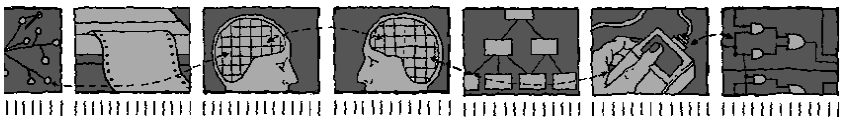
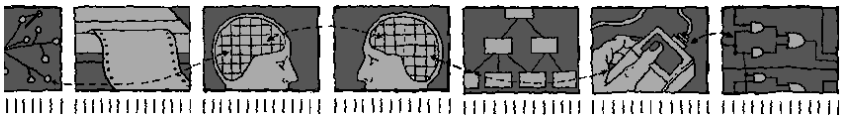


Department of Computing Science and Mathematics
University of Stirling



Defensive C++
Programming Guidelines for those who dislike Debugging

Jerry Swan

Technical Report CSM-194

ISSN 1460-9673

November 2012

Department of Computing Science and Mathematics
University of Stirling

Defensive C++
Programming Guidelines for those who dislike Debugging

Jerry Swan

Department of Computing Science and Mathematics
University of Stirling
Stirling FK9 4LA, Scotland

Telephone +44 1786 467 421, Facsimile +44 1786 464 551
Email jerry.swan@cs.stir.ac.uk

Technical Report CSM-194

ISSN 1460-9673

November 2012

Abstract

C++ has a reputation as a difficult and complex language in which one can achieve anything, but with an attendant risk of (what is politely termed) ‘undefined behavior’ when practiced by the uninitiated.

We offer guidelines that are intended to eliminate common causes of systemic and hidden error (e.g. ignorance of copy-assignment semantics) and also describe a number of practices that facilitate both design robustness and ‘programming in the large’.

Acknowledgements

Thanks to Bob Archer, who helped start the ball rolling.

Chapter 1

Introduction

"Programming can be fun, so can cryptography;
However they should not be combined"

Kreitzberg and Shneiderman

This document outlines practices for C++ software development. It is not intended to be overly prescriptive. Most observations or 'rules' about programming style usually work better as guidelines, and work *much* better if the rationale behind the guidelines is a sensible one. Blindly avoiding certain constructs or following rules without understanding them can lead to just as many problems as the rules were intended to avoid.

Also, many opinions on programming style are just that: opinions. It is wise to avoid getting dragged into "style wars," because on such religious issues, opponents can never seem to agree; agree to disagree, or stop arguing.

As far as possible, standards are therefore guidelines of the form "maximise use of X", "minimise use of Y" or "prefer X to Y".

Exceptions to guidelines are given when applicable.

Before we delve into the sometimes dry, dusty and arcane depths of the C++ language, below are some very general guidelines that are in the spirit of the overall aim of this document, which is to simplify the initially daunting task of writing robust C++ programs.

1.1 Write bug-free software...

OK, writing a program of any appreciable size that is completely bug-free is probably impossible, but that's no excuse for not trying. The fewer bugs you put into your software, the less work it is to debug. See [13] for examples of putting this ethos into practice.

1.2 ... but plan for debugging

Write all of your classes on the assumption that they are going to have to be debugged. In particular, if you find yourself needing direct access to encapsulated data members for debugging purposes, then do the job properly and write an output operator (**operator** $<<$) for your class.

1.3 Don't use hacks

If you don't have time to do it right the first time, where are you going to get time the second time?

Given the choice between quick hacks or a proper solution go for the proper solution. You will probably end up having to implement it anyway and will waste time on the quick hack.

1.4 Just because you can doesn't mean you should

C++ supports many different ways of modelling problems. Some of these features are good; others are bad; most are good in the right circumstances and bad in the wrong circumstances.

Never use a feature 'just because it's there' or 'just because person/organisation X does it that way' — use it because it's the right feature to use.

1.5 Know the Language

As with natural languages, the ability to express concepts clearly in a programming language requires fluency. Knowing enough to use the language in the manner intended by its designers will yield significant long-term benefits.

The bibliography contains a number of excellent books on all aspects of software engineering. Those wishing to become a 'native speaker' of C++ might be interested in some of the idioms in common use in the C++ community: many of them are documented at:
`http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms`.

Chapter 2

Principles of Software Development

"Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do"

Donald Knuth

A number of fundamental principles of modern software development are summarised here.

Adherence to these principles undeniably requires additional design and coding effort in comparison with more *ad hoc* methods. The underlying premise is that a linear, knowable development cost (of the design phase and the greater number of keystrokes needed to write a given section of code) be exchanged for the non-linear, unknowable cost of code re-factoring and debugging.

2.1 Minimise client responsibilities

It is imperative to design APIs so as to minimise the responsibilities of users (a.k.a. *client programmers* or clients) of the interface. This serves to isolate complexity within the implementation, rather than enforcing it on all users of the interface.

2.2 Write for people

Writing software that can be understood by a computer is hard; writing software that can be understood by a person is even harder. Nonetheless, every effort to make code easier to understand is likely to prove worthwhile in the longer term.

Never assume that someone reading your code possess all the knowledge about the code that you have - even if that person is your future self.

It is essential to write code with the client programmer in mind. In fact, a good way to determine the value of a given interface design is to write some client code to see how simple and intuitive it is to use [7].

2.3 Prefer compile-time to runtime errors

The benefits of being able to detect errors of logic in program code at compile-time are self-evident. This is clearly not possible in all situations, but there are many circumstances where we can use the features and constraints of the programming language to eliminate likely sources of coding error.

Programs should be designed for compile-time rather than runtime error trapping wherever possible.

2.4 Design by Contract

Design by Contract [14] is one of the most practical means of ensuring software reliability. In essence, a *software contract* specifies a division of responsibility between the caller and implementer of a function. Ideally, this contract is expressed in terms of executable assertions in the code. In object-oriented programming, design by contract is central in ensuring that inheritance hierarchies are correctly conceived.

2.5 Maximise encapsulation

Encapsulation is central to the creation of maintainable software.

The Y2K Problem is an artefact of poorly encapsulated software - it is not unreasonable to use 2 digits to represent the year, but it is a bad idea to make the underlying implementation visible to the client, resulting in many millions of lines of client code that need to be checked for integrity.

Object-oriented design seeks to partition the problem into classes, each of which a) has a well-defined mission statement and b) offers high-level *services* to other classes, rather than simply acting as a 'dumb bucket of bits' that we hope the client uses in the manner we had in mind when we wrote it.

2.6 Program in the vocabulary of the problem domain

If one aspires to the ideal described by Donald Knuth at the beginning of this chapter, then a significant goal of source code is to act as a human-readable model of the requirements, in a form also suitable for machine understanding.

