THE PROGRAMMING RESEARCH GROUP

HIGH-INTEGRITY C++ CODING STANDARD MANUAL

VERSION 2.2

FOR: GENERAL ISSUE

TITLE: HIGH INTEGRITY C++ CODING STANDARD MANUAL - VERSION 2.2

ISSUED: MAY 2004 © THE PROGRAMMING RESEARCH GROUP

Release History

Issue	Date
1.0	3 October 2003
2.0	20 October 2003
2.1	24 October 2003
2.2	18 May 2004

Table of Contents

1 Introduction

High quality code is portable, readable, clear and unambiguous. This document defines a set of rules for the production of high quality C++ code. An explanation is provided for each rule. Each rule shall be enforced unless a formal deviation is recorded. Note that Rule 2.1 outlines the process for deviation where this is deemed necessary. The guiding principles of this standard are maintenance, portability, readability and safety. This standard adopts the view that restrictions should be placed on the ISO C++ language standard[1] in order to limit the flexibility it allows. This approach has the effect of minimising problems created either by compiler diversity, different programming styles, or dangerous/confusing aspects of the language. Different compilers may implement only a subset of the ISO C++ standard or interpret its meaning in a subtly different way that can lead to porting and semantic errors. Without applying good standards, programmers may write code that is prone to bugs and/or difficult for someone else to pick up and maintain.

1.1 Typographical Conventions

Throughout this document, a rule is formatted using the following structure.

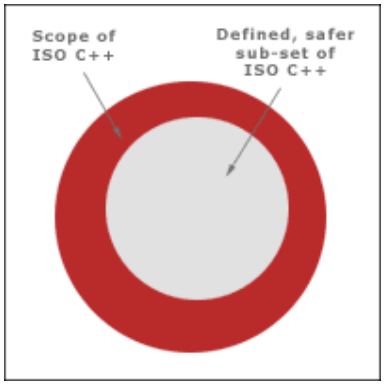
Rule	This statement describes a rule for C++. Adherence is mandatory.
Guideline	This statement describes a guideline for C++. Adherence is recommended.
Justification	This paragraph explains the rationale behind the rule or guideline.
Exception	This paragraph explains cases where the rule or guideline does not apply.
Exclusive with	This section lists references to rules or guidelines that should be disabled if this rule or guideline is selected.
See also	This section lists references to rules or guidelines that are relevant to the current rule or guideline.
Reference	This section lists sources of relevant material.
'code'	C++ keywords and code items are shown in single quotes in the text.

1.2 Escalation policy

This standard aims to enforce current best practice in C++ development by applying semantic and stylistic recommendations, including controlling the use of language features of C++ which can lead to misunderstanding and/or errors. In each case a justification is presented as to why the restriction is being applied. However, in view of the fact that research into usage of languages in general and C++ in particular is ongoing, this standard will be reviewed and updated from time to time to reflect current best practice in developing reliable C++ code.

[1] International Standard ISO/IEC 14882 First Edition 1998-09-01

1.3 Base Standard and Policy



1.3.1 ISO Standard C++

The Base Standard for this document is the ISO/IEC C++ Standard 14882 with no extensions allowed and further restrictions as detailed in the rules.

1.3.2 Statically detectable restrictions

This Standard requires that the use of the C++ language shall be further restricted, so that no reliance on statically detectable[1] undefined or unspecified behaviour listed in this standard is allowed. Coding practice that results in undefined behaviour is dangerous and must always be avoided. Where undefined behaviour can be identified statically, coding rules limit the potential for introducing it. The rules also prohibit practice which, although well defined, is known to cause problems.

1.3.3 Allowable environments

In general, only ISO C++ compliant compilers should be used. However, at the current time, compilers do not achieve full ISO compliance, and it may be some time before the mainstream compilers become completely ISO C++ compliant. Hence only the features of a compiler that are proven to be ISO C++ compliant should be used. Compiler validation is a useful way to gauge the compliance of a compiler with the ISO C++ standard.

1.3.4 Rule subsets

Some of the rules in this standard are mutually exclusive, hence only a subset of rules should be selected from this standard.

1.3.5 Examples

This standard contains many example code fragments which are designed to illustrate the meaning of the rules. For brevity some of the example code does not conform to all best practices, e.g. unless the rule or guideline relates explicitly to exception specifications, the example code may not be exception safe.

[1] That is, at compile time

1.4 Basis of requirements

Requirements in this standard express:

(a) restrictions on the use of language constructs or library functions that are not completely defined by the ISO C++ Standard.

(b) restrictions on language constructs that permit varied compiler interpretation.

(c) restrictions on the use of language constructs or library functions that are known to be frequently misunderstood or misused by programmers thereby leading to errors.

(d) restrictions on the use of language constructs that inhibit the capabilities of static analysis.

The basis of these requirements is that by meeting them it is possible to avoid known problems and thereby reduce the incidence of errors.

1.5 Inconsistencies across file boundaries

The rules in this standard refer directly to inconsistencies which can arise within a single translation unit, i.e. a file which is being checked or compiled. In C++, owing to its independent compilation model, many such inconsistencies arise across file boundaries, (this standard includes inter translation unit rules).

1.6 Policy on non-C++ code and non-standard pre-processors

The embedding of code, written in languages other than C++, within C++ code is forbidden unless accompanied by a written justification for its use. The generally poor definition of interfaces to embedded, non C++ code, can lead to problems. Any necessary use should therefore be localised as much as possible. Embedded code for pre-processors other than the Standard C++ pre-processor shall be similarly restricted.

1.7 Deviations

Notwithstanding the requirements of this standard, it may be necessary, and in some cases desirable to tolerate limited non-compliance. Such non-compliance shall, without exception, be the subject of a written deviation supported by a written justification.

1.8 Compliance matrices for C++ development

Good practice advocates using a compliance matrix to accompany all C++ development projects. A compliance matrix shall detail the following information about the project.

- Description of development.
- Compiler release(s) for development and whether this compiler is validated or not. Compiler conformance to ISO or pre-ISO standards shall be stated.
- Any compiler switches used (compilers must be validated under these conditions).
- Hardware type for which the development is intended.
- Operating system for the development, including version number and any patches applied.
- Third party libraries used for the development, including version numbers.
- Maximum number of characters assumed for unique identification of coded identifiers.
- Any software metric limits in force.

- List of compiler flaws if available (to ensure portability).
- All dependence on implementation defined behaviour.
- Conformance to Annex B (Implementation Quantities) of the ISO Standard.

Such a matrix should be laid out as a simple table.

For example:-

Description of Development	Edge-detection algorithm library
Compiler release	GNU compiler version x.x.x.x
Compiler validation status	Yes; ISO
Compiler switches	...
...	

2 General

Rule 2.1 Thoroughly document in the code any deviation from a standard rule.

Justification This standard addresses most situations, however a specific situation may require deviation from the standard introducing unexpected anomalies in system behaviour or affecting other system qualities. Since there are usually several ways to address such requirements, it is important to consider the benefits and drawbacks of each approach. Alternatives should be documented so that approaches are not taken, during maintenance, that have already been discarded. All the consequences of the choice should be documented so that correct assumptions can be made in maintenance.

Guideline 2.2 Specify in your compiler configuration that plain 'char' is implemented as 'unsigned char'.

Justification Support 8-bit ASCII for internationalisation. The size and sign of char is implementation-defined. If the range of type char corresponds to 7-bit ASCII, and 8-bit characters are used, unpredictable behaviour may result. Otherwise prefer to use wchar_t type.

See also Rule 8.4.5

3 Class

3.1 General

Rule 3.1.1 Organise 'class' definitions by access level, in the following order : 'public', 'protected', 'private'.
(QACPP 2108, 2109, 2191, 2192, 2195)

Justification Order by decreasing scope of audience. Client program designers need to know public members; designers of potential subclasses need to know about protected members; and only implementors of the class need to know about private members and friends.

```
class C           // correct access order
{
public:
// ...
protected:
// ...
private:
// ...
};
```

Reference Industrial Strength C++ A.12, A.13;

Rule 3.1.2 Define class type variables using direct initialisation rather than copy initialisation.
(QACPP 5012)

Justification In constructing both objects 'a1' and 'b1', a temporary String("Hello") is constructed first, which is used to copy construct the objects. On the other hand, for 'c1' only a single constructor is invoked. Note, some compilers may be able to optimise construction of 'a1' and 'b1' to be the same as 'c1'; however, conversion rules would still apply, e.g. at most one user-defined conversion.

```
String a1 = "Hello";             // avoid
String b1 = String( "Hello" );   // avoid

String c1( "Hello" );            // prefer
```

See also Rule 8.4.4

Rule 3.1.3 Declare or define a copy constructor, a copy assignment operator and a destructor for classes which manage resources.
(QACPP 2110, 2112)

Justification The compiler provided functions that perform copying (i.e. copy constructor and copy assignment operator), perform bitwise or shallow copy. This will result in copied objects pointing to the same resource (after copy) and both will share the resource when a duplicated resource may have been necessary. On destruction each object will free its copy of the resource, which may lead to the same resource being freed more than once.

The destructor should be declared because the implicit destructor will not release resources (normally dynamically allocated memory).

Explicitly declare your intentions when writing copy constructors and assignment operators. Make it clear when you wish to use a shallow copy in your assignment operator by explicitly coding the function even when your compiler generates the same code by default.

When a copy constructor and a copy assignment operator for a class with pointer types in its member data are considered impractical, declare the functions private, but do not define them hence preventing clients from calling them and preventing the compiler from generating them.

See also Rule 3.1.13

Reference Effective C++ Item 11;Industrial Strength C++ 5.11;

Rule 3.1.4 Use an atomic, non-throwing swap operation to implement the copy-assignment operator ('operator=')

(QACPP 2074, 2081, 2082, 4620, 4621)

Justification Herb Sutter recommends implementing the copy-assignment operator with a non-throwing Swap() member and the copy constructor. This has the advantage that copy assignment is expressed in terms of copy construction, does not slice objects, handles self-assignment and is exception safe. It relies on the Swap() member being guaranteed not to 'throw' and to swap the object data as an atomic operation.

```
class A
{
public:
   A& operator=( const A& rhs )
   {
      A temp( rhs );
      Swap( temp );    // non-throwing
      return *this;
   }

private:
   void Swap( A& rhs ) throw ();
};
```

Exclusive with Rule 3.1.5

Reference Exceptional C++ Item 41;

Rule 3.1.5 Ensure copy assignment is implemented correctly in terms of self assignment, inheritance, resource management and behaves consistently with the built in assignment operator.

(QACPP 2074, 2081, 2082, 4072, 4073, 4620, 4621)

Justification Scott Meyers recommends the following:

```
A& A::operator=( const A& rhs )
{
   if ( this != &rhs )                    // 1.
   {
      Release_All_Resources_Of( this );   // 2.
      Base::operator=( rhs );             // 3.
      Copy_Member_Data( this, rhs );      // 4.
   }
   return *this;                          // 5.
}
```

1. Prevent assignment to self. Assignment to self is inefficient and potentially dangerous, since resources will be released (step 2) before assignment (step 4).

2. Release all resources owned by the current object ('this'). Apply delete to any pointers to data owned and referenced solely by the current object. (This includes all pointers which point to space allocated using the new operator, unless a reference counting idiom is used.) Owned resources should be released to prevent problems such as memory leaks.

3. If the function is a member of a derived class, invoke operator=() for the base class. Invoking the assignment operator of the base class, instead of setting base class attribute values, reduces coupling between classes in the same inheritance hierarchy, improving maintainability.

4. Copy all member data in the argument object according to the copy semantics for the class. Ensure all data members are assigned. If a pointer value is simply copied and the copy semantics do not support multiple references to an object through a mechanism such as reference counting, a subsequent delete of one of the objects will cause the pointer in the other object to be invalid. This will cause a problem either through an access attempt via the invalid pointer, or through an attempt to delete the pointed-to object in the destructor of the containing object.

5. Return *this as a reference. Returning *this provides behaviour consistent with the built-in assignment operators.

When maintenance results in the addition of a data item to a class, all assignment operators must be updated.

Exclusive with Rule 3.1.4

Reference Effective C++ Items 15, 16, 17;Industrial Strength C++ 5.12, 7.7;

Rule 3.1.6 Only inline simple and non virtual functions, such as one line getters and setters.
(QACPP 2131, 4120, 4121)

Justification Inline functions do not necessarily improve performance and they can have a negative effect. Inappropriate use will lead to longer compilation times, slower runtime performance and larger binaries.

Virtual functions cannot be inlined due to polymorphism. A compiler cannot know which instance of a virtual function will be called at compile time so the inline keyword will be ignored.

Constructors and destructors should never be inlined as they implicitly call other constructors and destructors and are therefore not simple functions.

```
class A
{
public:
   int getVal() { return m_val; }          // ok is a getter
   void setVal( int val ) { m_val = val; } // ok is a setter
   virtual ~A() {}                         // ok destructor must be defined
   virtual void foo() {}                   // avoid function is virtual so
                                           // never inlined

private:
   int m_val;
};
```

See also Guideline 3.1.7, Guideline 8.1.2, Rule 11.8

Reference Industrial Strength C++ A.15;

Guideline 3.1.7 Do not use the 'inline' keyword for member functions, inline functions by defining them in the class body.
(QACPP 2133)

Justification The inline keyword is a hint to the compiler, its use does not mean that the function will actually be inlined. By putting the definition of a function in the class body the compiler will implicitly try to inline the function.

