C++ Programming Practice Guidelines

Geotechnical Software Services
Version 1.0, January 2011
Copyright © 2011

This document is available at http://geosoft.no/development/cpppractice.html

Table of Content

1 Introduction
  1.1 Layout of the Recommendations
  1.2 Recommendations Importance

2 Transition from C to C++
3 Constructors, Destructors and Assignment Operators
4 Operators
5 Inheritance
6 Exceptions
7 Miscellaneous
8 References

1 Introduction

C++ programming guidelines often mix coding style issues, good programming practice and program design issues in a somewhat confusing manner. The present document focus mainly on good C++ programming practice. For guidelines on C++ programming *style* refer to the C++ Programming Style Guidelines. For guidelines on programming design issues, consult writings on programming patterns, object-oriented software design etc.

The recommendations given in this document are based on writings by Scott Meyers (see [1] - [3]), as well as other sources.

1.1 Layout of the Recommendations.

The recommendations are grouped by topic and each recommendation is numbered to make it easy to refer to during code reviews.

Layout of the recommendations is as follows:

Recommendation short description
`Example if applicable`
Motivation, background and additional information.

1.2 Recommendation Importance

In the guideline sections the terms *must*, *should*and *can* have special meaning. A *must* requirement must be followed, a *should*is a strong recommendation, and a *can* is a general guideline.

2 Transition from C to C++

1. #define statements should be avoided.

```
const double PI = 3.1415;     // NOT: #define PI 3.1415
```

`#define` is not part of the C++ language definition and since C++ provides constructs that make `#define` superflous, these should be preferred.

Instead of #define constants, use const variables, or better yet: Use member functions to access constants (See [1]).

Instead of typeless macros use functions with template parameters.

2. The iostream library should be preferred to stdio.

```
#include <iostream>     // NOT: #include <stdio.h>
```

The stdio has been replaced by the much more powerful iostream library, and when programming C++ the latter should be preferred.

3. C++ style casts should be preferred to C style casts.

```
static_cast<double> intValue;     // NOT: (double) intValue
```

4. *0* should be used instead of *NULL*.

*NULL*is part of the standard C library, but is made obsolete in C++.

5. References should be preferred to pointers.

Pointers should be used if and only if it should be possible for the referred object to be *null*.

Example: If a person is modelled as an object, the persons parents should be *references* to persons (everybody has parents) while siblings should be *pointers* to persons.

6. const should be used wherever possible.

The *const* keyword is a documentation aid.

In particular, member functions that does not affect the state of an object is to be declared *const*. These are the only functions which may be invoked on a const object.

7. Pass-by-value should be avoided for objects.

```
myMethod (const SomeClass &object) // NOT: myMethod (SomeClass object)
```

There are two reasons for this. First is of performance. Pass by value for objects always involves creating a temporar object using the copy constructor of the objects class and destroying the object again on the method exit.

Second reason is that objects passed by value through a base-class variable will in effect behave as a base-class object without the extended information defined by the derived class.

8. Variable argument lists (...) should be avoided.

Variable argument lists prohibit strong type checking as provided by C++. In most cases variable argument lists can be exchanged by function overloading and by using default arguments.

9. new and delete should be preferred to malloc, realloc and free.

In C++ there is no need for the older memory management functions malloc, realloc and free. It enhances readability of the code to use one consistent set of memory management methods. Also it makes deallocating memory safer, since it is dangerous to do `delete` on a `malloc`'ed object or `free` on a `new`'ed object.

3 Constructors, Destructors and Assignment Operators

10. A copy constructor and an assignment operator should always be defined for classes with dynamically allocated memory ([1], Item 11).

Unless the copy constructor and the assignment operator is explicitly defined, the compiler will generate them automatically. If the class use dynamically allocated memory, the default generated copy constructor and assignment operator will most often not behave as expected. The exception to this if when multiple objects of the same class indeed *should* share a data area. In this case it is necessary to make sure that the shared data is not deallocated as long as there are references to it.

11. The assignment operator should always return a reference to *this.

```
MyClass& MyClass::operator= (const MyClass& rhs)
{
  ...
  return *this;
}
```

This is done to have the user defined assignment operator mimic assigment of built in types.

For instance will the following statement be both possible and meaningful:

```
MyClass a, b, c;
a = b = c;
```

12. The assignment operator should always check for assignment to self.

```
MyClass& MyClass::operator= (const MyClass& rhs)
{
  if (this != &rhs) {
    ...
  }

  return *this;
}


MyClass& MyClass::operator= (const MyClass& rhs)
{
  if (*this != rhs) {
    ...
  }

  return *this;
}
```

Which of the versions above that are chosen depends on the class at hand. First version simply just check if the rhs and this points to the same location in memory. This is sufficient in most cases. The second version checks for equality using operator == (assuming it is defined).

13. The assignment operator of a derived class must explicitly perform the assignment of its base class.

```
Derived& Derived::operaor (const Derived& rhs)
{
  if (this != &rhs) {
    Base::operator= (rhs);

    ... // Then do assignment of own stuff
  }

  return *this;
}
```

The base class assignment is never automatically called as one could beleve. Some compilers will not accept the above construct if the assigment operator of the base class is automatically generated. In this case use:

```
static_cast(*this) = rhs;
```

which downcasts this to its base class and force the assignment through this base class reference.