At the highest level of description, specifications are given in terms of the end-user's vocabulary, a.k.a. the *problem domain*.

At the lowest implementation level, program code is written in terms of operations on the scalar types of the language (**int** , **float** , **char** $*$ etc), a.k.a. the *solution domain*.

In order to write code which is comprehensible to others and hence maintainable, it is desirable to *"declare types in the vocabulary of the problem domain"* [2]. This is of great importance in bridging the considerable gap between the two vocabularies.

For example, it is often desirable to declare a new type (i.e. **typedef** or **class**) to represent an entity in the problem domain (e.g. product_id_type), even though at the current stage of development this type could be represented by a type in the solution domain (e.g. **int** or string).

As our appreciation of the problem domain changes (whether as a result of a change in specification or an increase in our understanding), the information hiding afforded by this technique localises these changes and prevents them from having to be made throughout client code.

The use of typedefs and classes should be therefore preferred over raw scalar types for all but the lowest level operations.

2.7 Maximise compile-time type-checking

It is highly desirable that variables be *type-safe*, i.e. always representing a *valid* object of their nominal type. This relieves the client of the responsibility for checking validity and avoids the possibility of his forgetting to do so.

For example, the practice of using a NULL pointer as a sentinel value has doubtless cost thousands of hours of debugging effort, particularly in poorly-documented APIs. A number of more benign alternatives to this practice are outlined in this document.

In languages with support for static (i.e. compile-time) type-checking such as C and C++, it is possible to use the type system to constrain (i.e. require or prohibit) the permissible operations on objects of that type.

2.8 Say everything exactly once

"Don't Repeat Yourself" (DRY) a.k.a. "Once and Only Once" (OAOO) is one of the basic maxims of *Extreme Programming* [1]. When something (code or data) is duplicated it is necessary to keep the copies in step with each other.

Sometimes, e.g. for efficiency purposes, it may be necessary to cache data (thereby creating a copy). In such a case, integrity checks should enforce the correspondence between the original and the cache value.

Should duplication appear inevitable, the following should be considered:

- Code duplication can be avoided by factoring out common code sequences (e.g. as subroutines, in base classes, or as class or function templates).
- Data that is derived from another source should be recalculated if the original data has changed. The *observer* design pattern [10] and *lazy evaluation* [16] are key techniques for doing this.

Chapter 3

Readability

"A good name for a routine clearly describes everything the routine does."

Steve McConnell

3.1 General Guidelines

If you find that a 'good' name for a routine is more like a paragraph than a short sentence then the routine is trying to do too much and should be split up.

Some other general readability guidelines:

- Choose meaningful names for variables. Note that, depending on context, variable names such as i and j might indeed be meaningful (for example as loop indices).
- Try and achieve a reasonable balance between text and whitespace. In particular, code that is evenly-spaced is easier to read than code that is highly terse.
- Prefer to declare one variable per line. In particular, dont mix type and pointer-to-type declarations on a single line — the eccentricities of C++ declaration syntax can easily cause confusion in such cases.
- Use parentheses to clarify expression evaluation order for all but the simplest expressions.
- Split complex expressions up into subexpressions, using temporary variables if necessary.

3.2 Commenting your Code

As far possible, try and write code so that it would be intelligible to a listener if it were read out loud. This minimises the need for explicit comments.

Place your comments so as to make it obvious which specific parts of the code they apply to.

Stroustrup [18] suggests that a mission statement for each nontrivial class should be provided, giving a more detailed explanation of its purpose. He also suggests that there should be a comment for each nontrivial function/ method detailing:

- Its purpose, if not obvious from the function name (it should be!).
- The space/time complexity of the algorithm used (if it is processor or storage intensive).
- Any assumptions made about its environment that arent covered by pre- and post-conditions[1] (ideally, there shouldn't be any).
- Any side-effects caused by the function that aren't covered by pre- and post- conditions. (ideally, there shouldn't be any).
- A comment in any place where the code is non-obvious/ non-portable.
- A comment for any language or compiler hacks.

Bertrand Meyer offers some definitive advice in [14]:

- In general, comments should be at a level of abstraction higher than the code that they document. In the case of header comments, the emphasis should be on the "what"of the routine rather than the "how" of the algorithm used.
- Assume the reader is reasonably competent. There is no need to repeat in the header comment information that is obvious from the immediately adjacent program text. For example, the header comment for a routine beginning with

  ```
  Line tangent( const Circle& c, const Point& p );
  // tangent to circle c through point p
  ```

  would be better commented as

  ```
  // tangent to c through p
  ```

  as it is clear from the function header that c is a circle and p is a point.
- Avoid noise words and phrases. An example is "Return the. . ." in explaining the purpose of functions. In the above case, writing "Return the tangent to..." does not bring any useful information as the reader knows that the function must return something (or else be void). Another example of a noise phrase is "This routine computes. . ." or "This routine performs. . .". For example, instead of:

  ```
  // this routine updates the display
  // according to the user's last input
  ```

  it is better to have:

  ```
  // update display according to last user input
  ```

- Be consistent. If a method of a class has the comment "length of string", a routine of the same class should not say "update width of string" if it acts on the same attribute.
- Much of the important semantic information about a routine is captured concisely by the use of pre- and post-conditions.

[1] See the section on 'Design by Contract' for more details on pre- and post- conditions

3.3 Code Formatting Conventions

Few programming issues are as hotly debated as code-formatting conventions, so we will try and avoid saying anything more inflamatory than strictly necessary. Ultimately, it is probably better to harmonize with the prevailing style of any project that you join, even if you happen to disagree with it.

The formatting conventions used in the C++ standard [4] and hence in the standard libraries were initially inherited from C [11], but now also include some C++ specifics:

- **lower_case_with_underscores**
  Used for type-names, variable names and fuctions/ methods. e.g.

```
int array_index;
class some_class;
void some_function( int i );
```

- **UPPER_CASE_WITH_UNDERSCORES**
  Used for preprocessor macros e.g.

```
#define DEBUG_WHERE std::cout <<__FILE__ << __LINE__
```

- **MixedCaseWithoutUnderscores**
  Used for template parameter names, e.g.

```
template < class InputIterator >
InputIterator next( InputIterator it );
```

The above conventions are also widely used in the international C++ community e.g. in the BOOST libraries [3] and are the ones used in this document. We also adopt from [7] the convention of using an underscore suffix on class atributes, e.g.

```
class address
{
    int         house_number_;
    std::string address1_;
    std::string address2_;
    // ...
};
```

As popularised by Microsoft [13], a combination of mixed case for class names and Hungarian notation for variable names also enjoys widespread use. In statically-typed languages like C and C++, Hungarian notation is arguably redundant and certainly makes variable names less easy to decipher.