In order for a function to be inlined its definition must be visible when the function is called, by placing the definition inside the class body it will be available where needed.

```
class C
{
public:
   int bar() { return 1; }        // prefer
   inline int car() { return 1; } // avoid
   inline int foo();              // avoid
};

inline int C::bar()               // avoid
{
   return 1;
}
```

See also Rule 3.1.6, Guideline 8.1.2, Rule 11.8

Reference Industrial Strength C++ A.15;

Rule 3.1.8 Declare 'const' any class member function that does not modify the externally visible state of the object.

(QACPP 4211, 4214)

Justification Although the language enforces bitwise const correctness, const correctness should be thought of as logical, not bitwise.

A member function should be declared const if it is impossible for a client to determine whether the object has changed as a result of calling that function.

The 'mutable' keyword can be used to declare member data which can be modified in const functions, this should only be used where the member data does not affect the externally visible state of the object.

```
class C
{
public:
   const C& foo() { return * this; }      // should be declared const
   const int& getData() { return m_i; }   // should be declared const
   int bar() const { return m_mi; }       // ok to declare const
private:
   int m_i;
   mutable int m_mi;
};
```

Reference Effective C++ Item 21;Industrial Strength C++ 7.13;

Guideline 3.1.9 Behaviour should be implemented by only one member function in a class.

Justification If two functions implement the same behaviour, they should be implemented in terms of each other or through a common helper function. An example is a binary 'operator+', which should be implemented in terms of 'operator+='.

```
class A
{
public:
   A operator+( const A& rhs )
   {
      A temp( *this );
      temp += rhs;
      return temp;
   }
   A& operator+=( const A& rhs );
```

```
};
```

This will increase code reuse and improve maintainability.

See also Rule 11.1, Rule 12.1

Rule 3.1.10 Do not declare conversion operators to fundamental types. (QACPP 2181)

Justification Conversion operators should not be used, as implicit conversions using conversion operators can take place without the programmers knowledge.

This standard advocates the declaration of all one argument constructors as 'explicit' to avoid implicit conversion by constructor. This rule extends the requirement by disallowing type conversion by conversion operators.

```
class B;

class C
{
public:
   operator B();  // conversion operator
};
```

See also Rule 3.1.11, Rule 3.2.3

Reference More Effective C++ Item 5;Industrial Strength C++ 7.19;

Rule 3.1.11 Do not provide conversion operators for class types. (QACPP 2181)

Justification Conversion operators should not be used because implicit conversions using conversion operators can take place without the programmers knowledge. Conversion operators can lead to ambiguity if both a conversion operator and a constructor exist for that class. In most cases it is better to rely on class constructors.

```
class C;

class D
{
public:
   D( C );          // 1
};

class C
{
public:
   operator D();    // 2
};

void foo( D );

void bar()
{
   C c;
   foo( c );        // ambiguous (convert to D by 1 or 2?)
}
```

See also Rule 3.1.10

Reference Effective C++ Items 18, 27;More Effective C++ Item 5;Industrial Strength C++ 7.19;

Guideline 3.1.12 Provide an output operator ('operator<<') for ostream for all classes.

Justification Providing an output stream operator is useful for the debugging and testing of code.

Rule 3.1.13 Verify that all classes provide a minimal standard interface against a checklist comprising: a default constructor; a copy constructor; a copy assignment operator and a destructor.
(QACPP 2110, 2111, 2112, 2114, 2142, 2185, 2610, 2611, 2612, 2613, 2614, 2615, 2616, 2617, 2618, 2631, 2632, 2633)

Justification The following functions are key to making a class behave like a fundamental type and providing for easier comprehension and maintenance.

```
class X
{
   X();                       // default constructor
   X( const X& );             // copy constructor
   X& operator=( const X& ); // copy assignment operator
   ~X();                      // destructor
};
```

The compiler will provide default versions of these functions if they are not explicitly declared. The behaviour of the compiler-generated default constructor is not always appropriate because it does not initialise members that are of POD type.

The behaviour of the other compiler-generated functions is satisfactory only if a class has no pointer member variables and if each of these implicitly generated functions may have public access.

Defining these functions results in a more consistent interface and a more maintainable and extensible implementation, and carries few penalties. If a class design does not require these functions then explicitly declare them private; this will prevent the compiler generated functions from being used.

See also Rule 3.1.3

Reference Effective C++ Item 11, 33;Industrial Strength C++ 5.11, 14.2;

3.2 Constructors and Destructors

Rule 3.2.1 Ensure all constructors supply an initial value (or invoke a constructor) for each virtual base class, each non virtual base class and all non-static data members.
(QACPP 4050, 4051, 4052, 4054, 4206, 4207)

Justification Each constructor must initialise all member data items. Explicit initialisation reduces the risk of an invalid state after successful construction. All virtual base classes, direct non virtual base classes and non-static data members should be included in the initialisation list for the constructor, for many constructors this means that the body becomes an empty block.

Regardless of how explicit initialisers are specified, the order of initialisation is as follows:
1. Virtual base classes in depth and left to right order as they, or a class that derives from them, appear in the inheritance list.
2. Base classes in left to right order of inheritance list.
3. Non-static member data in order of declaration in the class definition.

```
class B {};

class VB : public virtual B {};
```

```
class C {};

class DC : public VB, public C
{
public:
   DC()
      : B(), VB(), C(), i( 1 ), c() // correct order of initialization
   {}

private:
   int i;
   C c;
};
```

See also Rule 3.2.2

Reference Effective C++ Item 12;Industrial Strength C++ 5.5;

Rule 3.2.2 Write members in an initialisation list in the order in which they are declared.
(QACPP 4053)

Justification Data members are initialised in the order in which they are specified in the class definition, not the order they appear in the initialisation list of the constructor. Similarly destructors of members are called in reverse construction order.

See also Rule 3.2.1

Reference Effective C++ Item 13;Industrial Strength C++ 5.6;

Rule 3.2.3 Declare all single argument constructors as explicit thus preventing their use as implicit type convertors.
(QACPP 2180)

Justification By making single argument constructors explicit they cannot be used accidentally in type conversions.

```
class C
{
public:
   C( const C& );               // ok copy constructor
   C();                         // ok default constructor
   C( int, int );               // ok more than one non-default argument


   explicit C( int );           // prefer
   C( double );                 // avoid
   C( float f, int i=0 );       // avoid, implicit conversion constructor
   C( int i=0, float f=0.0 );   // avoid, default constructor, but
                                // also a conversion constructor
};

void bar( C const & );

void foo()
{
   bar( 10 );              // compile error must be 'bar( C( 10 ) )'
   bar( 0.0 );             // implicit convserion to C
}
```

Exception This rule does not apply to copy constructors as they do not perform a conversion.

See also Rule 3.1.10, Rule 7.1, Rule 7.8, Guideline 10.7, Rule 11.4

Reference Industrial Strength C++ 7.18;

Guideline 3.2.4 An abstract class shall have no public constructors.

Justification Abstract classes cannot be used to declare objects, by making constructors protected it is explicit that the class can only be used from derived classes.

Rule 3.2.5 Ensure destructors release all resources owned by the object.

Justification Failure to release resources owned by the object could result in resource leaks.

Reference Industrial Strength C++ 12.8;

3.3 Inheritance

Rule 3.3.1 Use public derivation only.
(QACPP 2193, 2194)

Justification Using public derivation maintains visibility of public base members in an intuitive way. Public derivation indicates the "is-a" relationship. Private derivation indicates the "is-implemented-by" relationship, which can also be indicated by containment (that is, declaring a private member of that class type instead of inheriting from it). Containment is the preferred method for "is-implemented-by", as this leaves inheritance to mean "is-a" in all cases.

```
class A {};

class B : private A {};         // avoid private derivation

class C : protected A {};       // avoid protected derivation

class D : A {};                 // avoid implicitly private derivation

class E : public A {};          // prefer public derivation
```

Rule 3.3.2 Write a 'virtual' destructor for base classes.
(QACPP 2116)

Justification If an object will ever be destroyed through a pointer to its base class, then that base class should have a virtual destructor. If the base class destructor is not virtual, only the destructor for the base class will be invoked. In most cases, destructors should be virtual, because maintenance or reuse may add derived classes that require a virtual destructor.

```
class Base {};

class Derived : public Base
{
public:
   ~C() {}
};

void foo()
{
   Derived* d = new Derived;
   delete d;  // correctly calls derived destructor
}

void boo()
{
   Derived* d = new Derived;
   Base* b = d;
   delete b; // problem! does not call derived destructor!
}
```

See also Guideline 17.7

Reference Effective C++ Item 14;Industrial Strength C++ 10.4;

Rule 3.3.3 Avoid downcasting base class object pointers to derived class. (QACPP 3070)

Justification Use virtual functions instead. The most common reason for casting down the inheritance hierarchy is to call methods particular to a class in the hierarchy when a pointer to the base class is passed or stored. This may be better achieved by the use of virtual functions.

```
class A
{
   virtual void bar();
};

class B : public A
{
   virtual void bar();
   virtual void foo();
};

void foo()
{
   A* a = new B;
   static_cast< B* >( a )->foo(); // avoid
   a->bar();                      // prefer
}
```

Reference Effective C++ Item 39;

Rule 3.3.4 Avoid casting to a virtual base class as this is irreversible. (QACPP 3071)

Justification Do not cast a pointer up an inheritance hierarchy to a virtual base class as this pointer may not be cast back down the hierarchy.

```
class A {};

class B : public virtual A {};

A* foo()
{
   B* b = new B;
   return static_cast< A* >( b );        // casting to virtual base
}
```

Reference ISO C++ 5.2.9/5, 5.2.9/8;

Rule 3.3.5 Override all overloads of a base class virtual function. (QACPP 2120)

Justification When a virtual function is overridden then the overloads of that function in the base class are not visible from the derived class. If all overloaded functions are not brought into the derived class, by overriding them or with a using declaration, then you can get surprising results when calling member functions of that name.

```
class Base
{
public:
   virtual void foo( short );
   virtual void foo( double );
};
```

```
class Derived : public Base
{
public:
   virtual void foo( short );
   void bar()
   {
      foo( 0.1 ); // calls Derived::foo( short )!
   }
};
```

Reference Industrial Strength C++ 7.16;

Rule 3.3.6 If a virtual function in a base class is not overridden in any derived class then make it non virtual.

Justification If each derived class is using the base class implementation of the virtual function then the function probably does not need to be virtual. Making it non virtual will improve performance by reducing the cost of calling the function.

See also Rule 3.3.7, Rule 3.3.8, Rule 3.3.9, Rule 3.3.11

Rule 3.3.7 Only define virtual functions in a base class if the behaviour will always be valid default behaviour for derived classes.

Justification Virtual functions in the derived class may or may not override the base class function implementation. If the behaviour will not be appropriate for most derived classes then it probably should not be defined in the base class.

Exception Destructors must always be defined in the base class.

See also Rule 3.3.6, Rule 3.3.8, Rule3.3.9

Reference Effective C++ Item 36;

Rule 3.3.8 Declare a function pure virtual in the base class if each derived class has to provide specific behaviour.

Justification If a function is pure virtual in a base class then derived classes must define it. Use pure virtual functions and abstract classes to create abstractions that are implemented in derived classes.

See also Rule 3.3.6, Rule 3.3.7, Rule3.3.9

Rule 3.3.9 If a virtual function is overridden in each derived class with the same implementation then make it a non virtual function in the base class.

Justification If each derived class has the same implementation for a function then that function can be implemented non virtually in the base class, this improves performance, code reuse and eases maintenance.

See also Rule 3.3.6, Rule 3.3.7, Rule 3.3.8

Rule 3.3.10 Ensure that the return type of the virtual function being overridden is compatible.

Justification A virtual function must be written in the derived class with the same signature as the virtual function it overrides in the base class, except that a covariant return type is allowed. This means that the return type of the derived function can be a type derived from the base class return type. If the original return type was B* or B&, then the return type of the overriding function may be D* or D&, provided B is a public base of D.

```
class Base
{
public:
   virtual Base* clone() { return new Base( *this ); }
};

class Derived : public Base
{
public:
   virtual Derived* clone() { return new Derived( *this ); }
};

void fn( Derived* d, Base* b )
{
   Derived* p1 = d->clone();
   Derived* p2 = b->clone();    // error, downcast needed here
}
```

Reference Stroustrup;

Rule 3.3.11 Do not overload or hide inherited non-virtual functions.

(QACPP 2121)

Justification Overloading or hiding non-virtual member functions can result in unexpected behaviour as non-virtual functions are statically bound. This results in the declaration type of the pointer or reference determining the selection of member functions and not what the pointer or reference is actually pointing at.

See also Rule 3.3.6

Reference Effective C++ Item 37;Industrial Strength C++ 7.16;

Rule 3.3.12 When redeclaring and overriding functions use the same default parameter values as in other declarations.

