Programming in C++

Rules and Recommendations

Original translation from Swedish by Joseph Supanich

Table of Contents

1 Introduction

The purpose of this document is to define ***one*** style of programming in C++. The rules and recommendations presented here are not final, but should serve as a basis for continued work with C++. This collection of rules should be seen as a dynamic document; suggestions for improvements are encouraged. A form for requesting new rules or changes to rules has been included as an appendix to this document. Suggestions can also be made via e-mail to one of the following addresses:

`erik.nyquist@eua.ericsson.se`
`mats.henricson@eua.ericsson.se`

Programs that are developed according to these rules and recommendations should be:

- correct
- easy to maintain.

In order to reach these goals, the programs should:

- have a consistent style,
- be easy to read and understand,
- be portable to other architectures,
- be free of common types of errors,
- be maintainable by different programmers.

Questions of design, such as how to design a class or a class hierarchy, are beyond the scope of this document. Recommended books on these subjects are indicated in the chapter entitled “References”.

In order to obtain insight into how to effectively deal with the most difficult aspects of C++, the examples of code which are provided should be carefully studied. C++ is a difficult language in which there may be a very fine line between a feature and a bug. This places a large responsibility upon the programmer. In the same way as for C, C++ allows a programmer to write compact and, in some sense, unreadable code.

Code written in **`bold`** type is meant to serve as a warning. The examples often include class definitions having the format "**`class <name> {};`**". These are included so that the examples may be compiled; it is not recommended that class definitions be written in this way. In order to make the code more compact, the examples provided do not always follow the rules. In such cases, the rule which is broken is indicated.

Many different C++ implementations are in use today. Most are based on the C++ Language System by AT&T. The component of this product which translates C++ code to C is called Cfront. The different versions of Cfront (2.0, 2.1 & 3.0 are currently in use) are referred to in order to point out the differences between different implementations.

Rule 0 Every time a <u>rule</u> is broken, this must be clearly documented.

2 Terminology

1 An ***identifier*** is a name which is used to refer to a variable, constant, function or type in C++. When necessary, an identifier may have an internal structure which consists of a prefix, a name, and a suffix (in that order).

2 A ***class*** is a user-defined data type which consists of data elements and functions which operate on that data. In C++, this may be declared as a **`class`**; it may also be declared as a **`struct`** or a **`union`**. Data defined in a class is called ***member data*** and functions defined in a class are called ***member functions.***

3 A **`class`**/**`struct`**/**`union`** is said to be an ***abstract data type*** if it does not have any public or protected member data.

4 A ***structure*** is a user-defined type for which only public data is specified.

5 ***Public members*** of a class are member data and member functions which are everywhere accessible by specifying an instance of the class and the name.

6 ***Protected members*** of a class are member data and member functions which are accessible by specifying the name within member functions of derived classes.

7 A ***class template*** defines a family of classes. A new class may be created from a class template by providing values for a number of arguments. These values may be names of types or constant expressions.

8 A ***function template*** defines a family of functions. A new function may be created from a function template by providing values for a number of arguments. These values may be names of types or constant expressions.

9 An ***enumeration type*** is an explicitly declared set of symbolic integral constants. In C++ it is declared as an **`enum`**.

10 A ***typedef*** is another name for a data type, specified in C++ using a **`typedef`** declaration.

11 A ***reference*** is another name for a given variable. In C++, the ‘address of’ (`&`) operator is used immediately after the data type to indicate that the declared variable, constant, or function argument is a reference.

12 A ***macro*** is a name for a text string which is defined in a **`#define`** statement. When this name appears in source code, the compiler replaces it with the defined text string.

13 A ***constructor*** is a function which initializes an object.

14 A ***copy constructor*** is a constructor in which the first argument is a reference to an object that has the same type as the object to be initialized.

15 A ***default constructor*** is a constructor which needs no arguments.

16 An ***overloaded function name*** is a name which is used for two or more functions or member functions having different types[1].

17 An ***overridden*** member function is a member function in a base class which is re-defined in a derived class. Such a member function is declared **`virtual`**.

18 A ***pre-defined data type*** is a type which is defined in the language itself, such as **`int`**.

19 A ***user-defined data type*** is a type which is defined by a programmer in a **`class`**, **`struct`**, **`union`**, **`enum`**, or **`typedef`** definition or as an instantiation of a class template.