Chapter 4

Coding Guidelines

Each section starts with a brief description, very often of the form "don't do X" where X is some technique considered to be poor (or possible fatal) coding practice. This is followed by a rationale, providing some of the background. Be warned that some of the code in the rationale section might not be correct – bad code is sometimes used to illustrate a point, though this is always annotated as such. Any exceptions are then discussed.

The coding guidlines are presented in four groups, appearing in order of their importance to program integrity:

- **Fatal coding errors**
  Describes practices (unfortunately all too common) that will lead to *undefined behaviour* on the part of the program. In the strict terms of the C++ standard [4] 'undefined behavior' means that the program may behave *in any manner whatsoever*, e.g. it might (perhaps temporarily or intermittently) appear to work as expected; it might write to protected memory and be terminated by the operating system, or it might send an email to Bjarne Stroustrup telling him what a bad programmer you are.
- **Defensive Programming**
  Describes practices that minimize the likelihood of undefined behaviour arising, whether currently or during subsequent modification to the code.
- **Good Practice**
  A miscellaneous collection of issues intended to lead to more readable and robust code.
- **Niceities**
  Practices that it would be helpful to adopt if development time permits.

4.1 Fatal coding errors

4.1.1 Implement copy-assignment correctly

For many user-defined classes, it is useful for copy-construction and assignment to operate in the same manner as for built-in types (**char**, **int**, **float** etc):

```
int i = 7, j;
int k( i );
j = i;
my_class a, b;
my_class c( a );
b = a;
```

In addition to the explicit cases above, the copy-constructor/ assignment operator are implicitly invoked whenever an object is passed or returned *by value*. Classes with correctly-implemented copy-assignment can safely appear as attributes of other classes and be stored in containers such as std :: vector, std :: map, std :: set etc. In order to guard against undefined behaviour arising from copy-assignment, it is vital to ensure that each class/ struct you create has the correct *copy-assignment category*:

1. **Non-copyable**
2. **Compiler can copy**
3. **Implementor must copy**

This is necessary because (in the absence of explicit programmer intervention to the contrary) the C++ compiler may synthesise (i.e. silently generate) the following operations for a class X[1]:

The default constructor: X();

The copy constructor: X(**const** X& copyMe);

The destructor: ~X();

The assignment operator: X& X::**operator** =(**const** X& copy_me);

The synthesised operations act as follows:

1. The default constructor calls the default constructor for any direct superclass and for all of the attributes of X.
2. The copy constructor will call the copy-constructor (which might itself be synthesised) for each direct superclass and attribute of X. Note that for pointer attributes this will simply lead to the pointer attribute being copied, not the thing being pointed to.
3. The destructor will call the destructors for each attribute and direct superclass of X. Note that when a pointer is destroyed the memory it points to is *not* automatically deleted.
4. The assignment operator will call the assignment-operator (which might itself be synthesised) for each direct superclass and attribute of X. Note that for pointer attributes this will simply to the pointer attribute being copied, not he thing being pointed to.

For classes that manage resources(e.g. via memory allocation, file handles, semaphores etc.), the compiler-synthesised versions of these operations may lead to resource leaks or undefined behaviour.

The solution is to *always* identify the correct copy-asignment category for each of your classes and implement any corresponding responsibilities. The description and responsibilities for each category are given below.

[1]Note that by default, the compiler also generates two address-of operators: X∗ **operator**&(); **const** X∗ **operator**&()**const**;

Non-copyable

Some classes do not rightfully support copy construction or assignment. For example, there are no intuitively obvious semantics for copying objects representing a file on disk, so we might simply decide that our file class is not copyable. Other examples include objects deemed too large to copy and *singleton* objects [10].

A class that is not copyable should enforce this by *declaring* the copy-constructor and assignment operator in the **private** section of the class, *but without a corresponding implementation*:

```
class non_copyable
{
private:
    non_copyable( const non_copyable& );
    // ^ Implementation intentionally omitted
    non_copyable& operator =( const non_copyable& );
    // ^ Implementation intentionally omitted
};
```

Any attempt to copy or assign a non_copyable object (whether by the compiler or in client code) will now result in a compile-time error.

Since any subclass of a non-copyable class is necessarily also non-copyable, non-copyability can be made easier and more explicit by simply privately inheriting from the above class (or the equivalent in the header file <boost/utility.hpp> in the BOOST library [3]):

```
class my_class
: private non_copyable
{
public:
    //
};
```

Compiler can copy

This category applies when the synthesised copy operations listed in the previous section suffice to cause objects of the class to behave in the desired manner. This will generally be the case if your class does not manage resources and all attributes have valid copy-assigment.

Implementor must copy

Classes managing resources (e.g. memory allocation, files, semaphores etc.) generally fall into this category.

A complete sample implementation of a class in this category is given in Appendix A.

If in doubt as to the copy semantics of a class, deem it to fall into the 'not copyable' category and solicit design advice.

References: See [7], FAQ 195 - FAQ 210 and [17], 'Item 45' for more information.

4.1.2 Don't perform bitwise copies on objects

Rationale: C++ classes, unions and structs are not, in general, the same kind of thing as C structs. In particular, it can be a fatal error to manipulate non-POD[2] C++ objects as raw memory. Such manipulations include passing an object on a variable length argument list (e.g. as used in printf) or using malloc, calloc , realloc , memcpy, memset or free.

Non-POD C++ objects *cannot* rightfully be copied in this fashion - they must be permitted to copy themselves via their copy constructors and assignment operators, as discussed in the guideline 'Implement copy-assignment correctly'.

4.1.3 Don't copy polymorphic objects by value

Rationale: If a subclass is passed to/ from a function that takes/ returns a superclass parameter/ result *by value*, then the derived class will be 'cut down' to the base class. This means that any data or redefined virtual functions in the derived class will not be present in the 'cut down' object.

There are no circumstances in which this is desirable, and this situation should never arise in correctly designed software.

If the base class is an abstract class, the *runtime* support for some compilers may terminate the program with a 'pure virtual function called' error under these circumstances. The solution is to change the offending function to pass/ return the superclass *by reference*.

References: See [16], 'Item 33' for a discipline that will give a *compile-time* error in these circumstances.

4.1.4 Beware of undefined expressions

Rationale: Don't use functions that have side effects on an object when the object appears more than once in the same expression - the order of evaluation (and hence the resulting program behaviour) is undefined.

Example: The most common misuse occurs with the increment and decrement operators, e.g.

```
a[ i ] = ++i;     // undefined behaviour
```

Note that parenthesising such expressions will not help.

References: A more pernicious but less common variant is detailed in [5] p105 in the chapter on 'Operator overloading'.

