Some C++ Coding Standards

Gerald Pfeifer

pfeifer@dbai.tuwien.ac.at

http://www.dbai.tuwien.ac.at/~pfeifer/

March 28, 2004

Abstract

This is a *very* preliminary draft of a set of coding guidlines for C++. While most abstract guidlines in this paper are hardly debatable, I am fully aware of the fact that some of the accompanying more specific ones are highly subjective and will try to separate these in the future. Thanks to Robert Bihlmeyer <`robbe@orcus.priv.at`> and Mads Martin Joergensen <`mmj@suse.de`> for providing valuable feedback.

1. Use `const` liberally

   For one, this helps the compiler to automatically detect some coding errors. Secondly, it may allow for the generation of more efficient code in some cases. And finally, labelling variables, respectively member functions, as `const` also serves as a kind a documentation.

   Especially const member functions (i.e. member functions of a class that do not modify `this` class upon invocation) should be clearly marked as `const`.

2. Refrain from using type casts

   Actually it is very rarely necessary to employ type casts. Whenever you *have* to use a cast, either use the cast operators (`const_cast`, `dynamic_cast`, `static_cast`, `reinterpret_cast`) or flag the cast by a comment containing the word `cast`. (It is thus strongly recommended against using the character sequence "cast" anywhere else in the source.)

   One notable exception here is when you want to treat an object of a derived type as an object of one of its ancestor types as in the following example.

```
class STUDENT : PERSON
    {
    unsigned id;
    ⋮
    };

ostream& operator<< (ostream& out, const STUDENT& student)
    {
    return out << (PERSON&)student << '(' << id << ')';
    }
```

3. Use intendation very carefully
   In theory, a compiler honoring intendation should be able to unambiguously analyse any well-written program that lacks all opening and closing curly braces, as illustrated by the following example:

   ```
   while( !  b )
       b=Compute(...);
   b=Compute(...);
   if( b )
       if( c )
           return false;
       else
           ⋮
   ```

   Each level of intendation should be indicated by a number of spaces that is an integer multiple of a given number m. Using different values for m in one project or even a single source file is a "very bad thing" ^{TM}.

   From my experience, $m = 4$ seems to be a good choice.

4. Consider keeping the names of local variables rather short
   Since the semantics of local variables should be rather clear, there is no need to give them long, overly descriptive names. Even a single-letter name should be sufficient for most loop-variables, for example.

5. Naming/1: Avoid leading underscores
   You should never use names starting with an underscore for types, variables, functions, constants or macros, as (most of) these names have been reserved for the compiler's internal use since ANSI C.

(Names starting with an underscore followed by a lowercase letter are not reserved by the standard, but might cause troubles with some non-strictly conforming compilers.)

6. Naming/2: Avoid names ending with `_t`

   In addition to those types ending with `_t` that are defined by ANSI, many OS vendors seem to use this "namespace" for system specific types which can lead to name clashes.

7. Naming/3: The name of a member function should not include the name of its class

8. Don't access data members directly

   Whenever you want to get, respectively set, some value or attribute of an object, do not do this by accessing data members of that class directly but employ member functions.
   Of course, the implementation of member functions of that very class represent an exception here, though it is often also a good idea to implement "higher-level" member functions by using those "low-level" functions instead of accessing data members directly.

9. Consistent naming/3: `get...`, `set...`, `is...`

   The member functions of a class that are used obtain, respectively set, some attribute or value of an object of that class should follow some unique naming scheme. Also member functions that check for some boolean property of an object should follow a similar scheme.

   I suggest to prefix each of these classes of functions with one of `get...`, `set...` respectively `is...`.

10. Use specific types; consider using `unsigned` types

    The C++ standard provides several types like `size_t`, `offset_t` or `time_t` (to represent memory sizes, pointer offsets, or dates/times). Use these instead of defining your own.

    If, for some variable, you can choose between several types, take the one whose range contains as few meaningless values as possible in the context of that variable.
    Specifically, consider using `unsigned`, `unsigned long`,..., instead of `int`, `long`,... wherever appropriate, but take great care *not* to use unary minus (`-`) or arithmetics also involving signed integers in that case. And try to understand what the program below is doing, and why.

```
#include <iostream>
using namespace std;

int main() {
    int i=2;
    unsigned u=3;
    cout << " u=" <<  u << " i=" <<  i << endl
         << "-u=" << -u << "-i=" << -i << endl
         << "u*-i=" << u*-i << endl
         << "-u*i=" << -u*i << endl;
}
```

11. Never use an empty compound statement

    Instead of using an empty compound statement as in

    ```
    while(...);
    ```

    you should use {} as in

    ```
    while(...)
        {}
    ```

    Code containing empty compound statements is very error-prone as can be seen from the following snippets of code, which are not equivalent:

    ```
    int i=-1;
    for(unsigned count=0; Compute(); count++);
    i++;

    int i=-1;
    for(unsigned count=0; Compute(); count++)
    i++;
    ```

12. Allow for multiple inclusion of include files

    Take provisions so that including any include file more than once does not cause any problems.

    A common trick to do this is to surround the contents of an include file, say `header.h` in our example, with pre-processor directives similar to the following:

    ```
    #ifndef HEADER_H
    #define HEADER_H
    ```

```
⋮
#endif
```

(If you go by that trick, you should refrain from using any identifiers whose names end with `_H`.)

13. Avoid using `??` and `<%`, `%>`, `<:`, `:>`, `%:`

    Do not use any of these character sequences anywhere in your code (including comments). Some combinations of two resp. three characters, so called digraphs resp. trigraphs, have a very special meaning since ANSI C, which is probably not what you wanted in the first place.

14. Do not use `and`, `bitor`, `or`, `xor`, `compl`, `bitand`, `and_eq`, `or_eq`, `xor_eq`, `not`, `not_eq`

    Do not use any of these as identifiers, for they have have been defined as alternative tokens for `&`, `|`, and so forth, since ANSI C.

15. Assignment operators should make sure that self assignment does nothing

16. Declare local variables near first use

17. A class with any of destructor, assignment operator, copy constructor generally needs all three

18. Constructors should use parental constructors, not assignment

19. Constructors should use member initialization list, not assignment