(QACPP 2015, 2018)

Justification An overridden virtual function should have the same default values as the base class function. Default parameter values are determined by the static type of the object. This means that the default values used may not match those of the virtual function being called.

```
class Base
{
public:
   virtual void goodvFn( int a = 0 );
   virtual void badvFn( int a = 0 );
};

class Derived : public Base
{
public:
   virtual void goodvFn( int a = 0 );
   virtual void badvFn( int a = 10 );
};

void foo( Derived& obj )
{
   Base& baseObj = obj;

   // Ok - derived and base have the same default value
   //
   baseObj.goodvFn();  // calls Derived::goodvFn with a = 0
   obj.goodvFn();      // calls Derived::goodvFn with a = 0

   // Uses default value from base even though calls derived function
```

```
   //
   baseObj.badvFn();  // calls Derived::badvFn with a = 0
   obj.badvFn();      // calls Derived::badvFn with a = 10
}
```

See also Rule 12.1

Reference Effective C++ Item 38;

Rule 3.3.13 Do not invoke virtual methods of the declared class in a constructor or destructor. (QACPP 4260, 4261)

Justification Invoking virtual methods in a constructor always invokes the method for the current class, or its base, even when the constructor is invoked as part of the construction of a derived class. This also applies to virtual methods called in a destructor.

```
class B
{
public:
   B();
   virtual void func();
};

class D : public B
{
   D() : B() {}
   virtual void func();
};

B::B()
{
   func();    // B::func called not D::func
}
```

Rule 3.3.14 Declare the copy assignment operator protected in an abstract class. (QACPP 2080)

Justification By ensuring that the copy assignment operator is protected, it can only be, and should only be, called by the assignment operator of the derived class.

```
class Base
{
public:
   Base& operator=( const Base& );   // should have protected access
};

class Derived : public Base
{
public:
   Derived& operator=( const Derived& );
};

void foo()
{
   Derived obj1;
   Derived obj2;

   Base* ptr1 = &obj1;
   Base* ptr2 = &obj2;

   *ptr1 = *ptr2;                    // problem; partial assignment
}
```

Reference More Effective C++ Item 33;

Rule 3.3.15 Ensure base classes common to more than one derived class are virtual. (QACPP 2151)

Justification If a class derives non-virtually from more than one class with a common non-virtual base class, then multiple copies of that base class will be created. Virtual inheritance ensures that there is only one instance of the base class object, making calls to its member functions unambiguous.

```
class base
{
public:
   void f();

};

class derived_left: public base {};

class derived_right: public base {};

class derived: public derived_left, public derived_right {};

void test()
{
   derived d;
   d.f();           // ambiguous - derived_left::base::f()
                    //          or derived_right::base::f()
}
```

If the intent was that the call should not be ambiguous, then derived should probably inherit base using virtual inheritance from both of it's immediate base classes. For example:

```
class derived_left: public virtual base {};

class derived_right: public virtual base {};
```

Reference Industrial Strength C++ 10.5;

Rule 3.3.16 Explicitly declare polymorphic member functions virtual in a derived class. (QACPP 2132)

Justification When examining the class definition of a derived class, documentation is needed to determine which members are virtual. Specifying 'virtual' explicitly helps to document the class.

```
class A
{
public:
   virtual void f();
   virtual void operator+( A const& );
   virtual ~A();
};

class B1 : public A
{
public:
   virtual void f();                      // virtual: make explicit
   virtual void operator+( A const& );    // virtual: make explicit
   virtual ~B1();                         // virtual: make explicit
};
```

3.4 Object Oriented Design

Rule 3.4.1 Make member data private.
(QACPP 2100, 2101)

Justification By implementing class interfaces with member functions the implementor achieves precise control over how the object state can be modified and allows a class to be maintained without affecting clients. If direct access to object state is allowed through public member data then encapsulation is reduced.

```
class C
{
   int a;        // avoid (implicitly private)
public:
   int b;        // avoid
protected:
   int c;        // avoid
private:
   int d;        // prefer
};
```

Reference Effective C++ Item 20, Industrial Strength C++ 10.1;

Rule 3.4.2 Do not return non-const handles to class data from const member functions.
(QACPP 4024)

Justification Non-const handles returned from const member functions indirectly allow modification of class data. Const functions returning pointers or references to member data should return const pointers or references.

```
class A
{
public:
   int* foo() const
   {
      return m_pa;    // permits subsequent modification of private data
   }

private:
   int* m_pa;
};

void bar()
{
   const A a;
   int* pa = a.foo();
   *pa = 10;          // modifies private data in a!
};
```

Exclusive with Rule 3.4.3

Reference Effective C++ Items 21, 29;Industrial Strength C++ 7.12;

Rule 3.4.3 Do not write member functions which return non const pointers or references to data less accessible than the member function.
(QACPP 2011)

Justification Member data that is returned by a non const handle from a more accessible member function, implicitly has the access of the function and not the access it was declared with. This reduces encapsulation and increases coupling.

Member functions returning pointers or references to member data should return const pointers or references.

```
class A
{
public:
   A () : m_private_i(0) {}

   int& modifyPrivate()
   {
     return m_private_i;
   }
```

```
   int const& readPrivate()
   {
     return m_private_i;
   }

private:
   int m_private_i;
};

void bar()
{
   A a;

   // m_private_i is modified.
   // m_private_i implicitly has the same access
   // as the member function foo, i.e. public.
   //
   a.modifyPrivate() = 10; // avoid

   // Generates a compile error as value is not modifiable.
   //
   a.readPrivate() = 10; // prefer
}
```

Exclusive with Rule 3.4.2

Reference Effective C++ Items 21, 29, 30;Industrial Strength C++ 7.12;

Rule 3.4.4 Ensure friends have a legitimate basis in design, otherwise avoid. (QACPP 2107)

Justification A function or class should not be made a friend simply for programmer convenience. Friends increase coupling, complicate interfaces and reduce encapsulation.

Rule 3.4.5 When publicly deriving from a base class, the base class should be abstract. (QACPP 2153)

Justification When thinking about object design it is common practice to take the commonality of each object and define an abstraction on these features. Leaf classes that inherit from this abstraction are then concerned primarily with object creation.

```
class Abstract
{
public:
   virtual ~Abstract() = 0;
   // ...
protected:
   Abstract& operator=( const Abstract& rhs );
};

class Concrete1 : public Abstract
{
public:
   Concrete1& operator=( const Concrete1& rhs );
   // ...
};

class Concrete2 : public Abstract
{
public:
   Concrete2& operator=( const Concrete2& rhs );
   // ...
};
```

Reference More Effective C++ Item 33;

Rule 3.4.6 Write derived classes to have at most one base class which is not a pure abstract class.

Justification Inheriting from two or more base classes, that are not pure abstract classes, is rarely correct. It also exposes the derived class to multiple implementations, with the risk that subsequent changes to any of the base classes could invalidate the derived class.

A pure abstract class is one for which all members are pure virtual functions. The purpose of a pure abstract class is to define an interface that one or more concrete classes may implement. It is reasonable that a concrete class may implement more than one interface.

Guideline 3.4.7 All members of a public base class must be valid for a derived class.

Justification Public inheritance should implement the subtype relationship, in which the subtype or derived type is a specialisation of the supertype or base type.

Hence the behaviour of the sub type as determined by its member functions and (the object state) by its member variables should be entirely applicable to the supertype.

3.5 Operator Overloading

Rule 3.5.1 Avoid overloading the comma operator (','), operator AND ('&&'), and operator OR ('||'). (QACPP 2077, 2078, 2079)

Justification The behaviour that users expect from these operators is evaluation from left to right, in some cases with shortcut semantics. When an operator is overloaded function call semantics come into play, this means that the right and left hand sides are always evaluated and become parameters to a function call. The order of evaluation of the parameters to the function is unspecified and it is possible that the right hand operand is evaluated before the left.

Reference More Effective C++ Item 7;

Rule 3.5.2 Always write operations, that are normally equivalent, to be equivalent when overloaded.

Justification Users of a class expect that overloaded operators will behave in the same way as the corresponding built-in operator.

```
a += b    // should give the same result as a = a + b
a += 1    // should give the same result as ++a
```

Reference Effective C++ Item 15;

Rule 3.5.3 Ensure that overloaded binary operators have expected behaviour. (QACPP 2071, 2072, 2073, 4222)

Justification Write overloaded operators such that the behaviour is understandable based on the behaviour of the operator on fundamental types.

As far as possible when overloading built-in operators they should follow the behaviour that the user has come to expect. This promotes reuse and maintainability. This does not mean that overloaded operators should have meanings identical to that of the normal usage.

Operator + should have an additive effect (e.g. string concatenation).

Equivalence operators (==,!=) should only be used to determine object equivalence. If operator!= is defined, operator== should be defined as well.

```
class Complex
{
public:
   Complex operator+( const Complex& c );
};

// This will be very confusing:
//
Complex Complex::operator+( const Complex& c )
{
   cout << "this function does nothing close to addition";
           return *this;
}
```

Reference Effective C++ Items 21, 22, 23;

Rule 3.5.4 Make binary operators non-members to allow implicit conversions of the left hand operand.
(QACPP 2070)

Justification By making binary operators members, a conversion to the left hand side of the binary operator is not possible.

```
class complex
{
public:
   complex( float r, float i = 0 );
   complex operator+( const complex& rhs );
};

void Add()
{
   complex a( 1, 0 );
   a = a + 2;            // fine: 2 is converted to complex
   a = 2 + a;            // error: no applicable operator +
}
```

Reference Effective C++ Item 19;

Guideline 3.5.5 When overloading the subscript operator ('operator[]') implement both const and non-const versions.
(QACPP 2140, 2141)

Justification Allow the operator to be invoked on both const and non-const objects.

```
class Array
{
public:
   Array()
   {
      for ( int i = 0; i < Max_Size; ++i )
      {
         x[ i ] = i;
      }
   }

   int& operator[] ( const int a )
   {
      std::cout<< "nonconst" << std::endl;
      return x[ a ];
   }

   int operator[] ( const int a ) const
```

```
   {
      std::cout << "const" << std::endl;
      return x[ a ];
   }

private:
   enum { Max_Size = 10 };
   int x[ Max_Size ];
};

int main()
{
   Array a;
   int i = a[ 3 ];   //non-const
   a[ 3 ] = 33;      //non-const

   const Array ca;
   i = ca[ 3 ];      //const
   ca[ 3 ] = 33;     //compilation error

   return 0;
}
```

Reference Effective C++ Item 18;

4 Complexity

Rule 4.1 Do not write functions with an excessive McCabe Cyclomatic Complexity.
(QACPP 5040)

Justification The McCabe Cyclomatic Complexity is a count of the number of decision branches within a function. Complex routines are hard to maintain and test effectively. Recommended maximum in this standard is 10.

This rule will highlight complex code which should be reviewed.

Rule 4.2 Avoid functions with a high static program path count.
(QACPP 5041)

Justification Static program path count is the number of non-cyclic execution paths in a function. Functions with a high number of paths through them are difficult to test, maintain and comprehend. The static program path count should not exceed 200.

Rule 4.3 Avoid functions with many arguments.
(QACPP 5042)

Justification Functions with long lists of arguments are difficult to read, often indicate poor design, and are difficult to use and maintain. The recommended maximum in this standard is six parameters.

5 Control Flow

Rule 5.1 Follow each flow control primitive ('if', 'else', 'while', 'for', 'do' and 'switch') by a block enclosed by braces, even if the block is empty or contains only one line.
(QACPP 4013, 4014, 4016, 4060, 4061, 4062, 4063, 4064, 4065, 4066)

Justification The consistent application of braces to delimit a block makes the code clearer, more consistent, and less error prone.

See also Rule 5.11

Reference Industrial Strength C++ 4.3;

Rule 5.2 For boolean expressions ('if', 'for', 'while', 'do' and the first operand of the ternary operator '?:') involving non-boolean values, always use an explicit test of equality or non-equality.

Justification The explicit test clarifies intent, and is more precise. If a boolean expression involves an object (e.g. a database pointer or smart pointer), the implicit test will have different behaviour than an explicit test if operator==() is overloaded.

If the expression contains an assignment, the explicit test indicates that the assignment was intended.

```
int bar();

void foo()
{
   if ( bar() )        // avoid
   {}
   if ( 0 != bar() )   // prefer
   {}
}
```

Rule 5.3 Avoid conditional expressions that always have the same result.
(QACPP 3260, 4090, 4091, 4092, 4093, 4094)

Justification If a conditional expression always has the same result, there is no need for the condition.

```
void bar( unsigned int ui )
{
   // By definition ui cannot be less than zero hence
   // this expression is always false.
   //
   if ( ui < 0U )
   {
      // never reached
   }
   else
   {
      // always executed
   }
}
```

Exception It is possible to have an expression that always evaluates to the same result on a given platform but not another platform.

```
void bar( unsigned int ui )
{
   if ( ui <= 0xFFFFU )
   {}
   else
   {
      // only reached depending on platform
   }
```

```
}
```

Rule 5.4 **Follow each non-empty case statement block in a switch statement with a break statement.**
(QACPP 4011)

Justification This practice has safety advantages and encourages maintainability. If only part of the action for multiple cases is identical, place that part in a separate function.

This rule does not require each case statement to have a unique statement block. It does prohibit fall-through from one case statement block to another.

```
void foo( int i )
{
   switch ( i )
   {
   case 0:
   case 1:
      ++i;             // non-empty case statement needs break
   default:
      break;
   }
}
```

Reference Industrial Strength C++ 4.4;

Rule 5.5 **Do not alter a control variable in the body of a for statement.**
(QACPP 4235)

Justification Users expect loop control variables to be modified in the for statement, and also that the variable is modified for every iteration. Changing this behaviour makes the code difficult to maintain and understand.