4.1.5 Don't divide by zero

Rationale: Dividing by zero is undefined. Depending on the circumastances and/ or the compiler you might get an exception, an arithmetic error or a NaN ('Not a Number') or something else. If you do get a NaN, the program will continue running but with potentially unpredictable results, e.g. all comparisons on a NaN return **false**, which might for example cause a sorting algorithm to exhibit undefined behaviour. Note that this gives a way of testing for NaN: x==x is defined to be **false** if x is NaN.

[2] POD stands for 'Plain Old Data' - see `http://www.parashift.com/c++-faq-lite/intrinsic-types.html#faq-26.7` for more details

4.1.6 Beware of the 'static initialization-order fiasco'

Rationale: The order of initialisation of static objects that compile to different object files is *undefined*. Fatal errors can therefore arise when there is an initialisation order dependency between static objects in different object files.

The simplest solution to this problem is to replace each static member object of a class with a static member function that returns a static local (by the most appropriately restrictive of value/const-reference/non-const reference). This provides what is known as "construct-on-first-use semantics" - the static object is initialised the first time the function is called.

e.g. replace:

```
// In box_of_cornflakes.hpp:
class box_of_cornflakes
{
    // ...
    static double average_contents_per_box_;
};

// In box_of_cornflakes.cpp:
double box_of_cornflakes::average_contents_per_box_ = 4137;
```

with:

```
class box_of_cornflakes
{
    // ...
    static double& get_average_contents_per_box()
    {
        static double average_contents_ = 4137;
        return average_contents_;
    }
};
```

Exceptions: There have been ammendments in the C++ standard [4] to help address this issue. In particular, it is now possible to initialize static members in the class declaration (much as is possible in Java):

```
class box_of_cornflakes
{
    // ...
    static double& average_contents_per_box_ = 4137;
};
```

Be aware that not all compilers yet support this functionality.

References: A number of possible alternative solutions are outlined in [7], pp169-171.

4.1.7 Never return a temporary object by reference

Rationale: Doing so creates a *dangling reference*, resulting in undefined behaviour.

Returning an object by reference is safe only if the lifetime of the object exceeds the lifetime of the referent. In practice, there are two ways in which you might end up returning a temporary by reference:

1. **Returning a local variable by reference**
   This is of the form:

```
int& f()
{
    int i;
    return i;
}
```

Since i is a *local* variable, its stack storage is reclaimed on exit from $f()$, leaving a dangling reference. Most compilers are smart enough to issue a warning about this.

2. **Returning a const reference parameter as a const reference**
   For a function $f()$ that takes a const reference to some type T as an argument, it is possible that the actual argument will be a temporary, unnamed object. A reference to this argument returned from f() will exceed the lifetime of the temporary object:

```
string create_temp() { return "hello"; }
const string& evil_fn( const string& x ) { return x; }

int main()
{
    // unnamed temporary is created which
    // dies at the end of the following line:
    const string& y = evil_fn( create_temp() );
    // undefined behaviour in the next line,
    // since the object y refers to no longer exists:
    std::cout << "y = " << y << "\n";
}
```

This case is more pernicious, since most compilers dont currently issue a warning about it.

4.2 Defensive programming

4.2.1 Don't ignore compiler warnings

Rationale: Programs should compile without warnings — if code habitually generates lots of warnings, there is a tendency to miss any warnings that are genuinely indicative of coding errors.

Exceptions: Sometimes warnings are issued incorrectly. They are indicative of errors or inadequacies in the compiler or third-party code rather than errors in your code. If this is indeed determined to be the case, it may be possible to work around or otherwise eliminate them.

4.2.2 Maximize type-safety

It is extremely desirable that instances of a type should always be usable without causing a nasty surprise for the user. Design by contract allows us to specify valid object states, function results and parameter values, but such specifications are checked at runtime. It is clearly far better to make full use of compile-time type-checking to eliminate the necessity for such runtime checks.

For example, consider the following common C-programming practice:

```
enum animal_type {
    dog=0,
    cat=1,
    rabbit=2,
    num_animals=rabbit+1
};
```

This enumeration is not type-safe, since num_animals is not an actual animal. The user of an instance of animal_type is therefore required to check that they are in possession of a valid animal before doing anything else with it. Forcing such responsibilities on the user of animal_type is unnecessarily error-prone.

Some further guidelines:

1. Constructors should create a valid object, not one in some 'not yet fully initialised' state.
2. Ensure that member functions leave objects in a valid state once constructed.
3. Try to avoid using a 'capability query' based on some notion of object type, since this is precisely the situation in *subtype polymorphism* should be used. Examples of such 'type-queries' include:

   ```
   if( x.get_type() != ostrich_type )
       x.fly();
   ```

   or:

   ```
   if( dynamic_cast< ostrich * >( x_ptr ) != NULL )
       x_ptr->fly();
   ```

It is also sometimes tempting to use some particular value of a type to indicate some kind of error-code instead of a legitimate value.

For example, a common way of indicating errors is to use a sentinel return value, for example, returning a negative index to indicate that an item is not found in a list, or returning a NULL pointer.

To some degree, the adoption of design by contract reduces the perceived necessity for these practices, since logic errors should be trapped by post-conditions and invariants rather than being passed back to the caller.

There are a number of more benign alternatives to functions having such 'dual-purpose' output parameters:

- Return a std :: pair or structure containing the result, together with an error flag or enumeration indicating its validity.
- Borrowing an idea from the functional programming community, we can use an optional <T> template class, such as the one provided by the BOOST library [3].

4.2.3 Don't let any error go undetected

The recommended strategy for handling errors generated by the client (e.g. if a function receives a parameter value that does not meet its contract) is to throw an exception.

The implication of this is that the exception will be caught at some point higher up the call-stack, where there is enough information about the context of the error for a recovery strategy to be put in place. If no successful handler exists for the exception, the program will terminate (possibly with a message of the form 'this program has terminated unexpectedly').

It is therefore sensible to always have a default exception handler in place in main, so that at as a minimum any diagnostic information associated with the exception can be printed out:

```
#include <iostream> // for std::cerr
#include <stdexcept> // for std::exception

#include <cstdlib>  // for EXIT_SUCCESS

//////////////////////////////////

int run( int argc, char * argv [] )
{
    // ...
    // actual main body of your program
    // ...
    return EXIT_SUCCESS;
}

//////////////////////////////////

int main( int argc, char * argv [] )
{
    try
    {
        return run( argc, argv );
    }
    catch( std::exception& ex )
    {
        std::cerr << "caught exception in main, what() = " << ex.what() << '\n';
    }
    catch( ... )
    {
        std::cerr << "caught unknown exception in main\n";
    }

    return EXIT_FAILURE;
}
```

4.2.4 Minimise the visibility of data

Rationale: The greater the number of entities that have access to an item of data, the greater the potential for error. Software that is inextricably dependent on its data format is extremely difficult to maintain.