14. Initialization should be preferred to assignments in constructors.

Even if the syntax of initialization is questionable, there are benefits of using it rather than constructor assignments.

15. The destructor should be virtual if and only if the class contains virtual methods ([1], Item 14).

When `delete` is used on a derived object where the base class destructor is not virtual, only the base class destructor will be called.

On the other hand, if a class does not contain any virtual members, it should not be used as a base class at all. In these cases it does not make sense to declare the destructor virtual both because it is not needed and because it will increase objects of the class with a completely unneccesary vptr and an associated vtbl.

4 Operators

16. Both == and != should be implemented if one of them are needed.

```
bool C::operator!= (const C& lhs)
{
  return !(this == lhs);
}
```

In most cases the != operator can be derrived from the == operator as shown in the example. This is not automatically done by C++ as one could expect however.

17. Operators &&, || and , must never be overloaded ([2], Item 7).

Problem with user defined versions of these operators is that they will not get the same behaviour as the default versions for the standard types.

C++ employs a short-circuit evaluation of boolean expressions which means that as soon the truth or falsehood of an expression has been determined, evaluation of the expression ceases. There is no way this behaviour can be transerred to the user defined operator methods.

5 Inheritance

18. "Isa" relationship should be modelled by inheritance, "has-a" should be modelled by containment.

```
class B : public A     // B "is-a" A
{
  ...
}


class B
{
  ...

  private:
    A a_;     // B "has-a" A
}
```

19. Non-virtual methods must never be redefined by a subclass.

There are two reasons for this. First is that if the method *needs* to be redefined, then the subclass should not inherit from the base class in the first place. It fails to be an "is-a" of the base class.

Then there is a technical reason. A non-virtual function is statically bound and a reference to the base class will always invoke the method of the base class even if the object is derived from the base class.

20. Inherited default parameters must never be redefined.

This is related to the previous rule. ...

21. Private inheritance must be avoided.

```
class C       // NOT: class C : private B
{             //      {
  ...         //        ...
  private:    //      }
    B b_;
}
```

While public inheritence model an "is-a" relationship, private inheritance doesn't model anything at all, and is purely an implementation construct for sharing code with the class being inherited. This is better achieved by containment.

Protected inheritance must be avoided.

22. Downcasting must be avoided.

```
derived = static_cast<DerivedClass*> base;
```

The need for downcasting reveals a design flaw. Properly written C++ code should never branch on the type of objects (*"if object A is of type so-and-so do this, else do that"*). Use virtual functions instead.

7 Exception

23. Exceptions should be caught by reference.

```
try {
  ...
}
catch (Exception& exception) {
  ...
}
```

7 Miscellaneous

24. Properly differentiate between member methods, non-member methods and friend methods.

- Virtual methods must be members
- operator>> and operator<< are never members. If they need access to non-public members of the class to which they are associated they should be friend functions.
- If a method needs type conversion on its left-most argument the method must be a non-member method. If in addition it need access to non-public members of the class to which it is associated it should be declared friend function
- Everything else should be member methods.

25. Implicitly generated methods that are not to be used should be explicitly disallowed.

```
class C
{
  ...

  private:
    C& operator= (const C& rhs); // Don't define
}
```

By declaring such methods private and skip their definition, attempts to call them will be trapped by the compiler.

Methods that are implicitly generated by the compiler if they are not explicitly defined are:

- Default constructor (`C::C()`)
- Copy constructor (`C::C (const C& rhs)`)
- Destructor (`C::~C()`)
- Assignment operator (`C& C::operator= (const C& rhs)`)
- Address-of operator (`C* C::operator&()`)
- Address-of operator (`const C* C::operator&() const;`)

26. Singleton objects should be preferred to global variables.

```
class C
{
  public:
    static const C* getInstance()
    {
      if (!instance_) instance_ = new C;
      return instance_;
    }

  private:
    C();
    static C *instance_; // Defined in the source file
}
```

The singleton approach solves the problem of the undefined order of initialization of global objects which can result in objects referring to other objects not yet initialized.

In general, there is no need to use global objects in C++ at all.

27. Functions that can be implemented using a class' public interface should not be members.

28. A public member function must never return a non-const reference or pointer to member data ([5], Rule 29).

Returning a non-const reference to memeber data violates the encapsulation of the class.

29. The return type of a function must always be specified explicitly.

```
int function()     // NOT: function()
{                  //      {
  ...              //        ...
}                  //      }
```

Functions, for which no return type is explicitly declared, implicitly receive `int` as the return type. Functions must never rely on this fact.

8 References

[1] Effective C++ Second Edition, Scott Meyers - Addison-Wesley 1998

[2] More Effective C++, Scott Meyers - Addison-Wesley, 1996

[3] How Non-Member Functions Improve Encapsulation, Scott Meyers - C/C++ Users Journal, February 2000

[4] C++ Programming Style Guide

[5] Programming in C++, Rules and Recommendations, M. Henricson / E. Nyquist, 1992