Reference Industrial Strength C++ 4.1;

Rule 5.6 **Do not alter a control variable more than once in a for, do or while statement.**
(QACPP 4236)

Justification The behaviour of iteration statements with multiple modifications of control variables is difficult to maintain and understand.

```
void foo()
{
   for ( int i = 0; i != 10; ++i ) // does this loop terminate?
   {
      if ( 0 == i % 3 )
      {
         ++i;
      }
   }
}
```

Reference Industrial Strength C++ 4.1;

Guideline 5.7 **The control variable in a for loop should be tested against a constant value, not a function or expression.**
(QACPP 4244)

Justification Efficiency

```
// Avoid:
//
```

```
for ( int i = 0; i < xxx.size(); ++i )
{}

// Prefer:
//
const int list_size = xxx.size();
for ( int i = 0; i < list_size; ++i )
{}
```

Rule 5.8 Do not use 'goto'.
(QACPP 4000)

Justification 'goto' should never be used to branch into a block, or to branch around a variable definition. There is always an alternative using the principles of structured programming.

Reference Industrial Strength C++ 4.6;

Rule 5.9 Ensure that every compound statement except the body of a switch statement has a single entry point and (barring the propagation of C++ exceptions) a single exit point.
(QACPP 4020)

Justification A single entry and exit simplifies the control graph for the compound statement and reduces the overall complexity. A single exit point for a function (whose body is also a compound statement) makes it easier for reviewers to check that the exit conditions (such as updating of output parameters) are always satisfied. It also provides a single point for post-condition assertions and for execution trace instructions.

Exclusive with Rule 5.10

Rule 5.10 For functions with non-void return type, ensure all paths have a return statement that contains an expression of the return type.
(QACPP 4022, 4023)

Justification Exiting a function without an explicit return statement is undefined behaviour.

Exclusive with Rule 5.9

Reference ISO C++ 6.6.3/2;

Rule 5.11 Include explicit cases for all alternatives in multi-way conditional structures.
(QACPP 4010, 4070)

Justification Forces programmers to consider all cases and reduces the risk of an unexpected value causing incorrect execution.

See also Rule 5.1

Reference Industrial Strength C++ 4.5;

Rule 5.12 Declare for loop control variables within the for statement instead of using an existing variable.
(QACPP 4230)

Justification This is a best practice rule. The main advantage is that the scope of the loop control variable is naturally limited to the for loop statement, using this construct achieves this minimum scope.

See also Rule 8.2.2, Rule 8.4.4

6 Constants

Rule 6.1 Use suffixes L, U, and UL for all constants of type 'long', 'unsigned int' and 'unsigned long'.

Justification It is good practice to be explicit with constant values. Use upper-case suffixes.

```
const unsigned int a = 0U;
const unsigned int b = 0u;     // avoid
const unsigned int c = 0;      // avoid
const long d = 0L;
const long e = 0l;             // avoid
const long f = 0;              // avoid
const unsigned long g = 0UL;
const unsigned long h = 0Ul;   // avoid
const unsigned long i = 0;     // avoid
```

See also Rule 6.2

Rule 6.2 Use suffixes F and L for all constants of type 'float' and 'long double'.
(QACPP 3012)

Justification It is good practice to be explicit with constant values. Use upper-case suffixes.

```
const float PI = 3.1415F;
const long double R = 0.003;   // avoid
const long double A = 0.0L;
const long double Z = 0.0l;    // avoid
```

See also Rule 6.1

Rule 6.3 Write the value of a character constant to be in the range of its type.

Justification If the value exceeds the range it will be truncated, but this truncation is not portable.

```
char c = 'abcde';              // avoid
int i = 'abcde';               // avoid
```

Rule 6.4 Only use escape sequences defined by the ISO C++ Standard.
(QACPP 0076, 0077, 0446, 0447)

Justification Escape sequences (those beginning with \) other than those defined by the standard have undefined behaviour.

The ISO C+++ Standard defines the following escape sequences:

```
Name              ASCII Name     C++ Name
-----------------------------------------
newline           NL(LF)         \n
horizontal tab    HT             \t
vertical tab      VT             \v
backspace         BS             \b
carriage return   CR             \r
form feed         FF             \f
alert             BEL            \a
backslash         \              \\
question mark     ?              \?
single quote      '              \'
double quote      "              \"
octal number      ooo            \ooo
hex number        hhh            \xhhh
```

Rule 6.5 Do not write character string literal tokens adjacent to wide string literal tokens. (QACPP 5065)

Justification This results in undefined behaviour.

```
#define WW "World"
#define LH L"Hello"

char* txt1 = LH WW;                  // undefined
const char* txt2 = "hello" L"world"; // undefined
```

Guideline 6.6 Global and static data should be const.

Justification Functions that use non-const global or static data are not re-entrant. This causes problems with recursion and multi threading. Global variables frequently cause problems in maintenance.

Exception Singleton functions use static data to ensure only one instance of an object is created.

7 Conversions

Rule 7.1 Always use casting forms: 'static_cast', 'const_cast', 'dynamic_cast' and 'reinterpret_cast' or explicit constructor call. Do not use any other form.
(QACPP 3080)

Justification These casting forms are easier to identify in code and their more narrowly specified purpose makes it possible for compilers to diagnose usage errors. The older, C-style cast - "(type) expr" used to convert one fundamental type to another is subject to implementation-defined effects. For scalar types it can result in silent truncation of the value. For pointers and references, it does not check the compatibility of the value with the target type.

The function style cast - "type(expr)" is equivalent to the C-style cast, is equally difficult to find in code, and has the same problems as the C-style cast. This standard does not preclude constructor style conversions, which use the same syntax as the function style cast. Thus, only function style casts that make use of a conversion operator are prohibited.

See also Rule 3.2.3, Guideline 7.2, Rule 8.3.5

Reference More Effective C++ Item 2;Industrial Strength C++ 6.2;

Guideline 7.2 Minimise the use of casts.
(QACPP 3081)

Justification Excessive use of casts in an implementation may be an indication of a poor design.

See also Rule 7.1, Guideline 10.7

Reference Industrial Strength C++ 6.1;

Rule 7.3 Avoid casting away volatile qualification.
(QACPP 3061)

Justification A volatile object is specified as modifiable outside the program, such as with memory mapped I/O. Casting away volatile means that the compiler may perform optimisations that are not valid, this may lead to unexpected results in optimised builds.

Rule 7.4 Avoid casting away const qualification.
(QACPP 3060)

Justification The existence of a 'const' attribute on a data member or variable is an indication to programmers that the given item is not expected to be changed. Casting away 'const'-ness for an object allows non-const methods to be called for that object which may lead to unexpected behaviour.

Reference Industrial Strength C++ 6.3;

Rule 7.5 Avoid using pointer or reference casts.
(QACPP 3030, 3031)

Justification Avoid using pointer or reference casts. They have been referred to as the goto of OO programming. 'goto' tends to complicate the control flow of a program making it difficult to statically determine the flow of control. Pointer and reference casts complicate the type flow making it harder to determine the type of an object. This, at best, produces difficult to maintain and very error prone code, as it takes control away from the compiler.

Most pointer and reference casts may be eliminated by using virtual functions and tighter control on typing so there is less ambiguity between the declared type of the pointer or reference and the type that is really there.

Exception — If your compiler supports run time type information you may use dynamic_casts. This operator will check that the type you are asking for is really the type of the pointer or reference and if it is not it will return a null. 'dynamic_cast' throws an exception in the case of a reference target type.

Rule 7.6 — Do not convert floating values to integral types except through use of standard library routines.
(QACPP 3011)

Justification — Since mixed precision arithmetic involves implementation defined and undefined behaviour, and since implicit conversions violate this standard, use of specific conversion functions is safer. This rule prohibits the use of casts on floating values.

See also — Rule 7.8, Guideline 10.7

Rule 7.7 — Do not cast pointers to and from fundamental types.
(QACPP 3036, 3037)

Justification — This occurs most in situations where pointers to objects are passed as integral types or stored as integral types. This practice disables the ability of the compiler to perform strong type checking.

See also — Rule 7.8

Rule 7.8 — Do not write code that relies on implicit conversions of arguments in function calls.
(QACPP 2180, 3050)

Justification — For user defined types, implicit type conversions imply construction and destruction of temporary objects. This can create unexpected side-effects and is often inefficient. Remove use of implicit casts by overloading the function(s) in question with respect to any arguments which are implicitly cast.

See also — Rule 3.2.3, Rule 7.6, Rule 7.7

Reference — Industrial Strength C++ 6.1, 7.18;

8 Declarations and Definitions

8.1 Structure

Guideline 8.1.1 With the exception of object definitions and unnamed namespace declarations and definitions, declare in header files: all non-inline functions, classes, variables, enumerations and enumerators, which are named at namespace scope and which have external linkage. (QACPP 5005)

Justification Note that the global scope is included in the term "at namespace scope", as this is the global namespace.

Include the following declarations and definitions in header files

```
enum level { low, med, high };          // enumeration
extern int a;                           // data declaration
int foo( int );                         // function declaration
class Org;                              // type declaration
struct Line{ float dx; float dy; };     // type definition
const float s = 3.0E8;                  // constant definition
                                        // (implicitly internal
                                        // linkage)
```

Do not include external object definitions in header files

```
float f = 3.0E8;                        // global variable definition
```

See also Rule 11.2

Reference Stroustrup;

Guideline 8.1.2 With the exception of unnamed namespace declarations and definitions, define in header files all inline functions which are at namespace scope and which have external linkage. (QACPP 5006)

Justification Note that the global scope is included in the term "at namespace scope", as this is the global namespace.

Where inline function definitions are to be visible to more than one translation unit, place them in header files.

```
inline int get( char* s );                    // inline function
                                              // declaration

inline int get( char* s ) { return ( *s )++; } // inline function
                                              // definition
```

See also Rule 3.1.6, Guideline 3.1.7

Reference Stroustrup;

Guideline 8.1.3 With the exception of unnamed namespace declarations and definitions, define in header files all template definitions which are at namespace scope and which have external linkage. (QACPP 5007)

Justification Where explicit instantiation is not the instantiation model, the template definition must be visible where it is used. Placing all template definition code in headers will mean that any usage of a template will always have the template available.

Reference Stroustrup;

8.2 Scope

Rule 8.2.1 Do not hide declarations in other scopes.
(QACPP 2500, 2501, 2502)

Justification Hiding variables is confusing and difficult to maintain. Changes in variable names may cause errors not detectable at compile time. Variables declared in function scope cannot be accessed if they are hidden by a declaration in an inner scope.

```
int i;
void foo()
{
   int i;  // avoid - hides i in global scope
   i = 10;
}
```

See also Rule 8.3.4

Rule 8.2.2 Avoid global variables.
(QACPP 2300, 2311)

Justification A global variable is one which is declared outside any function, class or unnamed namespace, and has external linkage. Such objects can be accessed directly by any module which contains the appropriate declaration, creating uncontrollable linkage between modules.

Order of initialisation across translation units is unspecified. During program start up, initialisation of globals in other translation units cannot be relied upon. If you need a global variable, use a singleton function.

```
class Application
{
    //
}

Application& theApp()
{
   static Application app;
   return app;
}
```

See also Rule 5.12, Rule 8.4.4

Reference Effective C++ Item 28;Industrial Strength C++ 1.4, 9.1;

Guideline 8.2.3 Always use using declarations or write explicit namespace qualifiers. Do not use using directives.
(QACPP 5134)

Justification Namespaces are an important tool in separating identifiers and in making interfaces explicit.

Using directives, i.e. 'using namespace', allow any name to be searched for in the namespace specified by the using directive.

Using declarations are better than using directives as the name is treated as if declared in the scope containing the using declaration, it will always be considered by lookup not just when a declaration for that name does not exist in the current scope. Only the name specified by the using declaration is brought in from the namespace, so the compiler will not attempt to find other names declared in that namespace as it would with the using directive.

Exclusive with Guideline 8.2.4

Guideline 8.2.4 Only have using namespace directives in the main source file, after all include directives. (QACPP 5135)

Justification A using namespace directive means that names declared in the nominated namespace can be used in the scope of the using namespace directive. This greatly increases the likelihood of hiding names and if an include file contains a using namespace directive then every file that uses that include will suffer the effects of the additional names. If there is more than one using namespace directive then there is a possibility of name collisions.

If a using directive occurs above an include directive then the include file contents may be dependent on names from the nominated namespace. This may lead to maintenance and reuse problems.

Exclusive with Guideline 8.2.3

Reference Herb Sutter, Migrating to namespaces;

8.3 Language Restrictions

Rule 8.3.1 Avoid using the keyword 'static' when declaring objects in namespaces. (QACPP 2313, 2314)

Justification The use of keyword static in declaration of objects in namespaces is deprecated by the C++ standard. Use unnamed namespaces instead.

Guideline 8.3.2 Restrict the use of the 'extern' keyword. Do not write 'extern' where it is implicit.

Justification Keyword 'extern' is used to specify external linkage. It is implicit in function declarations written at global and namespace scope and should not be used in such declarations. Global and namespace scope const objects and typedefs have internal linkage. Recommended practice is to define all const objects with internal linkage in header files only. Hence extern qualification is only necessary when declaring data objects with external linkage.

```
// Header file:
const float s = 3.0E8F;    // internal linkage constant definition
extern int a;              // external linkage object declaration
int foo( int );            // external linkage function declaration

// Implementation file:
int a = 2;                 // external linkage object definition
```

See also Rule 11.2

Reference Stroustrup;

Rule 8.3.3 Do not use the 'auto' or 'register' keywords. (QACPP 5069)

Justification The keyword 'auto' is redundant and 'register' is a hint to the compiler. Most modern compilers can do a better job of register allocation than a programmer.