There are effectively four categories of visibility, given here in increasing order:

1. Block scope or 'local' data, i.e. only visible within some function.
2. Class attributes, visible to the declaring class (and possibly also to friends and subclasses).
3. Data intended for visibility only within a single .cpp file (a.k.a. 'file scope'). These must be declared **static** to enforce the intended visibility.
4. Variables visible across source modules. These include 'global' variables declared in header files and public or protected static class attributes.

Always attempt to have your data belong to the most restrictive category possible.

4.2.5 Restrict access to class attrbutes

Rationale: Never implement an accessor method that returns a non-constant pointer / reference to a class attribute, since this exposes the internal state of the object. This means that the class invariant (see the chapter on 'Design by Contract') can be violated without warning, since a client can change the value of an attribute at any subsequent time via that pointer/ reference.

In general, only provide access functions to attributes if there is a compelling reason (the class should be providing services to other classes, rather than simply being 'Plain Old Data'). If implementing an accessor method proves necessary, provide it at the most restrictive possible level, listed in (approximate) order of restrictiveness:

- Grant read-only (const) access.
- Grant friendship to another function.
- Grant protected access.
- Grant friendship to another class.
- Grant public access.

Exceptions: Very occasionally a class is so simple it is clear that:

1. Its attributes are unlikely to be changed in future.
2. The class has no invariant properties.

There are very few genuine examples of such classes in practice. One example might be a class representing a mathematical vector that has a subscripting operator declared as `T& operator [](int x);`. Since any combination of values of the elements of a mathematical vector is valid, there is no invariant and this non-constant access is acceptable.

4.2.6 Make maximal use of const

Rationale: **const** is a vital tool for helping you write correct code - it can be viewed as part of the Design by Contract paradigm.

By declaring an object or method to be **const** you are making a contract with the compiler - namely "I will not attempt to modify the object that this function has been called on". The compiler will enforce this contract and will not compile a function that attempts to modify the object.

Exceptions: const means 'abstractly constant', not 'bitwise constant', i.e. we are concerned with the object's *conceptual state*, not its physical state.

This means that it is allowable to modify a nominally constant object as long as doing so does not change its *observable behaviour*. An example of this would be the use of an attribute to cache the result of a computationally-expensive calculation.

In order for the compiler to permit this, such cached attributes must be declared **mutable**, and modified via the use of **const_cast**. See [18] (or any worthwhile C++ textbook) for more details.

4.2.7 Don't provide unnecessary default constructors

Rationale: The purpose of a constructor is to leave an object in a usable state. Some classes cannot be left in a usable state by a default constructor, therefore they should not have one.

Exceptions: If a class does not have a default constructor it cannot be used in a C-style array allocation. Since C-style arrays are error-prone and should not be used (see the subsequent guideline on this), this is not a major problem.

Exceptions: If a third-party library requires a class to have a default constructor. If it is absolutely necessary to use the class with such a library, it will have to have a default constructor. It should be ensured that the class has some kind of `is_properly_constructed()` query method that is used as a precondition on every function (except the one that performs the post-construction initialisation).

References: [16], 'Item 4'.

4.2.8 Prefer explicit type-conversions

Rationale: Implicit conversions between types can cause a nasty surprise for the client programmer, as the series of function calls that actually get made in an expression may be very different than those intended. Some of the unpleasant things that can happen are detailed in [5].

In general, when providing (or allowing the compiler to provide) a conversion operator between types $T1$ and $T2$, ask yourself "do I want the compiler to *silently* create a T2 from a T1?" In many cases, the answer will be "no".

Note that *constructors* can also act as conversion operators - they permit the compiler to convert one data type to another. Consider the following:

```
class array
{
public:
    array( int size );
    // ...
};
```

In this case, the compiler is free to silently create an `array` from an **int**, which may lead to unpredictable results:

```
void f( const array& );
//
f( 7 );
```

The compiler will call the array constructor to construct an array from the constant value 7, then call $f()$ with this temporary array. This is almost certainly not what is required since it was probably an error to call $f()$ with the value 7.

The solution is to declare the array constructor to be **explicit** :

```
class array
{
public:
    explicit array( int size );
    // ...
};
```

The compiler will now refuse to compile the above code unless the client makes it explicit that constructing an array is what is required:

```
array a( 7 );
f( a );
```

Exceptions: If having the compiler to silently create a $T2$ object from a $T1$ object makes sense.

4.2.9 Minimize use of pointer and reference casts

Rationale: Valid reasons for using pointer/ reference casts are rare.

Casting a pointer/ reference to a subclass class into a pointer/ reference to a *public* superclass class is perfectly legal [3].

However, one should **never** cast a pointer/ reference from a subclass into a pointer/ reference to a *private or protected superclass*.

The practice of *unchecked downcasting* a pointer/ reference to a superclass into a pointer/ reference to a subclass class is also unwise. If you feel that you need to perform such an operation, then use **dynamic_cast** or (preferably) solicit design advice.

Solutions to the common situations in which downcasting is generally perceived as necessary are outlined in [7], p153.

Note also that casting a pointer-to-member-fn to a pointer-to-fn is *undefined behaviour*.

4.2.10 Minimize use of default parameters

Rationale: Default parameters cause code maintenance difficulties (see [7] for more details).

In particular:
Do not change a default parameter in code that has existing clients.
Do not use a default parameter in a virtual function.

[3] In fact, no cast is required, since a subclass IS-A superclass, and therefore any reference to a superclass may actually rightfully point to an instance of subclass

4.3 Good Practice

4.3.1 Don't use manifest constants

Rationale: Scattering 'magic numbers' or literal strings throughout code is poor programming practice. Prefer enumerations or class-scope static constants instead (but beware of the 'static-order initialization fiasco', described above).

4.3.2 Minimize the number of macros you write

Rationale: The problem with the macro pre-processor is that it operates on a different level from the compiler — it is concerned with characters and files, rather than types and scopes.

Historically **#define** has been used to both to introduce constants and as a means of eliminating function-call overhead by introducing hardwired variables into the current function scope.

These 'C-style' practices have superior C++ equivalents:

- The keyword **const** can now be used to create type-safe, scoped constants.
- Functions can be declared **inline**, achieving the same efficiency as macros but with compiler support for scope and type-checking.

Exceptions:

1. Debugging with __FILE__ and __LINE__ can only be achieved with **#define**.
2. Conditional compilation (in conjunction with **#ifdef**). This allows us to include assertion code in the debug version and remove it completely for the release version. There is no easy way to achieve this without **#define**.