20 A ***pure virtual function*** is a member function for which no definition is provided. Pure virtual functions are specified in ***abstract base classes*** and must be defined (overridden) in derived classes.

21 An ***accessor*** is a function which returns the value of a data member.

22 A ***forwarding function*** is a function which does nothing more than call another function.

23 A ***constant member function*** is a function which may not modify data members.

24 An ***exception*** is a run-time program anomaly that is detected in a function or member function. Exception handling provides for the uniform management of exceptions. When an exception is detected, it is ***thrown*** (using a **`throw`** expression) to the exception handler.

25 A ***catch clause*** is code that is executed when an exception of a given type is raised. The definition of an exception handler begins with the keyword **`catch`**.

26 An ***abstract base class*** is a class from which no objects may be created; it is only used as a base class for the derivation of other classes. A class is abstract if it includes at least one member function that is declared as ***pure virtual***.

27 An ***iterator*** is an object which, when invoked, returns the *next* object from a collection of objects.

28 The ***scope*** of a name refers to the context[2] in which it is visible.

29 A ***compilation unit*** is the source code (after preprocessing) that is submitted to a compiler for compilation (including syntax checking).

---

1. The type of a function is given by its return type and the type of its arguments.
2. Context, here, means the functions or blocks in which a given variable name can be used.

3 General Recommendations

Rec. 1 Optimize code only if you ***know*** that you have a performance problem. Think twice before you begin.

Rec. 2 If you use a C++ compiler that is based on Cfront, always compile with the +w flag set to eliminate as many warnings as possible.

Various tests are said to have demonstrated that programmers generally spend a lot of time optimizing code that is never executed. If your program is too slow, use **`gprof++`** or an equivalent tool to determine the exact nature of the problem before beginning to optimize.

Code that is accepted by a compiler is not always correct (in accordance with the definition of the C++ language). Two reasons for this are that changes are made in the language and that compilers may contain bugs. In the short term, very little can be done about the latter. In order to reduce the amount of code that must be rewritten for each new compiler release, it is common to let the compiler provide warnings instead of reporting errors for such code until the next major release. Cfront provides the +w flag to direct the compiler to give warnings for these types of language changes.

4 Source Code in Files

4.1 Structure of Code

Rule 1 Include files in C++ always have the file name extension ".hh".

Rule 2 Implementation files in C++ always have the file name extension ".cc".

Rule 3 Inline definition files always have the file name extension ".icc".

Rec. 3 An include file should not contain more than one class definition.

Rec. 4 Divide up the definitions of member functions or functions into as many files as possible.

Rec. 5 Place machine-dependent code in a special file so that it may be easily located when porting code from one machine to another.

The purpose of these conventions is to provide a uniform interpretation of file names. One reason for this is that it is easier to make tools which base their behaviour on the file name extension.

There are two kinds of include files in C++: those which contain code that is accepted by both ANSI-C and C++ compilers and those which contain code that is only accepted by C++ compilers. It is appropriate to distinguish between the two in order to avoid unpleasant compilation errors (from using the wrong kind of include file).

If a ".cc" file contains a large number of function definitions, the object file produced by the compiler may be unnecessarily large. In order to obtain the smallest possible executable files, it is necessary to have a separate file for each function definition. This is because the standard UNIX linker **`ld`** links all functions in an object file even if only one of them is actually used. It is especially important to remember that virtual functions are always linked[1]. On the other hand, there are problems in managing a large number of files, since sufficiently powerful tools are not currently available. Also, the time necessary to compile a program consisting of a large number of files is longer.

Some debuggers cannot debug inline functions. By placing inline functions in a separate file and by including that file in the implementation file, thus treating the inline functions as ordinary functions, it is possible to debug the functions while testing the program. For this to work some special preprocessor techniques must be used[2]. The inline definition file must not be included by the include file for the class and the keyword 'inline' must be removed.

When tools for managing C++ code are not available, it is much easier for those who use and maintain classes if there is only one class definition in each file and if implementations of member functions

1. Compilers based on Cfront refer to these via so-called virtual tables.
2. See Example 1!

in different classes are not present in the same file.

Exception to Rule 1: Include files which contain code that is accepted by both C and C++ compilers should have the file name extension “.h”.