Rule 8.3.4 Ensure each identifier is distinct.
(QACPP 5140)

Justification Names should not differ only in case (foo/Foo) or in use of underscores (foobar/foo_bar). Similarity of identifiers impairs readability, can cause confusion and can lead to mistakes.

Do not exploit ISO C++ Standard tolerance of the same identifier being declared as different types in the same scope.

```
// Valid C++:
class C;
int C;        // object C hides type of same name
```

See also Rule 8.2.1

Rule 8.3.5 Avoid ambiguous grammar between function style casts and declarations.

Justification Function style casts to fundamental types are not allowed by this standard per Rule 7.1 ('static_cast' should be used). However the following C++ grammar ambiguity remains, see example. All such ambiguities are resolved to declarations. The essence of this rule is do not write declarations with unnecessary brackets which potentially render the declaration ambiguous.

In the following example the declaration of b is ambiguous. It could be taken to mean conversion of b to type A, or a declaration of an A called b. In such cases of ambiguity, a declaration is always assumed.

```
class A {};

A a;      // ok
A (b);    // ambiguous
```

See also Rule 7.1

8.4 Object Declarations and Definitions

Rule 8.4.1 Do not write the characters 'l' (ell) and '1' (one) or 'O' (oh) and '0' (zero) in the same identifier.
(QACPP 5217)

Justification The characters are similar and may be confused by the reader.

Rule 8.4.2 Declare each variable on a separate line in a separate declaration statement. If the declaration is not self-explanatory, append a comment describing the variable.
(QACPP 5075)

Justification Declaring each variable on a separate line makes it easier to find the declaration of a particular variable name. Determining the types of variables becomes confusing when pointers and access specifiers are used for multiple declarations on the same line.

Reference Industrial Strength C++ 5.3;

Rule 8.4.3 Initialise all objects at definition. Never use an object before it has been given a value.
(QACPP 4101, 4102, 4104, 4105, 4200, 4201, 4204, 4205, 4231, 4238)

Justification Evaluating unset objects is guaranteed to cause problems.

Some compilers do not warn when variables are used before they are assigned a value. When the initialise-immediately-before-first-access strategy is used, maintenance invariably adds an access before the initialisation.

See also Rule 8.4.4

Reference More Effective C++ Item 12;Industrial Strength C++ 5.2, 5.5;

Rule 8.4.4 Postpone variable definitions as long as possible.

Justification Avoids unnecessary cost of construction and destruction when a variable is unused (e.g. when an exception is raised). Allows objects to be initialised when declared, hence avoiding default constructor being used followed by later initialisation. This assists in documenting the purpose of variables by initialising them in the context in which their meaning is clear.

```
#include "MyClass.h"

void initialiseBeforeFirstAccess( SomeType value )
{
   MyClass obj; // call default constructor
   obj = value; // call operator=
}

void initialiseAtDeclaration( SomeType value )
{
   MyClass obj( value ); // call constructor taking SomeType
}
```

See also Rule 3.1.2, Rule 5.12, Rule 8.2.2, Rule 8.4.3

Reference Industrial Strength C++ 5.1;

Rule 8.4.5 Do not use the plain 'char' type when declaring objects that are subject to numeric operations. In this case always use an explicit 'signed char' or 'unsigned char' declaration.

Justification Numeric operations that assume signedness of plain char are not portable as it is implementation defined whether plain char is signed or unsigned.

A good way to handle this issue is to have a project wide typedef for a byte type.

```
typedef unsigned char byte;
```

See also Guideline 2.2

Guideline 8.4.6 Use class types or typedefs to indicate scalar quantities.

Justification Using class types to represent scalar quantities exploits compiler enforcement of type safety. If this is not possible typedefs should be used to aid readability of code for manual checking.

```
#include "class_time_stamp.h";

ClassTimeStamp start_time; // prefer (compiler type checking)

long start_time;           // avoid

typedef long TimeStamp;
TimeStamp start_time;      // prefer
```

Reference Industrial Strength C++ 15.12;

Rule 8.4.7 Declare one type name only in each typedef declaration.
(QACPP 5078)

Justification The '&' and '*' in a typedef declaration only apply to the declarator they are adjacent to. Therefore, multiple declarations in a typedef can be confusing and difficult to maintain.

```
// It is not intuitive that value is an int type
// whereas pointer is an int* type.
//
typedef int* pointer, value;
```

Rule 8.4.8 Do not typedef array types.
(QACPP 2411)

Justification Using typedefs of array types can cause problems relating to bounds checking and deletion.

```
typedef int ARRAY_TYPE[ 10 ];

void foo ()
{
   int* array = new ARRAY_TYPE;  // calls new[]
   delete array;                 // incorrect should be delete[]
}
```

Exclusive with Rule 8.4.9

See also Rule 12.3

Reference Industrial Strength C++ 13.6;

Rule 8.4.9 Do not use unbounded (C-style) aggregate types.
(QACPP 0227)

Justification Array bounds checking is not performed on C-style arrays and any attempt to access memory outside the bounds of an array gives rise to undefined behaviour. Also C-style arrays do not maintain a record of the size of the array.

Array semantics should be provided by C++ classes that enforce appropriate bounds. Prefer to use STL vector template where possible.

Exclusive with Rule 8.4.8, Rule 10.2

See also Rule 17.9

Reference Industrial Strength C++ 13.6;

Guideline 8.4.10 Avoid pointers to members.
(QACPP 5070, 5071)

Justification The syntax of pointer to members is obscure and there are inconsistencies between different compiler implementations.

Rule 8.4.11 Use 'const' whenever possible.

Justification This allows specification of semantic constraint which a compiler can enforce. It communicates to other programmers that value should remain invariant - by explicit statement. For example, specify whether a pointer itself is const, the data it points to is const, both or neither:

```
char* p1;             // non-const pointer, non-const data
const char* p2;       // non-const pointer, const data
char* const p3;       // const pointer, non-const data
```

```
const char* const p4;  // const pointer, const data
```

Reference Effective C++ Item 21;

Guideline 8.4.12 Directly append the '*' and '&' to type names in declarations and definitions.

Justification This helps to emphasise that these tokens are part of the type specification.

```
char* str;   // preferred
char *str;   // avoid
```

Guideline 8.4.13 Prefer to use signed numeric values, not unsigned.

Justification Conversions between signed and unsigned types can lead to surprising results.

9 Exceptions

Rule 9.1 Do not throw exceptions from within destructors.
(QACPP 4032, 4631)

Justification When an exception is thrown, stack unwinding will call the destructors of any local objects from where the exception is thrown to where the exception is caught. Should one of these destructors throw another exception, the program will immediately terminate.

Reference More Effective C++ Item 11;Industrial Strength C++ 12.5, ISO C++ 15.5.1;

Rule 9.2 Only throw objects of class type.
(QACPP 3500)

Justification Exceptions pass information up the call stack to a point where error handling can be performed. Class types can have member data with information about the cause of the error, and also the class type itself is further documentation of the cause. User exception types should derive from std::exception or one of its derived classes.

Reference Industrial Strength C++ 12.11;

Rule 9.3 Catch exceptions by reference.
(QACPP 4031)

Justification Using pass-by-pointer for exceptions requires extra calls for memory allocation and deletion which may themselves cause further exceptions or memory loss if the exception object is not deleted. If an exception object is caught by value, information in a derived class may be sliced from the exception in this exception handler.

See also Rule 11.4

Reference More Effective C++ Item 13;Industrial Strength C++ 12.13;

Guideline 9.4 Only use the C++ exception handling mechanism to handle error conditions.

Justification Do not rely on exceptions in normal operation of code. Using exception handling as a control mechanism violates principles of structured programming and can complicate maintenance.

Guideline 9.5 Each application must have some scheme for ensuring that all orphaned resources are properly released when an exception is thrown.

Justification Orphaned resources are resources that are created between the time the try block is entered and the time the exception is thrown. This includes any objects created on the heap (using new) and resources acquired through function calls (e.g. a call to open a database).

Ensure that the application functions correctly when an exception is thrown and that an error condition does not corrupt persistent resources such as databases. Standard exception handling behaviour only invokes destructors for local objects.

See also Guideline 9.6

Reference Stroustrup;

Guideline 9.6	Each application that acquires resources that are not automatically freed at program termination must use some mechanism to ensure that acquired resources are freed if the program unexpectedly terminates.
Justification	This ensures that an error condition does not corrupt persistent resources such as databases.
See also	Guideline 9.5

10 Expressions

Rule 10.1 Use symbolic names instead of literal values in code. Do not use "magic" numbers. (QACPP 4400, 4401, 4402, 4403, 4404)

Justification By eliminating "magic" numbers from the body of the code and placing them in header files, the code becomes easier to maintain. Symbolic names should be self documenting.

Exception Literals with intuitive meaning: the character literal '\0', numeric literals 0 & 1 and the boolean literals true and false.

String literals that only occur in the code once. This exception does not apply where there is a requirement for internationalisation.

Reference Industrial Strength C++ 5.4;

Rule 10.2 Access to an array should be demonstrably within the bounds of the array. (QACPP 4307)

Justification This improves robustness and security. This applies to indices and also to C library functions that modify arrays, such as sprintf() and scanf(). Functions that do not provide a means of bounds checking, such as gets(), should not be used.

Exclusive with Rule 8.4.9

Rule 10.3 Do not assume the order of evaluation of operands in an expression. (QACPP 3220, 3221)

Justification The C++ language standard does not guarantee the order of evaluation of sub-expressions within an expression between sequence points. Sequence points are those points in the evaluation of an expression at which all previous side-effects can be guaranteed to have taken place.

The following example has implementation defined results:

```
x = foo( ++i, ++i ); // either ++i could be evaluated first
```

Reference Industrial Strength C++ 15.1, 15.22;

Rule 10.4 Use parentheses in expressions to specify the intent of the expression. (QACPP 3700)

Justification Rather than letting the operator precedence specify the order, use parentheses to ensure clarity and correctness. Remove doubt about behaviour of complex expressions. What is obvious to one programmer may not be to another, and may even be incorrect.

Each pair of operands of a binary operator, except for arithmetic and assignment operators, should be surrounded by parentheses. Each operand to a relational or boolean operator should be either a single element (no exposed operators) or should be enclosed in parentheses.

Rule 10.5 Always discard the result of an assignment operator. (QACPP 4071)

Justification Assignment operators are frequently mistaken for comparison operators.

Assignment operators should not be used in any type of statement other than an assignment statement, where the result of the assignment operator is discarded and only the side effect (changing the value referenced by the left-hand side) is retained.

```
int main( int argc, char** argv )
{
   int i = 1;
   int j = 2;

   // Confusing use of assignment operator, always discard the result
   //
   if ( ( j = i ) == 1 )
   {
      std::cout << "hit" << std::endl;
   }

   // Prefer to write
   //
   j = i;
   if ( 1 == j )
   {
      std::cout << "hit" << std::endl;
   }

   return 1;
}
```

Guideline 10.6 When comparing variables and constants for equality always place the constant on the left hand side.

Justification A common mistake in C++ is to write '=' for '==' in comparisons. By placing the constant on the left hand side the compiler protects against this mistake.

```
int a = getValue();
if ( a == 10 )   // avoid: error prone
{}

if ( 10 == a )   // prefer: compiler will warn if '=' is used
{}
```

Guideline 10.7 Do not use expressions which rely on implicit conversion of an operand. (QACPP 0150, 3000, 3001, 3010, 3011, 3012, 3050, 3051, 3054, 3062, 3072, 3073)

Justification The effect of implicit conversions are frequently either undefined or implementation-defined. Be explicit about any type conversions that are required.

Implicit conversions include those resulting from implicit use of a user-defined constructor and conversion operator.

See also Rule 3.2.3, Guideline 7.2, Rule 7.6

Reference Industrial Strength C++ 6.1;

Rule 10.8 Ensure expressions used in assertions are free from side-effects.

Justification Neither insertion nor removal of the assertion should affect the execution of the system when the routine is used correctly.

Reference Industrial Strength C++ 11.1;

Rule 10.9 Do not code side effects into the right-hand operands of '&&', '||', 'sizeof' or 'typeid'. (QACPP 3230, 3240)

Justification The right-hand operands of the logical AND and logical OR operators are conditionally executed with the result that side-effects present in these operands might not be executed. The operand of sizeof is never evaluated so that the side-effects that would normally occur from evaluating the expression do not take place. The operand of typeid is evaluated only if it represents a polymorphic type.

```
bool doSideAffect();

class C
{
public:
   virtual ~C(); // polymorphic class
};

C& foo();

void foo( bool condition )
{
   if ( false && doSideAffect() ) // doSideAffect not called!
   {}

   if ( true || doSideAffect() )  // doSideAffect not called!
   {}

   sizeof( doSideAffect() ); // doSideAffect not called!
   typeid( doSideAffect() ); // doSideAffect not called!
   typeid( foo() );          // foo called to determine the polymorphic
type
}
```

Reference ISO C++ 5.2.8, 5.3.3;

Rule 10.10 Avoid statements that have no side effects. (QACPP 3242, 3243, 3244, 3245)

Justification For example: The left hand side of a comma operator is evaluated for its side effects only, and does not affect the value of the expression. If the left hand side has no side effects it is redundant. Removing it makes the expression more readable.

```
static void foo( void )
{
   unsigned int a = 0U;
   unsigned int b = 0U;
   a = (0U, b);   // left side of comma operator has no side effect
   a++;
   b++;
}
```

Rule 10.11 Do not apply the following bitwise operators to signed operands: shift operators ('<<', '>>'), bitwise AND ('&'), exclusive OR ('^') and inclusive OR ('|'). (QACPP 3003)

Justification Although left-shift is defined for signed operands, right-shift applied to a negative operand is implementation defined. This asymmetry can cause confusion unless shift operations are restricted to unsigned operands only. Bitwise operations on signed operands rely on the representation used for integral types and should be avoided.