4.3.3 Keep functions short

Rationale: The longer a function is, the more difficult it is to understand and maintain. Also, the longer it is the less likely it is to have a single simply-defined purpose.

One way of determining if a function is too complex is to use *McCabe's complexity metric*:

Start with 1 for the straight path through the routine. Add 1 for each of **if**, **while**, repeat, **for**, &&, || etc. Add 1 for each **case** in a **switch** statement. If the **switch** statement doesnt have a **default** case, add 1 more. Total up the score and mark it against the following:

0-5 The routine is probably fine.

6-10 Start to think about ways to simplify the routine.

10+ Factor the routine into subroutines.

[13] has more information on this kind of complexity management.

Another measure is the number of lines in a function. This is more contentious, but an upper limit of 100 lines has entered software engineering folklore (though perhaps this was initially suggested by FORTRAN programmers). The author prefers a maximum of 30 lines, since well-designed OO programs do not generally require lengthy method implementations.

Neither of these measures is absolute or foolproof, however they both provide a useful guide.

Exceptions: Functions that consist purely of one big switch statement where each case of the switch is simply making a function call are OK. Although it violates the number of lines rule and McCabe's complexity metric, its purpose and simple, regular structure are easy to understand. (Note that for this to be an allowable exception each case should really just involve a function call).

4.3.4 Don't use C-style arrays

Rationale: C-style arrays are error prone [7].

The std :: vector class is a much safer way alternative. There is no need to worry about having allocated enough space, and std :: vector can be resized dynamically as required.

Exceptions: When interfacing with third-party code.

4.3.5 Prefer not to use raw pointers

Rationale: Raw pointers (e.g. **char** $*$, **int** $*$, my_class $*$) are error prone and are nowadays considered rather impolite. This is particularly true in an environment in which exceptions may be thrown, since they may then be the source of memory leaks.

Consider instead the appropriateness of std :: auto_ptr , or one of the BOOST smart pointer classes [3].

4.3.6 Prefer the standard string class to char *

Rationale: The std :: string class is a much safer way of dealing with character strings than arrays or pointers are. There is no need to worry about having allocated enough space or adding a zero terminator.

Exceptions: For compatibility with third-party code or C library functions. Even in these cases, it is rarely necessary to drop the string class altogether - the c_str () method returns a **const char** $*$ corresponding to the string data.

4.3.7 Minimise use of casts

Rationale: Casts are a way of subverting the type-checking system and prevent the compiler from flagging type-compatibility errors.

In general, if you find that you need to cast something, you should first check that your design is valid, then make certain that the cast is valid, preferably adding some assertion which checks that the value which is the source of the cast is representable as the cast destination.

4.3.8 Prefer standard C++ casts to C-style casts

Rationale: Use of the C-style cast is now deprecated, in favour of the standard C++ casts:

```
const_cast< T >
static_cast< T >
reinterpret_cast< T >
dynamic_cast< T >
```

The guideline 'Minimise use of casts' still applies. In particular, reconsider your design before resorting to **reinterpret_cast** or **dynamic_cast**.

See [18] for more details.

4.3.9 Prefer references to pointers

Rationale: References support stronger compile-time checking than pointers. Pointers may be NULL or otherwise be initialised to garbage values by default. The compiler requires that references be initialised at the point of construction.

Further, unless the programmer carelessly initialises a reference by dereferencing an uninitalized, NULL or dangling pointer, references will always refer to a real object.

The basic rule is "use a reference when you can, use a pointer when you have to" [7].

4.3.10 Don't override non-virtual functions

Rationale: It is possible to override non-virtual functions if the overriding function in the subclass has the same signature as the function being overridden.

The appropriate paradigm for redefining behaviour in C++ is the use of *virtual functions*. Attempting to achieve redefine behaviour by overriding a non-virtual function is incorrectly conceived and misleading from a maintenance perspective.

4.3.11 Use operator overloading with care

Rationale: Unless one takes care to mimic the signature and semantics of the built-in operators, then overloaded versions will not behave as expected, causing potentially nasty surprises for the client programmer.

In general, if an object of class $T1$ needs to be converted to an object of class $T2$ then $T2$ should provide a T2(**const** T1&) constructor, instead of T1::**operator** T2(), or T1::**operator const** T2&() etc.

Binary operators (e.g. **operator** +) should generally be implemented in terms of their op= forms (e.g. **operator** +=). Be aware that binary operators require an object to be created as a temporary and then passed back on the stack, so it is not desirable to implement them for large objects.

The full complement of arithmetic operators (not forgetting unary minus) should generally be overloaded for classes which model arithmetic types (while still taking the above recommendation about the size of temporaries into account).

Traditional C semantics should preferably be maintained between groups of related operators, such as the equivalence between x += 1 and ++x, (&p)->x and p.x etc.

As discussed in 'Prefer explicit type conversions', be aware that the constructor T1(**const** T2&) is effectively a conversion operator for the purpose of compile-time ambiguities.

References: [6] discuss the subtleties of conversion operators in detail.

4.3.12 Use a standard form for accessor methods

Rationale: An accessor method for a class is one that has (or more accurately, *looks* as if it has) access to internal data of the class, either to 'get' or 'set' the data.

A popular naming convention for access methods is based on the name of the attribute (but without the underscore suffix). e.g.

```
class some_class
{
    int value_;
public:

    // ...
    int get_value() const { return value_; }

    void set_value( int new_value )
    {
        value_ = new_value;
    }
};
```

The following points are emphasised:

- The existence of an attribute should not necessarily imply the existence of the corresponding 'get' method.
- The existence of a 'get' method should not necessarily imply the existence of the corresponding 'set' method.
- Even if these methods are provided, they do not necessarily have to be **public** — in cases where the attributes only need to be accessible to subclasses, then **protected** accessor methods will suffice.
- The existence of what stylistically appears to be an accessor method does not imply that the class actually has that object as a data member — in actual fact, the class may be calculating the data via some function, or obtaining it from some remote source.

4.3.13 Declare variables near first use

Rationale: Keeping the scope of a variable as small as possible minimises the potential for misuse.

It is helpful to only declare variables in the scope in which they are needed. Unlike C, C++ allows a variable to be declared anywhere where a statement could be placed. This can be used to keep the declaration of a variable close to where it is used, rather than further away at the top of the function.

4.3.14 Prefer for loops to while loops

Rationale: Although **for** and **while** loops are conceptually equivalent, unlike a **while** loop, **for** has placeholders for the initialisation, termination and increment steps.

These placeholders serve as a convenient reminder to actually implement these steps – if any step is missing, then it is obvious that either something is incorrect or else that the loop is written in an 'unorthodox' fashion, something that is more difficult to determine with **while**.