Exception to Rule 2: When using a compiler that does not accept the extension “.cc”, the extension “.C” is used instead.

Exception to Rule 3: No exceptions.

Example 1 Inline definitions in a separate file for conditional compilation

```
// AnyClass.hh
#ifndef OUTLINE
#include "AnyClass.icc"
#endif

//AnyClass.cc
#ifdef OUTLINE
#define inline
#include "AnyClass.icc"
#undef inline
#endif
```

4.2 Naming Files

Rec. 6 Always give a file a name that is unique in as large a context as possible.

Rec. 7 An include file for a class should have a file name of the form <class name> + extension. Use uppercase and lowercase letters in the same way as in the source code.

There is always a risk for name collisions when the file name is part of identifier names that are generated by the compiler. This is a problem in using any Cfront-based compiler.

AT&T’s Cfront-based compiler creates two functions for every file in order to call constructors and destructors of static objects in the proper order. These functions are named:
char __sti__file_cc___Fv(); //filename is file.cc
char __std__file_cc___Fv(); //filename is file.cc
It is easily understood that if a program has two files with the same name but in different subdirectories, there will be name collisions between the functions generated above.

Since class names must generally be unique within a large context, it is appropriate to utilize this characteristic when naming its include file. This convention makes it easy to locate a class definition using a file-based tool.

4.3 Comments

Rule 4 Every file that contains source code must be documented with an introductory comment that provides information on the file name and its contents.

Rule 5 All files must include copyright information.

Rule 6 All comments are to be written in English.

Rec. 8 Write some descriptive comments before every function.

Rec. 9 Use `//` for ***comments.***

It is necessary to document source code. This should be compact and easy to find. By properly choosing names for variables, functions and classes and by properly structuring the code, there is less need for comments within the code.

Note that comments in include files are meant for the users of classes, while comments in implementation files are meant for those who maintain the classes.

All our code must be copyright marked. If the code has been developed over a period of years, each year must be stated.

The standardization of comments makes it possible to automatically generate **`man`**-pages from source code. This may be used to keep source code and documentation together until adequate tools for information management are available.

Comments are often said to be either ***strategic*** or ***tactical***. A strategic comment describes what a function or section of code is intended to do, and is placed before this code. A tactical comment describes what a single line of code is intended to do, and is placed, if possible, at the end of this line. Unfortunately, too many tactical comments can make code unreadable. For this reason, it is recommended to primarily use strategic comments, unless trying to explain very complicated code.

If the characters `//` are consistently used for writing comments, then the combination `/* */` may be used to make comments out of entire sections of code during the development and debugging phases. C++, however, does not allow comments to be nested using `/* */`.

Exception to Rule 4: No exceptions.

Exception to Rule 5: No exceptions.

Exception to Rule 6: No exceptions.

Example 2 Documentation of a file

```
//
//  File:        test.cc
//  Description: This is a test program
//  Rev:         A
//  Created:     Thur. Oct 31, 1991, 12:30:14
//  Author:      Erik Nyquist
//  mail:        erik.nyquist@eua.ericsson.se
//
//  Copyright Ellemtel Utvecklings AB 1991
//  BOX 1505
//  125 25 ALVSJO
//  SWEDEN
//  tel int + 46 8 727 3000
//
//  The copyright to the computer program(s) herein
//  is the property of Ellemtel Utvecklings AB, Sweden.
//  The program(s) may be used and/or copied only with
//  the written permission of Ellemtel Utvecklings AB
//  or in accordance with the terms and conditions
//  stipulated in the agreement/contract under which
//  the program(s) have been supplied.
//
```

Example 3 Strategic and Tactical Comments

```
// THE NEXT TWO LINES ARE STRATEGIC COMMENTS
// This function does some complicated things. It works like this:
//  blah-blah-blah ...
int
insanelyGreatAndComplicatedFunction( int i )
{
   int index = i++ + ++i * i-- - --i;  // THIS IS A TACTICAL COMMENT

   return index;
}
```

4.4 Include Files

Rule 7 Every ***include file*** must contain a mechanism that prevents multiple inclusions of the file.