Exclusive with Rule 10.12

Rule 10.12 Validate arguments to be used in shift operators.
(QACPP 3321, 3322)

Justification Right hand side operands to a shift operator which are negative or are larger than the number of bits in the left hand side will lead to undefined behaviour.

Exclusive with Rule 10.11

Reference ISO C++ 5.8;

Rule 10.13 Do not mix signed and unsigned data items in the same expression.
(QACPP 3000, 3002)

Justification Conversion from unsigned to signed integral types, taking integral promotion into account, involves implementation defined behaviour and is a portability risk.

Rule 10.14 Do not mix arithmetic precision in expressions.
(QACPP 3000, 3001, 3010, 3011, 3012, 3051, 3054)

Justification Since mixed precision arithmetic involves implementation defined and undefined behaviour, it is safer, for portability reasons, to consistently use double precision for floating point expressions, unless the application specifically requires single or extended precision, or homogeneous integral types.

Rule 10.15 Do not write code that expects floating point calculations to yield exact results.
(QACPP 3270, 4234)

Justification Equivalence tests for floating point values should use <, <=, >, >=, and not use == or !=. Floating point representations are platform dependent, so it is necessary to avoid exact comparisons.

```
bool double_equal( const double a, const double b )
{
   const bool equal = fabs( a - b ) < numeric_limits<double>::epsilon;
   return equal;
}

void foo( double f )
{
   if ( f != 3.142 )                  // avoid
   {}

   if ( double_equal( f, 3.142 ) )   // prefer
   {}
}
```

Rule 10.16 Do not use the increment operator ('++') on a variable of type 'bool'.
(QACPP 3291)

Justification This is deprecated. Use specific assignment or user functions like 'toggle()', 'set()' and 'clear()'.

Rule 10.17 Guard both division and remainder operations by a test on the right hand operand being non-zero.
(QACPP 0015, 0435, 4308)

Justification For defensive programming purposes, either a conditional test or an assertion should be used.

```
int doDivide( int number, int divisor )
{
   assert( 0 != divisor );
   return number / divisor;
```

```
}
```

Reference Rule 10.18;

Guideline 10.18 Guard the modulus operation to ensure that both arguments are non-negative.

Justification Use defensive programming to reduce the effect of implementation defined and undefined behaviour.

Reference Rule 10.17;

Rule 10.19 Do not use the comma operator.
(QACPP 3243)

Justification Using the comma operator is confusing and is nearly always avoidable without any loss of readability, program size or program performance.

Rule 10.20 Do not use the ternary operator (?:) in expressions.
(QACPP 5120)

Justification Evaluation of a complex condition is best achieved through explicit conditional statements. Using the conditional operator invites errors during maintenance.

Rule 10.21 Apply unary minus to operands of signed type only.
(QACPP 3002)

Justification Unary minus on an unsigned expression, after applying integral promotion, gives an unsigned result which is never negative.

11 Functions

Rule 11.1 **All functions that have the same name should have similar behaviour, varying only in the number and/or types of parameters.**

Justification This aids maintainability, reuse and conceptual clarity. An overloaded function should represent a set of variations on the same behaviour.

See also Guideline 3.1.9, Rule 12.1

Rule 11.2 **Enclose all non-member functions that are not part of the external interface in the unnamed namespace in the source file.**

Justification The preferred method of making functions non-linkable from other translation units is to place the definitions inside an unnamed namespace; explicitly declaring functions static is now deprecated. All other non-member functions shall not use a storage class specifier and hence by default are externally visible.

See also Guideline 8.1.1, Guideline 8.3.2

Rule 11.3 **Specify the name of each function parameter in both the function declaration and the function definition. Use the same names in the function declaration and definition.**
(QACPP 2017)

Justification This helps to document the function, reducing the need for comments and making it easier to refer to a parameter within documentation.

Exception However, names of unused parameters may be omitted to avoid "unused variable" warnings. e.g. where the implementor of a function does not have control over the function interface.

Rule 11.4 **Use pass-by-reference in preference to pass by value or pass by pointer.**
(QACPP 2010, 2013, 2014)

Justification Pass by reference is more efficient than pass by value as the copy constructor of the object will not be invoked. Passing class objects by value can result in an object of a base class being passed instead of a copy of the actual object, reducing extendibility (not to mention slicing of the object).

The C-style use of pointer types as function formal parameters in order to update object(s) in the calling function should be avoided. These formal parameters should be declared as reference types.

See also Rule 3.2.3, Rule 9.3, Rule 11.5, Rule 17.5

Reference Effective C++ Item 22;Industrial Strength C++ 7.5, 7.6;

Rule 11.5 **Declare read-only parameters of class type as const references. Pass by value read-only parameters that are of a fundamental type.**

Justification Declaring parameters as const references allows for compile time checking that the object is not changed.

There is no advantage to passing a read-only argument of fundamental type by reference, since the size of most fundamental types is less than or equal to the size of a pointer.

See also Rule 11.4

Reference Industrial Strength C++ 7.3;

Rule 11.6 Do not use ellipsis '...' in function parameters.
(QACPP 3074)

Justification Use of the ellipsis notation (...) to indicate an unspecified number of arguments should be avoided. It is better to develop specific methods for all situations. Use of ellipsis defeats the type checking capability of C++. The use of ellipsis for non-POD types is undefined.

Reference ISO C++ 5.2.2/7;

Rule 11.7 A function should not return a reference or a pointer to an automatic variable defined within the function. Instead, it should return a copy of the object.
(QACPP 4026, 4027, 4028)

Justification Memory for the variable will be deallocated before the caller can reference the variable. This error might not cause an error in testing. Returning local objects by value is ok.

For example:

```
// Do not return a pointer or reference to a local variable :
class String
{
public:
   String( char* A );
   String( const String& );
};

String& fn1( char* myArg )
{
   String temp( myArg );
   return temp;           // temp will be destroyed before
                          // the caller gets it
}

String fn2( char* myArg )
{
   String temp( myArg );
   return temp;
}
```

Reference Effective C++ Items 23, 29;Industrial Strength C++ 5.9;

Rule 11.8 Only declare trivial functions 'inline'.
(QACPP 2131, 2133, 2134)

Justification The 'inline' keyword is only a hint, and a compiler may not inline every function declared with the 'inline' keyword.

Inline functions do not necessarily improve performance and they can have a negative effect. Inappropriate use will lead to longer compilation times, slower runtime performance and larger binaries.

See also Rule 3.1.6, Guideline 3.1.7

Reference More Effective C++ Item 24;Industrial Strength C++ 7.1, 7.2;

Rule 11.9 Do not overload on both numeric and pointer types.
(QACPP 2020)

Justification When there are both pointer and numeric overloads of a function it is not obvious which function is called when there is a numeric argument. The ambiguity and confusion is best avoided altogether.

```
void f( char );
void f( class X* );

void fn()
{
   f( 0 );         // ambiguous
   f( 1 );         // calls f( char )
   f( '1' );       // calls f( char )
}
```

Reference Effective C++ Item 25;

12 Memory Management

Rule 12.1 Do not use default arguments with overloaded functions.

Justification Default arguments or overloading allow for the same function to be called in more than one way. If an overloaded function has default arguments, ambiguities may arise when calling that function. It is better to avoid the problems that this creates in code comprehension and choose between using overloaded functions or a single function with default arguments.

```
// Avoid, calls to foo with 1 arg are ambiguous
//
void foo( int );
void foo( int, char c = 10 );

// Prefer, bar(int) is implemented in terms of bar( int, char )
//
void bar( int, char c );
void bar( int );

// Prefer, default arg is okay here as there is no overloads of car
//
void car( int, char c = 10 );
```

See also Guideline 3.1.9, Rule 3.3.12, Rule 11.1

Reference Effective C++ Item 24;

Rule 12.2 Allocate memory using 'new' and release using 'delete'. Do not use the C memory management functions malloc(), realloc(), and free().
(QACPP 3332, 3901)

Justification Functions 'new' and 'delete' invoke constructors and destructors.

Undefined results will occur if 'delete' is invoked on a malloc'ed pointer, or free is invoked on an object created with 'new'.

C functions such as strdup() that use any of the C memory management functions should also not be used.

Reference Effective C++ Item 3;Industrial Strength C++ 13.1;

Rule 12.3 Ensure the form used when invoking 'delete' is the same form that was used when memory was allocated using 'new'.
(QACPP 3330, 3331)

Justification For every allocation of an array using new[], the corresponding delete of the array shall use delete[]. If delete without the array operator is used on memory allocated using the array new operator, the behaviour is undefined.

```
void foo()
{
   int * array = new int[ 10 ];
   delete array;              // undefined behaviour
}

void bar()
{
   int * obj = new int;
   delete[] obj;              // undefined behaviour
}
```

See also Rule 8.4.8

Reference Effective C++ Item 5;More Effective C++ Item 9;Industrial Strength C++ 8.1, 8.2;

Rule 12.4 Do not specify the number of elements when deleting an array of objects.
(QACPP 0013)

Justification This is an obsolete feature which was never added to the ISO C++ standard.

Rule 12.5 Do not return a dereferenced pointer initialised by dynamic allocation within a function.

Justification In resource management it is important that ownership of resources is clearly documented. If a resource is returned from a dereferenced pointer, it will not be clear to the caller of the function that a resource is changing ownership.

Reference Effective C++ Items 29, 31;

Rule 12.6 Write operator delete if you write operator new.
(QACPP 2160)

Justification Operator new and operator delete should work together. Overloading operator new means that a custom memory management scheme is in operation, if the corresponding operator delete is not provided the memory management scheme is incomplete.

Reference Effective C++ Item 10;Industrial Strength C++ 8.5;

Rule 12.7 Document that operator new and operator delete are static by declaring them static.
(QACPP 2162)

Justification 'Operator new' and 'operator delete' are implicitly static functions, however specifying 'static' explicitly helps to document the class.

Rule 12.8 On use of delete always set the pointer to zero after the delete.

Justification Setting the pointer to zero after a delete operation helps to trap continued use of that pointer as well as giving a clear indication that it no longer points to anything.

Note that zeroing is not necessary:
1. When the pointer is assigned immediately after the delete.
2. On deallocating in a destructor as the object goes out of scope.

13 Portability

Guideline 13.1 **Avoid implementation defined behaviour.**
(QACPP 0027, 0029)

Justification Implementation-defined behaviour can vary dramatically across compilers, this causes portablily problems between different compilers and different versions of the same compiler.

See also Rule 13.4

Reference Industrial Strength C++ 15.1;

Guideline 13.2 **Use standard language features and standard library functions in preference to extra functionality provided by the operating system or environment.**

Justification The extra functionality may not be available on different compilers or different platforms.

Reference Industrial Strength C++ 15.2;

Rule 13.3 **Do not exceed the translation limits imposed by the ISO C++ Standard.**

Justification Exceeding the translation limits may hamper the compilation of the source code and make the code non-portable.

This rule requires that the code comply with the limits stated in Annex B of the ISO C++ Standard.

Reference ISO C++;

Rule 13.4 **Do not use compiler specific language or pre-processor extensions.**
(QACPP 0027, 0028, 0029, 0060, 0095, 1040)

Justification Portability and compiler compatibility, including upward compatibility with future versions of the same compiler.

Examples include compiler-specific pre-processor directives, functions and keywords beginning with a double underscore. Note that #pragma is also non-portable, but is sometimes essential.

See also Guideline 13.1

Reference Industrial Strength C++ 15.2;

Rule 13.5 **Do not use the 'asm' declaration.**
(QACPP 1100)

Justification Use of inlined assembler should be avoided since it restricts the portability of the code. If it is essential to use inlined assembler, the assembler dependency should be abstracted out to function(s) that contain inlined assembler only.

Rule 13.6 **Do not make any assumptions about the internal representation of a value or object.**
(QACPP 2176, 3033)

Justification This rule helps ensure portability of code across different compilers and platforms. Here are some recommendations on assumptions that can be made about the target architecture or compiler.

- Do not assume that you know the size of the basic data types, an int is not always 32 bits in size.

- Do not assume byte order, you may wish in the future to port your code from a big endian architecture to a little endian architecture or vice versa.
- Do not assume that non POD class data members are stored in the same order as they are declared.
- Do not assume that two consecutively declared variables are stored consecutively.
- Only use built-in shift operators on unsigned fundamental types.

Reference Industrial Strength C++ 13.7;

Rule 13.7 Do not cast a pointer to fundamental type, to a pointer to a more restrictively aligned fundamental type.
(QACPP 3033)

Justification Aids portability. Different hardware architectures may have different byte alignment rules. In most cases, this rule is equivalent to saying that a pointer to a fundamental type should not be cast to a pointer to a longer fundamental type.

Reference Industrial Strength C++ 15.8;

14 Pre-processor

Rule 14.1 Use the C++ comment delimiters "//". Do not use the C comment delimiters "/* ... */".
(QACPP 1050)

Justification The scope of C++ comments is clearer. Errors can result from nesting of C comments.

Reference Effective C++ Item 4;Industrial Strength C++ 3.4;

Guideline 14.2 Do not use tab characters in source files.
(QACPP 5200)

Justification Use \t in string and character literals instead of the tab character.