4.4 Niceities

4.4.1 Prefer to avoid boolean function parameters

Rationale: The values **true** or **false** in a function call tells you nothing about what the value actually means. A better option is to declare an enumerated type and pass that instead.

```
screen.display_box( 10, 10, 300, 200, blue, true );
// is far less informative than:
enum box_style { solid_style, outline_style };
screen.display_box( 10, 10, 300, 200, blue, solid_style );
```

Should there be a need for another type of style in future, it is much easier to extend the enumerated type. Similar arguments apply to boolean return values.

Exceptions: It's OK to pass a boolean as the only argument to a function if the function name makes it obvious what it means. e.g.:

```
error_log.stream_errors_to_console( true );
// ^ OK - meaning is obvious
```

4.4.2 Prefer prefix operators

Rationale: Where there is no algorithmic reason to chose between prefix (++i, −−i) and postfix (i++, i−−) versions of these operators (i.e. the value of i prior to increment/ decrement is not needed), the prefix version should be preferred, since it is more efficient.

While the efficiency gain is probably minimal for integer types and pointers, there might be a considerable gain for standard library iterators such as std :: set< T >:: iterator .

4.4.3 Minimise vendor-specific dependencies

Rationale: Third-party standards and APIs are not within your control — they are subject to change without notice and may contain errors you are unable to correct.

Standard C and C++ library functionality should be preferred above third-party libraries.

In particular, vendor-specific language extensions (e.g. **#pragma** and extra keywords) should be avoided where possible.

4.4.4 Use forward slashes in pathnames

Rationale: The backslash character ('\') is actually the escape character for C/ C++ strings, so e.g. the string "myproject\test.hpp" will be interpreted as by the compiler as "myproject[tab]est.hpp".

Although this problem can be solved by using a double backslash ('\\') it is easy to forget to do this.

Example:

```
#include "sound/samples.hpp"
const std::string sound_file = "data/sound/beep.wav";
```

4.4.5 Prefer a single point of exit

Rationale: Having one exit point from each function or block is a well-known technique designed to make understanding loops and functions easier.

It also makes it easy to add pre- and post- conditions.

Exceptions: When single exit requires a complicated collection of flags and conditional code. In such a case, first consider maintaining single exit by factoring code out into subroutines.

4.4.6 Read or write attributes individually

When reading or writing binary data, don't stream entire classes/ structs to or from storage as a composite binary object.

Rationale: The alignment of class/struct members is compiler and memory-model dependant, so class/struct members should be streamed individually, e.g.

```
struct some_class { int i; float f; double d; } ifd;

std::ostream out1( "someclass-nonportable.dat",
    std::ios::binary );
out1.write( &ifd, sizeof( ifd ) );
// ^ WARNING: this is non-portable

std::ostream out2( "someclass-portable.dat",
    std::ios::binary );
out2.write( &ifd.i, sizeof( int ) );
out2.write( &ifd.f, sizeof( float ) );
out2.write( &ifd.d, sizeof( double ) );
```

Similarly for read. Note that this approach is still insufficient to yield cross-platform portability between processors that have different byte orderings within words and long words.

Chapter 5

Design by Contract

Applying Design by Contract is one of the most effective things that can be done to improve software quality.

5.1 Implementing Design by Contract

The intent of by Design by Contract is to completely specify the *observable behaviour* of functions and objects by means of predicates known as preconditions, postconditions and invariants.

Consider the specification of a function for calculating the square root of a floating point value:

```
double square_root( double input_value );
// REQUIRE( input_value >= 0.0 );
// ENSURE( result * result == input_value );
```

The REQUIRE and ENSURE statements are known as *preconditions* and *postconditions* respectively. The preconditions of a function make explicit the range of acceptable input values and the postconditions and post-conditions give the constraints on the return value that the user of this function (the client programmer) can expect to get back.

A sketch of the corresponding implementation looks like this:

```
double square_root( double input_value )
{
    if( input_value < 0.0 )
        throw std::invalid_argument();

    // do the hard part...
    double result = some_newton_raphson_squre_root_implementation( input_value );

    assert( result * result == input_value );
    return result;
}
```

Recall that the macro `assert ( expression )` is designed to terminate the program if `expression` does not evaluate as true. The writer of the function has now established a contract with the user (or client) of the function:

> If you promise to give me a legal input value (i.e. `input_value >= 0.0`), then I promise to give you back the square root of the input value i.e. `result * result == input_value` [1].

[1] For example purposes, we won't concern ourselves with dealing with floating-point rounding errors.

Every function should have such associated pre- and post- conditions as can readily be asserted.

This approach means that invalid parameter values are rejected at the beginning of a function and eliminates any further need for error recovery code within the function itself. It is often helpful to specify contracts prior to implementing classes and functions, since the specification serves to clarify the intended purpose of the implementation.

5.2 Class invariants

A class invariant is something that should always be true about a object of that class. To be more specific, an invariant is a sequence of statements about the state of a class, chosen so that they must hold true for a valid object of that type on exit from all functions that change the state of that class.

By way of example, consider the contract for a simple date class:

```
class date
{
public:
    typedef int day_of_month;

    enum month
    {
        jan, feb, mar, apr, may, jun,
        jul, aug, sep, oct, nov, dec
    };

    typedef int year;

private:
    day_of_month    day_of_month_;
    month           month_;
    year            year_;

public:

    date( day_of_month d, month m, year y );
    // REQUIRE( is_valid_date( d, m, y ) );
    // ENSURE( get_day_of_month() == d );
    // ENSURE( get_month() == m );
    // ENSURE( get_year() == y );

    //////////////////////////////////////

    day_of_month
    get_day_of_month() const { return day_of_month_; }
    month   get_month() const { return month_; }
    year    get_year() const { return year_; }

    //////////////////////////////////////

    bool is_leap_year() const;

    int get_day_of_year() const;
    // ENSURE( result >= 0 );
    // ENSURE( implies( !is_leap_year(), result <= 365 ) );
    // ENSURE( implies( is_leap_year(), result <= 366 ) );

    //////////////////////////////////////
```

```
    static bool is_valid_date( day_of_month, month, year );

    bool invariant() const
    {
        return is_valid_date( get_day_of_month(), get_month(), get_year() ) );
    }
};
```

The implementation for the date constructor is then:

```
date::date( day_of_month d, month m, year y )
: day_of_month_( d ),
  month_( m ),
  year_( y )
{
    if( !is_valid_date( d, m, y ) )
        throw std::invalid_argument( DIAG_WHERE );

    assert( get_day_of_month() == d );
    assert( get_month() == m );
    assert( get_year() == y );

    assert( invariant() );
}
```

Note also the use of the helper predicate implies(a, b) with this truth table:

a	b	implies(a, b)
false	**false**	**true**
true	**false**	**false**
false	**true**	**true**
true	**true**	**true**

This may be implemented as:

```
bool implies( bool a, bool b )
{
    return a ? b : true;
}
```

5.3 Design by Contract in Practice

As we saw with the constructor for **class** date, we implement REQUIRE by throwing exceptions and ENSURE with the assert (expression) statement.