Rule 8 When the following kinds of definitions are used (in implementation files or in other include files), they must be included as separate include files:

- classes that are used as ***base classe***s,
- classes that are used as ***member variables***,
- classes that appear as ***return types*** or as ***argument types*** in function/member function prototypes.
- ***function prototypes*** for functions/member functions used in ***inline member functions*** that are defined in the file.

Rule 9 Definitions of classes that are only accessed via pointers (`*`) or references (`&`) shall ***not*** be included as include files.

Rule 10 **Never** specify relative UNIX names in `#include` directives.

Rule 11 Every ***implementation file*** is to include the relevant files that contain:

- declarations of types and functions used in the functions that are implemented in the file.
- declarations of ***variables*** and ***member functions*** used in the functions that are implemented in the file.

Rec. 10 Use the directive `#include "filename.hh"` for user-prepared include files.

Rec. 11 Use the directive `#include <filename.hh>` for include files from libraries.

Rec. 12 Every implementation file should declare a local constant string that describes the file so the UNIX command `what` can be used to obtain information on the file revision.

Rec. 13 Never include other files in an “.icc” file.

The easiest way to avoid multiple includes of files is by using an `#ifndef/#define` block in the beginning of the file and an `#endif` at the end of the file.

The number of files included should be minimized. If a file is included in an include file, then every implementation file that includes the second include file must be re-compiled whenever the first file is modified. A simple modification in one include file can make it necessary to re-compile a large number of files.

When only referring to pointers or references to types defined in a file, it is often not necessary to include that file. It may suffice to use a forward declaration to inform the compiler that the class exists. Another alternative is to precede each declaration of a pointer to the class with the keyword `class`.

True portable code is independent of the underlying operating system. For this reason, relative UNIX search paths should be avoided when including files. The processing of such search paths depends on the compiler and UNIX should not be taken for granted. Instead, search paths should be provided in ‘make’ files as options for the compiler.

If a file only contains information that is only needed in an implementation file, that file should not be included in another include file. Otherwise, when the information is no longer needed in the implementation file, it may be necessary to re-compile each file that uses the interface defined in the include file.

Every C++ course teaches the difference between the include directives for user-prepared and for library include files. If the file name is bracketed between “<” and “>”, the preprocessor will not search for the file in the default directory. This reduces the risk of unintended name collisions between user-prepared and library include files.

By declaring a local constant string, the compiler becomes self-identifying. This may be used to easily determine the version of the program that is used. The string must begin with the characters **`@(#)`** to be read by the UNIX **`what`** command.

Exception to Rule 7: No exceptions.

Exception to Rule 8: No exceptions.

Exception to Rule 9: No exceptions.

Exception to Rule 10: No exceptions.

Exception to Rule 11: No exceptions.

Example 4 Technique for preventing multiple inclusion of an include file

```
#ifndef FOO_HH
#define FOO_HH

// The rest of the file

#endif
```

Example 5 Never use explicit UNIX path names

```
// NOT RECOMMENDED
#include <../include/fnutt.h>

// NOT GUARANTEED TO WORK
#include <sys/socket.h>
```

Example 6 **Local constant string for identifying implementation files.**

```
static const char* sccsid =
"@(#) Exception.cc, rev. A, Copyright Ellemtel Utvecklings AB 1991";
```

Example 7 **Include file for the class PackableString**

```
// file: PackableString.hh
#ifndef PACKABLESTRING_HH
#define PACKABLESTRING_HH

#include "String.hh"
#include "Packable.hh"

// It is not necessary to extern-declare class Buffer when
// each pointer declaration specifies the keyword class as shown below.
// An explicit extern-declaration makes the code easier to
// understand.

extern class Buffer;

class PackableString : public String, public Packable
{
   public:
      PackableString( const String& s );
      class Buffer* put( class Buffer* outbuffer );
      // ...
};

#endif
```

Example 8 **Implementation file for the class PackableString**

```
// PackableString.cc

#include "PackableString.hh"

// To be able to use Buffer-instances, Buffer.hh MUST be included.

#include "Buffer.hh"

Buffer*
PackableString::put( Buffer* outbuffer )
{
   // ...
}
```

5 Assigning Names

Rule 12 The identifier of every globally visible class, enumeration type, type definition, function, constant, and variable in a class library is to begin with a prefix that is ***unique for the library.***

Rule 13 The names of variables, constants, and functions are to begin with a ***lowercase*** letter.

