BCLFormatting

Added by csbsyadmin, last edited by Mendenhall, Jeffrey L on Aug 10, 2012

Contents

Naming, Casing

- Variable Names should be **meaningful**, even if it means making them longer. Generic names like `data` for a variable, should not be used as the do not tell anything about the kind of information it contains.
  - **Global** variables of namespaces are indicated by `g_` and start with one capital letter, e.g. `g_VariableName` ;
  - **Member** variables of classes are indicated by `m_` and start with one capital letter, e.g. `m_VariableName` ;
  - **Static** variables in **classes** are indicated by `s_` and start with one capital letter, e.g. `s_VariableName` ;
  - **Static** variables in **functions** are indicated by `s_` and start with one lowercase letter, words are separated by underscores, e.g. `s_variable_name` ;
  - **Enumerator items** should be preceded by `e_` , e.g. `e_Alanine` ;
  - Variables passed as **function arguments** are named in capital letters, e.g. `VARIABLE_NAME` ;
  - **Local** variables are named in small letters, e.g. `variable_name` ;
  - In general, **Variable names** must have an underscore, or start with a lower case letter, or be ALL_CAPS
- **Template parameter names** should be preceded by `t_` and start with one capital letter, e.g. `t_DataType` ;
- **Templated class and function names** must contain both upper and lower case letters and not have any _'s in them
  - **What happens if this rule is broken** it is possible that the spacing around the template class name will be incorrect after the autoformatter is run

- **Namespace names** should be short (<= 9 letters) and all lowercase;
- **Filenames** should match the class name that the file contains. A file containing the class `ClassName` should be named in the following way: `namespace1_namespace2_class_name.ext` ;

- **No numbers** should be used in variable names, instead use spelled out numbers or letters, e.g. `variable_a` , `variable_b` or `variable_one` , `variable_two` ;

- **Abbreviations** used are:
  - AA for amino acid,
  - `SiPtr` for SimplePointer,
  - `ShPtr` for SharedPointer;

Indentation

Spaces and Tabs

- Spaces
  - **No spaces** before or after the -> operator or =.= operator
  - See below for spaces before and after brackets
- **Tabs** should be **two spaces** and should consist of two space characters rather than the tab character

Indentation and Line Breaks

- General Indentation is **two spaces**
- In **derived classes**, the base class should be on the next line and idented
- In **template classes**, the class name should be on the line following the template keyword and should not be indented
- `public` , `private` and `protected` keywords should **not** be indented
- One **empty line** at the **end of the file**
- There should be **no more than one line** of whitespace anywhere in the file

- There should be a **maximum of 120 characters** on a line, longer lines have to use line breaks (most editors can display a line at 120 characters to help you with this; a guide for Eclipse and VS is on the wiki)

```
.
      virtual storage::Pair< Alignment< t_Member>, double> Align
      (
        const Alignment< t_Member> &ALIGNMENT_A,
        const Alignment< t_Member> &ALIGNMENT_B
      ) const = 0;
```

- Breaking **strings** >120 characters, indent so that it does not appear as another parameter

```
.
```

```
      BCL_Message
      (
        util::Message::e_Standard,
        s_GapPenaltiesDescription[ i] + " penalty :" +
          util::Format()( m_Gap_penalty_params( i)-
>GetNumericalValue< double>())
      );
```

- **Loops** longer >120 characters

```
.
      for
      (
        typename util::ShPtrVector< t_Member>::const_iterator itr(
SEQUENCE.Begin()), itr_end( SEQUENCE.End());
        itr != itr_end;
        ++itr
      )
      {
         // code
      }
```

- **if** statements longer >120 characters

```
.
      if
      (
        math::sqr( m_CellWidth.X() * delta_x) + math::sqr(
m_CellWidth.Y() * delta_y)
        + math::sqr( m_CellWidth.Z() * delta_z) < radius2
      )
      {
        within_radius = true;
      }
```

Brackets

- **No space before** and **one space after** opening and closing brackets, **unless** the bracket encloses nothing, e.g. `std::string application_name( bcl::app::GetApps().Begin()->GetName());`
- These rules apply for all types of brackets: `(` , `)` , `<`, `>` , `[` , `]` , e.g. `template< typename t_AminoAcid>`
- A space has to be added between multiple closing >, e.g. `storage::Vector< util::ShPtrVector< biol::AABase> > sequence` to distinguish it from the stream extraction operator >>
- `}` , `{` are not covered by those rules