Tab width is not consistent across all editors and tools. The conventional C++ tab width is 4, but most tools use 8. Not all tools provide the ability to change tab widths and making sure that tab widths are correct across all tools can be challenging. Storing spaces ensures that formatting is preserved on printing and editing using different tools.

This does not mean that tabs may not be used when editing, provided the editor can convert tabs to spaces when the file is stored.

Guideline 14.3 Write pre-processor directives to begin in column 1 with no whitespace between the '#' and the pre-processor directive.
(QACPP 5229)

Justification It is good practice to adopt a common approach to writing pre-processor statements and this is a common convention.

```
#ifdef SOME_FLAG
#define SOME_OTHER_FLAG
#else
#define YET_ANOTHER_FLAG
#endif
```

Exclusive with Guideline 14.4

Guideline 14.4 Write pre-processor directives to begin in column 1 with whitespace between the '#' and the pre-processor directive representing nesting in preprocessor conditionals.
(QACPP 5230)

Justification It is good practice to adopt a common approach to writing pre-processor statements.

```
#ifdef SOME_FLAG
#   define SOME_OTHER_FLAG
#else
#   define YET_ANOTHER_FLAG
#endif
```

Exclusive with Guideline 14.3

Rule 14.5 Control conditional compilation by the use of, or absence of, a pre-processor token definition.
(QACPP 0016)

Justification Control of conditional compilation through a specific value of a pre-processor token is prone to error. Use #ifdef and #ifndef, rather than #if. Tokens which control conditional compilation should be checked only for the presence of a definition.

Exception In some cases it is necessary to use a specific value, for example if conditionally compiled code is specific to a particular version of a compiler.

Rule 14.6 Use the __cplusplus identifier to distinguish between C and C++ compilation.

Justification C++ compilers are required to define __cplusplus to indicate a C++ environment, which can be used to select function prototypes and differentiate between C and C++ environments.

Guideline 14.7 Do not include comment text in the definition of a pre-processor macro. (QACPP 5117)

Justification Since macros are expanded without regard for C++ syntax, an incorrect form of comment can result in required code being commented out. The resulting compiler diagnostics (if any) can be very hard to interpret. Also, some compilers have implementation-defined limits on the length of the fully expanded line and do not necessarily report errors when this limit is exceeded.

Place comments just before the line containing the '#define' directive and not in the body of the macro.

Rule 14.8 Ensure that the last line of all files containing source code is followed by a new-line. (QACPP 5118)

Justification Behaviour is undefined where a source file, that is not empty, does not end in a new-line character

Reference ISO C++ 2.1/2;

Rule 14.9 Use <> brackets for system and standard library headers. Use "" quotes for all other headers. (QACPP 1011, 1012)

Justification It is important to distinguish the two forms of #include directive not only for documentation purposes but also for portability. Different compilers may have different search methods and may not find the correct header file if the wrong form of #include is used.

Reference Industrial Strength C++ 15.4;

Rule 14.10 Do not include a path specifier in file names supplied in #include directives. (QACPP 1010, 1013)

Justification Specific directory names and paths may change across platforms and compilers. For example, limits.h is in the sys subdirectory for the Sun compiler, but in the normal include directory for MSVC++.

The include directory path should be specified in the makefile. Paths in include file names are not portable:

```
#include <sys/limits.h>
```

Put paths in the makefile; #include only the filename:

```
cc -I$(INCLUDE) -I$(INCLUDE)/sys
#include <limits.h>
```

Reference Industrial Strength 15.5;

Rule 14.11 Incorporate include guards in header files to prevent multiple inclusions of the same file. (QACPP 0063, 0103, 5209)

Justification This resolves the problem of multiple inclusion of the same header file, since there is no way of knowing in which sequence the header files will be included or how many times they will be included.

This prevents compiler and linker errors resulting from redefinition of items. It also prevents recursive file inclusion.

The defined macro should be the same as the name of the file, with any '.' changed to '_', and all characters in upper-case.

```
// File example.h:

#ifndef EXAMPLE_H
#define EXAMPLE_H

// All declarations and definitions

#endif
```

Reference Industrial Strength C++ 1.7;

Rule 14.12 Use lower-case for file names and references to file names (such as include directives). (QACPP 5121)

Justification This rule results from an incompatibility between mono-case and case-sensitive file systems. For example, an include directive with a mixed-case file name will work successfully in DOS, but fail in UNIX unless the case of the file name matches.

Reference Industrial Strength C++ 15.6;

Rule 14.13 Write header files such that all files necessary for their compilation are included.

Justification This means that every header file is self sufficient: a programmer who puts #include "header.h" should not have to #include any other headers prior to that file.

Rule 14.14 Enclose macro arguments and body in parentheses. (QACPP 1030, 1031)

Justification If the body of a macro contains operators and these operators and the macro arguments on which they operate are not enclosed within parentheses then use of the macro in certain contexts could give rise to unexpected results.

For example:

```
#define BAD_SQ( A )  A * A
#define OK_SQ( A )   ( ( A ) * ( A ) )

int x = BAD_SQ( 6 + 3 );   // expands to: 6 + 3 * 6 + 3
int y = OK_SQ( 6 + 3 );    // expands to: ( ( 6 + 3 ) * ( 6 + 3 ) )
```

Exception It is not necessary to place the body of an object-like macro in parentheses if it consists of a single token. For example, a macro body comprising one token which is a literal or an identifier should not be parenthesized.

Reference Effective C++ Item 1;

Rule 14.15 Do not use pre-processor macros to define code segments. (QACPP 1023)

Justification Macro expansion is performed by textual substitution, without regard for the underlying syntax or semantics of the language.

Use inline expansion and/or function templates to achieve the desired effect. These obey all the normal language rules.

Rule 14.16 Do not use the NULL macro.
(QACPP 5128)

Justification The NULL macro is not defined by the C++ standard and so its usage is not portable. Use 0 instead as it is valid for any pointer type.

Varying definitions of NULL in third-party libraries may be incompatible and lead to significant porting problems.

Rule 14.17 Use const objects or enumerators to define constants, not #define.
(QACPP 1020, 1021)

Justification This should be done for type safety and maintainability. Preprocessor constants do not have a type other than the literal type and this allows for misuse. In addition, most debuggers do not understand #define'd values, whereas values with symbolic names can be accessed.

Reference Effective C++ Item 1;Industrial Strength C++ 13.5;

Rule 14.18 Do not use digraphs or trigraphs.
(QACPP 5210)

Justification Trigraphs are special three character sequences, beginning with two question marks and followed by one other character. They are translated into specific single characters like \ or ^. Digraphs are special two character sequences that are similarly translated.

Do not use '??' at any point in the code as in combination with some characters this will be translated and cause confusion. Be aware of digraph character sequences and avoid them. It is possible to avoid such sequences arising accidentally by using spaces.

```
Trigraph   Equivalent    Digraph    Equivalent
??=        #             %:%:       ##
??(        [             %:         #
??<        {             <:         [
??)        ]             <%         {
??>        }             :>         ]
??/        \             %>         }
??'        ^
??!        |
??-        ~

// Here the ??/??/?? becomes \\?? after trigraph translation

cout << "Enter date ??/??/??";

// Here the <::std::pair becomes [:std::pair
//
::std::vector<::std::pair<int, int> > vector_of_pairs;
```

15 Structures, Unions and Enumerations

Rule 15.1 Do not use variant structures (unions).
(QACPP 2176)

Justification Unions provide a way to alter the type ascribed to a value without changing its representation. This reduces type safety and is usually unnecessary. In general it is possible to create a safe abstraction using polymorphic types.

Reference Industrial Strength C++ 13.7;

Rule 15.2 Do not include member functions or access specifiers in struct types.
(QACPP 2171, 2173, 2175)

Justification Treating struct as a true class type has no advantages over using the class specifier. Struct should be used to designate the POD type equivalent to a C struct. Note that the default access specifier for structs and unions is public.

Reference Industrial Strength C++ A.14;

Rule 15.3 Do not rely on the value of an enumerator.

Justification When using an enumerator to represent a constant value, using the symbolic name improves documentation and maintainability.

```
enum Colours { RED = 0xA, GREEN, BLUE };
bool foo()
{
   Colours colour = GREEN;
   if ( 11 == colour )        // avoid
   {}
   else if ( BLUE == colour ) // prefer
   {}
   else
   {
      // colour is red?
   }
}
```

See also Rule 15.4

Rule 15.4 Avoid casting an integer to an enumeration as the result of this cast is unspecified if the value is not within the range of the enumeration.
(QACPP 3013)

Justification The underlying type chosen to represent an enumeration is implementation defined. The type is only required to be large enough to hold all the values defined in the enumeration set. A cast from an integral value to an enumeration may cause overflow on the integral value.

Even when an overflow does not occur, an enumerator may not exist for that integral value and the enum object will not have a symbolic name for its value.

```
enum Colours { RED, GREEN = 2, BLUE };
void bar()
{
   Colours colour = static_cast<Colours>( 1000 ); // may cause overflow
   if ( 1000 == colour )
   {
                                                  // may not be reached
   }
```

```
}

void foo()
{
   Colours colour = static_cast<Colours>( 1 );    // value not in set
   switch ( colour )
   {
   case RED:
   case GREEN:
   case BLUE:
      break;
   default:
      break;                                      // value not handled
   }
}
```

See also Rule 15.3

16 Templates

Rule 16.1 Avoid implicit conversions from class templates to non-dependent types as this ensures that clients cannot bypass the class interface.
(QACPP 2183)

Justification Each instantiation of a class template is a different type, but when there is a conversion operator to a non dependent type then different instantiations can be treated as if they were that type. This is a particular problem when an implicit conversion to a fundamental type is available, as any two different instantiations may be operands to every built-in operator.

```
// Example with smart pointers:

template< typename T >
class SmartPointer
{
public:
   SmartPointer( T* );

   operator void const*();
};

// Here the designer added conversion to void
// const* to help comparison to the null pointer.
// However a bad side effect to this feature
// is that it is possible to generate equality
// comparison between two smart pointers on
// different types:

void doIt( AA* pa, BB* pb )
{
   SmartPointer<AA> spa( pa );
   SmartPointer<BB> spb( pb );
   if (spa == spb)   // problem! comparing different pointer types
   {}
}
```

Reference More Effective C++ Item 28;

Rule 16.2 Do not define a class template with potentially conflicting methods.

Justification Defining a template with potentially conflicting methods will cause problems with some instantiations of that template.

```
template< typename T >
class A
{
public:
   void foo( T );
   void foo( int );
};

template class A< int >;    // error void foo(int) declared twice
```

Rule 16.3 Only instantiate templates with template arguments which fulfill the interface requirements of the 'template'.

Justification Using a template argument where some of the requirements for the argument are not met may cause compilation errors. Implicit instantiation only occurs for the parts of a template definition that are used. If an instantiation error is contained in a function definition that is not called then the error will not be seen unless maintenance leads to that function being called. Explicit template instantiation will instantiate all parts of the template defintion, ensuring that the template argument is valid.

```
class person
{
public:
   int getAge( void );
};

template< class T >
class singleVal
{
public:
   bool isMatch( T t )
   {
      return ( singleton == t );
   }
private:
   T singleton;
};

void foo( person const& other )
{
   singleVal< person > emperor;     // no error as isMatch not yet called
   if ( emperor.isMatch( other ) )  // instantiation error,
   {}                               //    no 'op==' for person
}

// explicit instantiation of complete template definition
//
template class singleVal< person >; // error! no op== for person
```

Guideline 16.4 Only use templates when the behaviour of the class or function template is completely independent of the type of object to which it is applied.

Justification If behaviour varies with the type of object, inheritance with virtual functions should be used.

Templates should implement genericity and should be applicable to any type that meets the interface requirements specified in the template definition.

```
template< typename T > void foo( T t )
{
   // Avoid: a template definition should not have behaviour
   // defined by the dynamic types of template arguments
   //
   if ( 0 != dynamic_cast<SomeType*>( t ) )
   {}
}

template< typename T > void bar( T t )
{
   // Prefer: this template works with any type that provides
   // the member someFunction in its interface
   //
   t.someFunction();
}
```

17 Standard Template Library (STL)

Rule 17.1 Use Standard C++ Library headers defined by the language standard and not outdated .h headers. For example, use <iostream> and not <iostream.h>, <cstdio> and not <stdio.h>. (QACPP 1014)

Justification ISO C++ defines the standard implemention of library components. Programmars should use these versions of the library rather than vendor-specific or C library versions.

Rule 17.2 Use Standard Template Library containers and algorithms in preference to custom designs.

Justification The STL forms part of the language standard and represents a well-tested library of re-usable code.

Guideline 17.3 Make copying efficient for objects in containers.

Justification It is important to be aware that STL containers use copy operations and to implement these operations as efficiently as possibly. When an object is added to a container it is copied and the copy is stored inside the container; for objects of class type this copy is made by the class copy constructor. For some container types, including vector, objects may be moved inside the container; this move will use the class copy assignment operator.

```
class MyClass
{
public:
   MyClass();
   MyClass( MyClass const& rhs );              // copy ctor
   MyClass& operator=( const MyClass& rhs );   // copy assignment
};

void foo( const MyClass& obj, const MyClass& anotherObj )
{
   std::vector< MyClass > vec;

   // Call to push_back will call the copy constructor.
   //
   vec.push_back( obj );

   // Call to insert will call the copy assigment operator for each
   // object stored after the insert iterator.
   //
   vec.insert( vec.begin(), anotherObj );
}
```

See also Guideline 17.4

Reference Effective STL Item 3;

Guideline 17.4 Where copying is expensive use containers of pointers or smart pointers.