Rule 14 The names of abstract data types, structures, **`typedefs`**, and enumerated types are to begin with an ***uppercase*** letter.

Rule 15 In names which consist of more than one word, ***the words are written together*** and each word that follows the first is begun with an uppercase letter.

Rule 16 Do not use identifiers which begin with one or two underscores (‘_’ or ‘__’).

Rule 17 A name that begins with an uppercase letter is to appear directly after its prefix.

Rule 18 A name that begins with a lowercase letter is to be separated from its prefix using an underscore (‘_’).

Rule 19 A name is to be separated from its suffix using an underscore (‘_’).

Rec. 14 Do not use typenames that differ only by the use of uppercase and lowercase letters.

Rec. 15 Names should not include abbreviations that are not generally accepted.

Rec. 16 A variable with a large scope should have a long name.

Rec. 17 Choose variable names that suggest the usage.

Rec. 18 Write code in a way that makes it easy to change the prefix for global identifiers.

Rec. 19 Encapsulate global variables and constants, enumerated types, and typedefs in a class.

In this chapter, it is important to distinguish between identifiers and names[1]. The name is that part of an identifier that shows its meaning. An identifier consists of a prefix, a name, and a suffix (in that order). The prefix and the suffix are optional. A suffix is only used by tools that generate C++ code, to avoid name collisions with user-written C++ code and is not given further consideration here.

It is recommended identifiers not be extremely long, to reduce the risk for name collisions when using tools that truncate long identifiers.

*The Unix command **`ar`** truncates file names that are longer than 15 characters.*

Cfront normally modifies generated C-identifiers that are longer than 31 characters by truncating them and adding a hash value that is generated from the truncated part of the string.

The use of two underscores (‘__’) in identifiers is reserved for the compiler’s internal use according to the ANSI-C standard.

1. See terminology 1!

Underscores ('_') are often used in names of library functions (such as " _main" and "_exit"). In order to avoid collisions, do not begin an identifier with an underscore.

One rule of thumb is that a name which cannot be pronounced is a bad name. A long name is normally better than a short, cryptic name, but the truncation problem must be taken into consideration. Abbreviations can always be misunderstood. Global variables, functions and constants ought to have long enough names to avoid name conflicts, but not too long.

Classes should be named so that "object.function" is easy to read and appears to be logical.

There are many class libraries available for purchase and there may be tens of thousands of classes in a large project!! Because of this, it is important to be careful that name collisions do not occur. One way of preventing collisions is to have strict rules for assigning names to globally visible objects (such as our use of a prefix). In this way, classes from several different class libraries may be used at the same time.

Names for the following types of objects are to be prefixed:
- Type names (classes, **`typedef`**s, **`enum`**s, **`struct`**s, **`union`**s, etc.)
- Global variables and constants
- Function names (not member functions names)
- Preprocessor macros (**`#define`**)

The use of prefixes can sometimes be avoided by using a class to limit the scope of the name. Static variables in a class should be used instead of global variables and constants, enumerated data types, and typedefs. Although nested classes may be used in C++, these give rise to too many questions (in connection with the language definition) to be able to recommend their use.

Exception to Rule 12: No exceptions.

Exception to Rule 13: No exceptions. (At times, an identifier begins with an abbreviation written in uppercase letters, to emphasize the way in which the name is used. Such an abbreviation is considered to be a prefix).

Exception to Rule 14: If the last letter in a word is in uppercase, an underscore is to be used as a word separator.

Exception to Rule 15: No exceptions.

Exception to Rule 16: No exceptions.

Exception to Rule 17: No exceptions.

Exception to Rule 18: No exceptions.

Exception to Rule 19: No exceptions.