Recall that C and C++ files can be compiled in either *debug* and *release* mode. Debug mode is the default. Release mode and is enabled by defining the preprocessor macro NDEBUG (whether using a **#define** or via the compiler command-line).

The debug version of a C++ program differs from the release version in that assert expressions are evaluated. If the expression evaluates as **false**, the program will be terminated.

Some implementation guidelines for design by contract:

- **Habitually use assertions for logic errors.**
  Assertions should be used to trap situations that should only arise if there is a flaw in implementer (as opposed to client) program logic and for which the remedy is bug identification, code modification and recompilation.

- **Don't assert *runtime errors*.**
  Assertions are not valid error-handling in situations that arise at runtime, such as 'file not found', 'invalid user input', 'out of memory' etc. Such conditions should never be the subject of an assertion, and should instead be handled as gracefully as possible, whether by an 'recovery and retry' strategy at the point at which the error is first encountered or by throwing an exception.

- **Preconditions should be client-testable.**
  A function's preconditions should be at least as visible as the function: prefer to state preconditions, post-conditions and invariants in terms of query functions of the class, rather than attributes. This allows clients of the function to ensure that they are passing valid parameters. The preconditions and postconditions of **class** date all follow this practice - in particular, the public static method is_valid_date () method permits easy client validation of parameters.

- **Don't assert (false).**
  You should not usually assert (**false**) — include the test being asserted in the assertion itself:

```
// dont do this:
if( ptr == NULL )
    assert( false );

// do this:
assert( ptr != NULL );
```

  The exception to this guideline is the use of assert (**false**) to denote 'unreachable code', e.g. in the **default** case in a **switch** statement or in a trailing **else** clause in a chain of **if** ... **else** statements.

Chapter 6

Optimization

Why do you rob banks? Because thats where the money is.

Sutton's Law

Premature optimization is the root of all evil

Donald Knuth

This is not a lengthy section, since our primary concern is that programs work at all rather than that they might or might not work very quickly.

It is sensible to be predominently concerned with high-level optimisations than low-level optimisations. High-level optimisations usually involve choosing more efficient algorithms or data structures - using a quicksort algorithm rather than a bubblesort is an example of this. Low-level optimisations involve counting assembler cycles or finding out whether `i <<= 2` is faster than `i *= 4`. Modern compilers extremely good at these low level optimisations and can be expected to produce the same code for both cases.

If your code is really not running fast enough, make sure that you profile it before starting to optimize so that you are optimising the right bits of it.

Appendix A

Example copy-assignment implementation

The following generalised implementation of copy-assignment is provided for a class my_class in the 'implementor must copy' category. my_class manages a dynamically-allocated array of integers and additionally inherits from superclasses some_superclass1 and some_superclass2, both of which we will assume have a valid implementation of copy-assignment.

When a my_class object is copied, we must therefore ensure that the corresponding copy-assignment for the array and superclasses takes place as desired:

```
class my_class
: public some_superclass1, some_superclass2
{
    int *       array_;
    std::size_t length_;
public:

    explicit my_class( std::size_t length )
    : length_( length ),
      array_( new int [ length ] )
    {
    }

    my_class( const my_class& other )
    : some_superclass1( other ),        // copy superclass 1
      some_superclass2( other ),        // copy superclass 2
      length_( other.length_ ),     // memberwise copy
      array_( new int [ other.length_ ] )   // allocate data
    {
        // copy data
        std::memcpy( array_, other.array_,
            other.length_ * sizeof( int ) );
    }

    ~my_class() { delete [] array_; }

    // /////////////////////////////////

    my_class& operator =( const my_class& rhs )
    {
        if( this != &rhs )
        // ^ protect against 'self-assignment'
        {
            some_superclass1::operator =( rhs );
```

```
            // ^ assign superclass 1
            some_superclass2::operator =( rhs );
            // ^ assign superclass 2

            // memberwise assignment of attribute
            length_  = rhs.length_;

            // perform assignment of data pointed to...
            int * new_array = new int [ rhs.length_ ];
            std::memcpy( new_array, rhs.array_,
                rhs.length_ * sizeof( int ) );
            delete [] array_;
            array_ = new_array;
        }

        return *this;
    }
};
```

References

[1] Beck, Kent : *Extreme Programming Explained: Embrace Change*, Addison Wesley Professional, 2004.

[2] Booch, Grady et al. : *Object-oriented analysis and design with applications*, Addison-Wesley Professional, 2007.

[3] Dawes, Beman and Klarer, Robert and Abrahams, Dave et al. : *Boost - free peer-reviewed portable C++ source libraries*, Available at www.boost.org.

[4] ISO/IEC Standards Committee : *Working Draft, Standard for Programming Language C++*, Available from `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf`.

[5] Cargill, Tom : *C++ Programming Style* , Addison Wesley Longman, Redwood City, CA, 1992.

[6] Carroll, Martin D. and Ellis, Margaret A. : *Designing and Coding Reusable C++*, Addison-Wesley Longman, Boston, MA, 1995.

[7] Cline, Marshall P. et al. : *C++ FAQs*, Addison-Wesley Longman, Boston, MA. 1998.

[8] Czarnecki, Krzysztof and Eisenecker, Ulrich : *Generative Programming: Methods, Tools, and Applications* , Addison-Wesley, 2000.

[9] Fowler, Martin: *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, , Addison-Wesley Longman, Boston, MA. 2003.

[10] Gamma, Erich et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[11] Kernighan, Brian W. and Ritchie, Dennis M. : *The C Programming Language*, Prentice Hall, 1988.

[12] Kernighan, Brian W. and Pike, Robert : *The Practice of Programming*, Addison-Wesley Professional, 1999.

[13] McConnell, Steve : *Code Complete, Second Edition*, Microsoft Press, Redmond, WA, 2004.

[14] Meyer, Bertrand : *Object-Oriented Software Construction*, Prentice Hall, 1997.

[15] Meyers, Scott : *Effective C++ (3rd Edition)*, Addison-Wesley, 2005.

[16] Meyers, Scott : *Effective C++ (3rd Edition)*, Addison-Wesley, 1995.

[17] Meyers, Scott : *Effective STL: : 50 specific ways to improve your use of the standard template library*, Addison-Wesley, 2001.

[18] Stroustrup, Bjarne : *The C++ Programming Language*, Addison-Wesley, 2000.

[19] Sutter, Herb and Alexandrescu, Andrei : *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*, Addison-Wesley, 2004.