Justification Pointers are small and have builtin operators for copying values. Because of this containers of pointers are efficient for insertion, sorting and other operations.

If containers of pointers are used then it is the responsibility of the programmer to manage the lifetime of the objects. This can be done with a reference counting smart pointer class.

```
class BigClassWithLotsOfData {};

void badVectorUsage( std::vector< BigClassWithLotsOfData >& vec )
{
   BigClassWithLotsOfData newObj;
```

```
   vec.push_back( newObj );   // calls expensive copy constructor!
}

void goodVectorUsage( std::vector< BigClassWithLotsOfData* >& vec )
{
   // This only copies a pointer so insertions are cheap.
   //
   BigClassWithLotsOfData* pNewObj = new BigClassWithLotsOfData;
   vec.push_back( pNewObj );
}
```

See also Guideline 17.3, Rule 17.5

Reference Effective STL Item 3;

Rule 17.5 Do not attempt to insert derived class objects in a container that holds base class objects.

Justification If you attempt to insert an object of derived type into a container of base type objects then slicing will occur and the container will not hold the intended object. The problem of slicing is eliminated when pointers to base class objects are stored.

See also Rule 11.4, Guideline 17.4

Reference Effective STL Items 3, 7;

Rule 17.6 Use empty() instead of checking size() against zero.

Justification Testing empty() and comparing size() to zero are the same thing, however for some containers it is expensive to calculate the number of elements so it is less efficient to compare the size to zero. It is always better to use empty when testing if a container has elements.

```
std::list< Node > myList;

if ( false == myList.empty() )   // constant time test
{
   doSomething();
}

if ( 0 == myList.size() )        // linear complexity operation
{
   doSomethingWheneverFinishedCountingAllNodes();
}
```

Reference Effective STL Item 4;

Guideline 17.7 Do not use STL containers as public base classes.

Justification All STL containers lack a virtual destructor. If a class with a non virutal destructor is used as a base class it is possible to get undefined behaviour on destruction, this happens if the derived class is allocated on the heap and later deleted through a base class reference.

```
class MyVector : public std::vector {};

void doSomething()
{
   MyVector* pHeapVec = new MyVector;   // allocate derived obj on heap
   std::vector* pBaseVec = pHeapVec;    // access through base class
   delete *pBaseVec;                    // undefined behaviour!
}
```

See also Rule 3.3.2

Rule 17.8 Never create containers of auto_ptrs.

Justification 'auto_ptr' has destructive copy semantics, this means that when you copy an auto_ptr the source loses its value. STL containers require that element types provide copy semantics such that the source and destination are equivalent after a copy.

The C++ Standard prohibits containers of auto_ptrs so they should not compile. However some STL implementations and some compilers do not reject them.

```
class MyClass {};

void foo( vector< auto_ptr< MyClass > >& myVec )
{
   // After myObj2 is initialised myObj1 is a 0 ptr!
   //
   auto_ptr< MyClass > myObj1 = myVec[ 0 ];
   auto_ptr< MyClass > myObj2 = myObj1;
}
```

Reference Effective STL Item 8;

Rule 17.9 Use vector and string in place of dynamically allocated arrays.

Justification vector and string automatically manage their storage requirements so the programmer does not need to manage dynamically allocated memory. This removes the potential for inefficiency and memory related bugs that can occur with dynamically allocated arrays.

vector and string contain commonly needed operations and are interoperable with STL algorithims so programmers can avail themselves of a large body of efficient and reliable code.

See also Rule 8.4.9

Reference Effective STL Item 13;

Guideline 17.10 Where possible pre-allocate in containers to save unnecessary reallocations.

Justification STL containers grow as needed when elements are inserted. However, increasing the capacity of a container can be costly as it involves allocation of memory and potentially moving previously inserted elements. While this overhead is not always an issue it is better to reserve the required storage space in advance as this reduces the number of memory allocation requests and limits having to move elements.

```
void badPushBackManyNumbers( vector< int >& vec )
{
   // This code may result in the vector increasing
   // its capacity several times.
   //
   for ( int i = 0; i < 100; ++i )
   {
      vec.push_back( i );
   }
}

void goodPushBackManyNumbers( vector< int >& vec )
{
   // This code cleverly preallocates so the vector only
   // increases its capacity once.
   //
   vec.reserve( vec.size() + 100 );

   for ( int i = 0; i < 100; ++i )
   {
      vec.push_back( i );
   }
}
```

Reference Effective STL Items 14, 30;

Rule 17.11 When passing vector types to C style functions use '&v[0]'.

Justification The STL class vector is designed to be usable as a C style array. The elements in a non-empty vector are guaranteed to be stored contiguously so it is possible to use the address of the first element in the container as a pointer to an array of elements. The best way to do this is by '&v[0]' where v is a vector of some object with a C compatible type. Other methods of treating a vector as an array are implementation defined and not portable.

```
extern "C" void functionTakingArrayOfInt( int i[] );
extern "C" void functionTakingPointerToArrayOfInt( int* pvi );

void goodWayToUseCFunctionWithVector( vector< int >& vec )
{
   assert( false == vec.empty() && "this doesnt work with empty vectors!"
);
   functionTakingArrayOfInt( &vec[ 0 ] );            // ok
   functionTakingPointerToArrayOfInt( &vec[ 0 ] );   // ok
}

void badWayToUseCFunctionWithVector( vector< int >& vec )
{
   functionTakingArrayOfInt( vec.begin() );    // may not work as intended!
   functionTakingArrayOfInt( &vec.front() );   // bad! front returns by
value!
}
```

Reference Effective STL Item 16;

Rule 17.12 Only use STL string's member c_str to get a const char* to use with legacy functions.

Justification The c_str method is defined to return a valid, null terminated C style string. Other methods of getting a C style representation are implementation defined and not portable.

Reference Effective STL Item 16;

Rule 17.13 Do not use vector<bool>.

Justification vector<bool> does not conform to the requirements of a container and does not work as expected in all STL algorithms.

In particular &v[0] does not return a contiguous array of bools as it does for other vector types.

Reference Effective STL Item 13;ISO C++ 23.1;

Rule 17.14 Return false for equivalent values in relational predicates.

Justification Sorted containers, and algorithms that operate on sorted containers, require comparison predicates that define the sort order of the elements. These predicates are used to test if elements are equal, they do so by checking that neither element preceeds the other in the sort order.

Returning true from a comparison predicate for equivalent elements means the container will never detect that elements are equal, resulting in an invalid state.

```
// Potential algorithm determining equivalent elements for sorted
// containers.
//
bool areElementsEqual( T& a, T& b )
{
   // pred is a comparison predicate that defines the sort order
```

```
   // of a & b.
   //
   if ( !pred( a, b ) && !pred( b, a ) )
   {
      // a and b are equivalent
   }
}
```

See also Rule 17.15

Reference Effective STL Item 21;

Rule 17.15 Never modify the key part of a set or multiset element.

Justification sets and multisets sort elements as they are inserted into the container, therefore any change to an element that affects its sort position will corrupt the container and result in very hard to find bugs.

See also Rule 17.14

Reference Effective STL Item 22;

Guideline 17.16 Minimise mixing of iterator types.

Justification Iterator types are implementation defined. Portability issues may arise as different STL implementations may have different operations defined for particular iterators.

Efficiency may suffer where different iterator types are used as operands in operator expressions. Potentially the operator is a function call for which one or both of the iterators must undergo a conversion.

Certain container member functions may not be called as they only accept the plain iterator type as a parameter.

```
// May or may not be a member in some implementations.
//
template< typename T >
bool operator== ( vector< T >::const_iterator& lhs,
                  vector< T >::const_iterator& rhs );

void bar()
{
   vector< int > v;
   vector< int >::iterator lhs = v.begin();
   vector< int >::const_iterator rhs = v.end();

   // Should operator== be implemented
   // as a member of const_iterator then
   // this this code will not compile.
   //
   // rhs implicitly converted
   // to const_iterator followed by
   // function call to operator ==.
   //
   if ( lhs == rhs )
   {}
}

void foo( vector< int >& v,
          vector< int >::const_iterator& iter )
{
   // Error cannot convert from
   // const_iterator to iterator.
   //
   v.insert( iter, 10 );
}
```

Reference Effective STL Item 26;

Rule 17.17 The result of a predicate should depend only on its parameters.

Justification For certain algorithms there is no requirement that the order of evaluation, or even that the same predicate object be used, when iterating through a container. A predicate should not be dependent on the order of evaluation; it should return the same result for an element regardless of previous calls or any external state.

In addition, algorithms can copy predicates so there is no guarantee that the state of the predicate will be maintained.

By declaring operator() const it is explicit that the state of the predicate is not modified by calling the function.

```
class Bad_Predicate
{
public:
   Bad_Predicate() : m_count( 0 ) {}
   bool operator()( const int& )
   {
      return ++m_count == 5;
   }
private:
   int m_count;
};

// Irrespective of the number of elements in the deque m_count may never
// reach 5 as the predicate might be copied.
//
bool is_large_deque( std::deque< int >& d )
{
   return d.end() !=
          std::find_if( d.begin(), d.end(), Bad_Predicate() );
}
```

Reference Effective STL Item 39;

Guideline 17.18 Use STL algorithms rather than hand-written loops.

Justification Implementations may optimize algorithms for particular container types to improve efficiency. Code using algorithms is generally clearer, more straightforward, less error prone, and easier to maintain.

See also Rule 17.19

Reference Effective STL Item 43;

Rule 17.19 Use container member functions rather than algorithms with the same name.

Justification Where particular container operations are implemented as members these members should be used instead of the generic algorithm. Member implementations can take advantage of internal container structure and this leads to a more efficient implementation.

```
void foo( std::set< int >& s )
{
   std::set< int >::iterator iter;

   // Generally std::find cannot take advantage of the structure
   // of the container it operates on and executes with linear
   // complexity.
   //
   iter = std::find( s.begin(), s.end(), 10 );

   // set.find takes advantage of the structure of the set and
```

```
   // executes with logarithmic complexity.
   //
   iter = s.find( 10 );
}
```

See also Guideline 17.18

Reference Effective STL Item 44;

Rule 17.20 Directly include necessary STL headers.

Justification Some implementations may include extra STL headers not explicitly specified by the standard. Code that is dependent on these indirect inclusions and does not directly include the appropriate header in the source file will be non portable.

```
#include <vector>

void foo()
{
   // May work with some STL implementations which include
   // < string > in < vector >
   //
   std::string s;
}
```

Reference Effective STL Item 48;

Guideline 17.21 Minimise use of the Standard Template Library 'auto_ptr'.

Justification 'auto_ptr' has destructive copy semantics which may be non-intuitive and can lead to erroneous usage.

```
void foo( auto_ptr<int> ai );

void bar()
{
   auto_ptr<int> ai( new int );
   foo( ai );   // destructive copy
   *ai = 10;    // error ai no longer exists
}
```

Exception If you use 'auto_ptr' take note of the following:
- Do not use auto_ptr with array types.
- Only use auto_ptr with dynamically allocated variables.
- Be aware that implicit conversions can take place between auto_ptrs of different types where a conversion exists for the underlying pointer types.
- Use auto_ptr where "ownership-transfer semantics" are required, for example do not use auto_ptr where you need two pointers to the same object concurrently.

18 Future Direction of Standard

This section provides some guidance on the future direction of this standard as this may affect the way you currently program. It lists Rules and Guidelines which are marked for review in subsequent editions of this standard. This means that items mentioned here may be promoted to Rules, demoted to Guidelines or dropped altogether.

No issues in this release.

Glossary

Functor	Object created from a functor class. Also known as a function object.
Functor class	Any class that overloads the function call operator (operator()) is a functor class.
POD	An acronym for Plain Old Data. A POD-struct is 'an aggregate class that has no non-static data members of type pointer to member, non-POD-struct, non-POD-union (or array of such types) or reference, and has no user-defined copy assignment operator and no user-defined destructor.' A POD-union is 'an aggregate union that has no non-static data members of type pointer to member, non-POD-struct, non-POD-union (or array of such types) or reference, and has no user-defined copy assignment operator and no user-defined destructor.' A POD-class is a class that is either a POD-struct or a POD-union. Arithmetic types, enumeration types, pointer types, and pointer to member types are collectively called scalar types. Scalar types, POD-class types, and arrays of such types are collectively called POD types.
Predicate	A predicate is a function that returns either bool or a type that can be implicitly converted to bool.
Predicate class	A predicate class is a functor class whose operator() function is a predicate, i.e. its operator() returns true or false.

Bibliography

[Stroustrup, 2000]	Bjarne Stroustrup: The C++ Programming Language. Addison-Wesley. 2000
[C++ Standard, 1999]	International Standard ISO/IEC 14882:1998(E) Programming Language C++.
[Effective C++, 1996]	Scott Meyers: Effective C++. Addison-Wesley. 1996
[More Effective C++, 1996]	Scott Meyers: More Effective C++. Addison-Wesley. 1996
[Effective STL, 2001]	Scott Meyers: Effective STL. Addison-Wesley. 2001
[Industrial Strength C++, 1997]	Mats Henricson, Erik Nyquist, Ellemtel Utvecklings AB: Industrial Strength C++. Prentice Hall. 1997
[Exceptional C++, 2000]	Herb Sutter: Exceptional C++. Addison-Wesley. 2000