Example 9 Exception using compound names

```
const char* functionTitle = "EUA_Special";
int currentIO_Stream = 1; // Last Character in currentIO is in uppercase!
```

Example 10 Choice of names

```
int groupID;                     // instead of grpID
int nameLength;                  // instead of namLn
PrinterStatus resetPrinter;      // instead of rstprt
```

Example 11 Ambiguous names

```
void termProcess();              // Terminate process or
                                 // terminal process?
```

Example 12 Names having numeric characters can cause errors which are difficult to locate.

```
int I0 = 13;                     // Names with digits can be
int IO = I0;                     // difficult to read.
```

Example 13 Definition of a class in the class library Emc2

```
class Emc2Class
{
   public:
      Emc2Class();    // Default constructor
      // ...
   private:
      int id;
      // ...
};
```

Example 14 **One way to avoid global functions and classes**

```
// Instead of declaring:
void Emc2_myFunc1();
void Emc2_myFunc2();
class Emc2MyClass { /* ... */ };

// Encapsulate the functions using an abstract class:
class Emc2
{
   public:
      static void myFunc1();
      static void myFunc2();
      class MyClass { /* ... */ };
   private:
      virtual dummy() = 0;   // Trick to make the class abstract
};

// Now, functions and classes may be accessed by using the scope-operator:
Emc2::myFunc1();
Emc2::myFunc2();
Emc2::MyClass myObject;
```

6 Style

6.1 Classes

Rule 20 The public, protected, and private sections of a class are to be declared in that order (the public section is declared before the protected section which is declared before the private section).

Rule 21 No member functions are to be defined within the class definition.

By placing the *public* section first, everything that is of interest to a user is gathered in the beginning of the class definition. The *protected* section may be of interest to designers when considering inheriting from the class. The *private* section contains details that should have the least general interest.

A member function that is defined within a class definition automatically becomes inline. Class definitions are less compact and more difficult to read when they include definitions of member functions. It is easier for an inline member function to become an ordinary member function if the definition of the inline function is placed outside of the class definition. This rule will be in effect at least as long as traditional text editors are used.

A similar problem is that members are private if no access specifier is explicitly given. This problem is avoided by following Rule 20.

Exception to Rule 20: No exceptions.

Exception to Rule 21: No exceptions.

Example 15 A class definition in accordance with the style rules

```
class String : private Object
{
   public:
      String();                     // Default constructor
      String( const String& s );    // Copy constructor
      unsigned length() const;
      // ...
   protected:
      int checkIndex( unsigned index ) const;
      // ...
   private:
      unsigned noOfChars;
      // ...
};
```

Example 16 No definitions of member functions within the class definition (See Example 1)

```
// Instead of writing like this:
class String
{
   public:
      int length() const // No !!
         {
            return len;
         }
      // ...
   private:
      int len;
};

// Do it this way:
class String
{
   public:
      int length() const;
      // ...
   private:
      int len;
};

inline
int
String::length() const
{
   return len;
}
```

6.2 Functions

Rec. 20 Always provide the ***return type*** of a function explicitly.

Rec. 21 When declaring functions, the leading parenthesis and the first argument (if any) are to be written on the ***same line*** as the function name. If space permits, other arguments and the closing parenthesis may also be written on the same line as the function name. Otherwise, each additional argument is to be written on a separate line (with the closing parenthesis directly after the last argument).

Rec. 22 In a function definition, the ***return type*** of the function should be written on a separate line directly above the function name.

Rec. 23 Always write the left parenthesis directly after a function name.

If no return type is explicitly provided for a function, it is, by default, an `int`. It is recommended to always provide the return type explicitly, to increase the readability of the code. By defining the return type on a separate line directly above the function definition, the function name is more easily seen.

The other recommendations are meant to give a uniform appearance to the code that is written. Until such time as formatting tools are available, programmers should follow these guidelines.

Example 17 The left parenthesis always directly after the function name

```
void foo ();     // No!!
void foo();      // Better
```

Example 18 Right and wrong ways of declaring formal arguments for a function (in function definition)

```
// Right:
int
myComplicatedFunction( unsigned unsignedValue,
                       int intValue,
                       char* charPointerValue,
                       int* intPointerValue,
                       myClass* myClassPointerValue,
                       unsigned* unsignedPointerValue );

// Wrong:
int
myComplicatedFunction( unsigned unsignedValue, int intValue,
char* charPointerValue, int* intPointerValue, myClass* myClassPointerValue,
unsigned* unsignedPointerValue );
```

6.3 Compound Statements

Rec. 24 Braces (“{}”) which enclose a block are to be placed in the same column, on separate lines directly before and after the block.

The placement of braces seems to have been the subject of the greatest debate concerning the appearance of both C and C++ code. We recommend the style which, in our opinion, gives the most readable code. Other styles may well provide more compact code.