Curly brackets

- curly brackets } , { are used for scopes. They are always on an empty line all by themselves,
  - **exception**: when initializing arrays, it is allowed to keep all members on the same line, if and only if the line is still < 120 characters
- **correct:**

```
.
      void function()
      {
        ..
      }
      or
      if( true)
      {
        ..
      }
      const double x[] = { 5.0, 10.0 };
```

- **wrong**

```
void function(){}
      or
      void function()
      {}
      or
      if( true) {
      ..
      }
      or
      if( true) { return;}
      or
      if( true)
      {
        return;}
```

Other

- General
  - Use the **American date format** delimited by slashes (mm/dd/yyyy).
  - For any **normal user output** use the macro `BCL_Message( MessageLevel, Description)` defined in `bcl_message.h` and pass the appropriate `MessageLevel` (=SILENT, CRITICAL, STANDARD= , or =VERBOSE=) and a description.
  - For **errors** use `BCL_Assert( Condition, Description)` defined in `bcl_message.h` . The program will terminate if the condition is false. Pass an appropriate description.
  - To get an undefined value for a specific type, e.g. double, use `util::GetUndefined< double>`

```
();
```

- To **check** whether a is defined or undefined use `IsDefined( a);`

- **Always compile after running the FixObviousBclGuidelinesViolations script!**
  - The autoformatter currently removes namespace scope resolution (e.g. in the math namespace, all math:: will be removed)
  - In rare cases this may cause problems, e.g. if math::Vector::Absolute() calls math::Absolute, the scope resolution is, in fact, necessary
  - Recompilation / rerunning the examples will identify problems of this nature; if you encounter them, please consider whether the class really needs the same function. If it does, please inline the function in the classes code

- Namespaces
  - Avoid `use namespace std` instead **indicate non-bcl namespace** for every element, e.g. `std::` for std namespace elements.
  - Maximum depth of 2 namespaces.

- Classes
  - **All classes** should be **derived** from `util::ObjectInterface` . Every object should have a default **Read() and Write()** function that should read and write the object's data. These functions are to be used to overload the << and >> operators. All read and write functions work on `std::istream` and `std::ostream` to keep flexibility to use other streams (e.g. zipped streams, std::cin, std::cout and filestreams). Read() and Write() functions should be **protected**. More detailed information about the library design is on the BCLLibraryDesign wiki page.
  - All classes should be **declared in a forward class declaration** in a .fwd.hh file

```
#ifndef BCL_BIOL_AA_DATA_FWD_HH_
         #define BCL_BIOL_AA_DATA_FWD_HH_

         namespace bcl
         {
           namespace biol
           {

             // Forward declaration of AAData
             class AAData;

           } // namespace biol
         } // namespace bcl

         #endif // BCL_BIOL_AA_DATA_FWD_HH_
```

- Functions
  - **Avoid function-pointers**, instead derive a wrapper class from the `bcl::math::Function` class!
  - **Avoid non-constant reference arguments**, use pointer arguments or return values instead!
  - Write **only necessary** functions, e.g. write an explicit copy-constructor and assignment-operator only if a deep copy is needed, but always write both.
  - Do **not use virtual** if nothing is derived from this class. (The keyword "virtual" is used, to enable overwriting of the function in derived classes. Once virtual in the base class, every function in a derived class will overwrite a function with the same signature.)

```
#include <iostream>

          class BaseA
          {

          public :

            virtual void Print() const
            {
              std::cout << "BaseA" << std::endl;
            }

          }; // class BaseA

          class BaseB :
            public BaseA
          {

          public :

            void Print() const
            {
              std::cout << "BaseB" << std::endl;
            }

          }; // class BaseB

          int main ()
          {
            std::cout << "Hello, world!\n";

            BaseA baseA;
            BaseB baseB;

            baseA.Print();
            baseB.Print();

            BaseA* ptrAA( &baseA);
            BaseA* ptrAB( &baseB);
            BaseB* ptrBB( &baseB);

            std::cout << "AA: ";
            ptrAA->Print();
            std::cout << "AB: ";
            ptrAB->Print();
            std::cout << "BB: ";
            ptrBB->Print();

            return 0;
          }

          Hello, world!
          BaseA
          BaseB
          AA: BaseA
          AB: BaseB
          BB: BaseB
```

- Typedefs
  - Only use =typedef=s for external uses of a class
  - Avoid names similar to classes; the Naming rules apply

- C++
  - Use **new style casts** (`const_cast` , `static_cast` and `dynamic_cast`).
  - Use **const** as much as possible to indicate constant arguments, pointers, or member functions which do not alter the object's members! (The keyword `mutable` in front of a member variable allows changes although `const` is used.)

