. Declaration: Introduces a name into a program. Provides a description of the named entity, but does not create it. Does not result in any memory allocation. A declaration may be repeated many times. For

example, the following are declarations:

```
class Point;
double f(int i, int j);
```

7. Definition: Provides a unique description of an entity. May result in memory allocation. There must be exactly one definition of an entity. For example:
   - A function body:
     ```
     float recip(float x) {
       return 1/x;        }
     ```
   - A class body:
     ```
     class Ruler {
       // private and public variables
       // and methods declared or defined
       // here inside the class body   };
     ```
   - Instantiating variables or objects:
     ```
     int x;
     Ruler yardstick;
     ```
8. A definition is also a declaration unless:
   It defines a static class data member or
   It defines a member function of a class, and is not inside the body of code that defines that class.
9. A declaration is also a definition unless:
   It declares a function without specifying its body or
   It specifies "extern" and has no initializer or function body or
   It declares a static class data member within a class definition or
   It is a class name declaration or
   It is a typedef declaration.
10. Component: A .hh file together with the corresponding .cc files.

11. Package: A collection of components organized as a physically cohesive unit. (eg: the "muon" package would have all the code for muon reconstruction.)