Commenting

General Comments

- Make comments **meaningful** and explain conceptually what is going on and why, rather than stating the obvious, i.e. don't say: *a for loop*, be more specific like *looping over all amino acids to examine the coordinates of each amino acid and store them in an array entitled X*.
- Within functions, separate blocks logically and add a **comment before each logical block**.
- Leave a **space after //** before the text begins, i.e. `// comment is here`
- Use **block comments in classes**, both in `.h` and `.cpp` (consistently between the two): data, data access, construction and destruction, get and set, operators, operations, input and output
- Comment the **closing bracket for blocks** larger than 10 lines, e.g. for namespaces close with `} // namespace bcl`

Doxygen conform Comments

- **Each** variable (local, member, or global), each function, and each class must have a comment conform to the **doxygen** style
- Required **variable** comments: one line for the definition of the member, unless there is room on the same line, like:

```
//! member variable comment
    double m_LongNamedMemberVariableWithoutSpaceLeftForDocumentation;

    m_ShortMemberVariable //!< detailed description
```

- Required **class** comments (block style)
  - `@brief`
  - `@author`
  - `@example`
  - `@date`
- Required **function** comments:
  - `@brief` (should be written for function headers; the full comment should be with the implementation)
  - `@param`

  - `@return` (unless return type is void)
  - `@see` if your function calls another function
- No `@return` in constructor comments

```
//! @brief construct FlagStatic from NAME, DESCRIPTION
    //! @param NAME - the name of the flag as a string
    //! @param DESCRIPTION - a string description of the flag
    //! @see FlagBase( NAME, DESCRIPTION)
    FlagStatic( const std::string &NAME, const std::string
&DESCRIPTION)
    {
    }
```

File Organization

- **Header** (.h) and **forward header** (.fwd.hh) files should be stored in the **include** directory, **source** files (.cpp) should be stored in the **source** directory.
- All files should be stored in the subfolders corresponding to their namespace (e.g. `bcl_contact_map.h` should be stored in `bcl/include/contact/bcl_contact_map.h`).

- One header and one forward header for each **namespace**.
- **Each header file** may contain a maximum of **one class** (not including helper classes only used for the main class).
- All class **forward declarations** should go in forward headers, **declarations** should go in headers, and **implementation** should go in source files except:
  - **Template** classes implementation must go into the header; Put larger functions that would usually go into the source code file (.cpp) to the end (so that they can easily be transferred later).
    This is currently done since the "export" keyword is not supported on template functions in MVC. In turn it is impossible to put template class functions into separate source code files: The linker will not recognize the function specified for a particular template class (error 2019). Given this fact the linker also reports multiple definitions of in non-template classes (error 2005) for template functions of included template classes if header and source code were separated!
  - **Small functions** (2-3 lines) should be written in the header.
- Each class `"bcl_NAMESPACE_CLASS.h"` must have an **example** `"example_NAMESPACE_CLASS.cpp"` demonstrating the usage of the class.
- A header and forward header should have **preprocessor tags** that check if they are defined and if not, define them.

Includes

- **Order** of includes (use code templates):
  - Forward header of the class
  - Other forward headers (use forward headers instead of bcl headers whenever possible)
  - BCL headers; always include namespace header
  - Third party libraries (ex: MySQL++) and standard headers (use <iostream> rather than "iostream")
- The **header to a .cpp** should be on the top and separated by an empty line

- Include the **.fwd.hh only** if you just need the type **declaration** for a class (no member function calls) and include the actual header in your cpp (where you implement the methods and need the methods of this type)
- Headers should be in **alphabetical order** within their subcategories and one empty line between subcategories

Examples, Test Cases

How to write Examples is explained on the specific page in detail.

-- Main.woetzen - 21 Jan 2009

Labels: None

Add Comment