6.4 Flow Control Statements

Rec. 25 The flow control primitives **`if`**, **`else`**, **`while`**, **`for`** and **`do`** should be followed by a ***block***, even if it is an empty block.

At times, everything that is to be done in a loop may be easily written on one line in the loop statement itself. It may then be tempting to conclude the statement with a semicolon at the end of the line. This may lead to misunderstanding since, when reading the code, it is easy to miss such a semicolon. It seems to be better, in such cases, to place an empty block after the statement to make completely clear what the code is doing.

Example 19 Flow control structure without statements

```
// No block at all - No!
while ( /* Something */ );

// Empty block - better!
while ( /* Something */ )
{
   // Empty !
}
```

6.5 Pointers and References

Rec. 26 The dereference operator '**`*`**' and the address-of operator '**`&`**' should be directly connected with the ***type names*** in declarations and definitions.

The characters '*' and '&' should be written together with the types of variables instead of with the names of variables in order to emphasize that they are part of the type definition. Instead of saying that **`*i`** is an **`int`**, say that **`i`** is an **`int*`**.

Traditionally, C recommendations indicate that '*' should be written together with the variable name, since this reduces the probability of making a mistake when declaring several variables in the same declaration statement (the operator '*' only applies to the variable on which it operates). Since the declaration of several variables in the same statement is not recommended, however, such a advice is unneeded.

Example 20 * and & together with the type

```
char*
Object::asString()
{
   // Something
};

char* userName = 0;
int sfBook = 42;
int&  anIntRef = sfBook;
```

Example 21 Declaration of several variables in the same statement

```
// NOT RECOMMENDED
char* i,j; // i is declared pointer to char, while j is declared char
```

6.6 Miscellaneous

Rec. 27 Do not use ***spaces*** around '**`.`**' or '**`->`**', nor between unary operators and operands.

Rec. 28 Use the c++ mode in GNU Emacs to format code.

In our opinion, code is more readable if spaces are not used around the . or -> operators. The same applies to unary operators (those that operate on one operand), since a space may give the impression that the unary operand is actually a binary operator.

Ordinary spaces should be used instead of tabs. Since different editors treat tab characters differently, the work in perfecting a layout may have been wasted if another editor is later used. Tab characters can be removed using the UNIX command **`expand`**. One alternative is to format code using the editor GNU Emacs.

We recommend that everyone use GNU Emacs to format code. Code will then have a uniform appearance regardless of who has written it.

7 Classes

7.1 Considerations Regarding Access Rights

Rule 22 Never specify ***public*** or ***protected member data*** in a class.

The use of public variables is discouraged for the following reasons:

1. A public variable represents a violation of one of the basic principles of object-oriented programming, namely, encapsulation of data. For example, if there is a class of the type BankAccount, in which account_balance is a public variable, the value of this variable may be changed by any user of the class. However, if the variable has been declared private, its value may be changed only by the member functions of the class[1].
2. An arbitrary function in a program can change public data which may lead to errors that are difficult to locate.
3. If public data is avoided, its internal representation may be changed without users of the class having to modify their code. A principle of class design is to maintain the stability of the public interface of the class. The implementation of a class should not be a concern for its users.

The use of protected variables in a class are not recommended, since its variables become visible to its derived classes. The names of types or variables in a base class may then not be changed since the derived classes may depend on them. If a derived class, for some reason, must access data in its base class, one solution may be to make a special **`protected`** interface in the base class, containing functions which return private data. This solution would not imply any degradation of performance if the functions are defined inline.

The use of ***struct***s is also discouraged since these only contain public data. In interfaces with other languages (such as C), it may, however, be necessary to use structs.

Exception to Rule 22: In interfaces with other languages (such as C), it may be necessary to use **`structs`** having public data.

1. Not completely true. If a class has a member function which returns a reference to a data member, variables may be modified. This is avoided by following Rule 29.

Example 22 **The correct way to encapsulate data so that future changes are possible.**

This shows why member functions should be used to access data (instead of using direct references). This usage provides long term advantages, since internal data in a class may be changed without having to modify interfaces and to re-write the code which uses them.

```
// Original class:

class Symbol {};
class OldSymbol : public Symbol {};

class Priority
{
   public:
      // returns pd
      int priority();

      // returns symbol
      class Symbol* getSymbol() const;
      // ...
   private:
      int pd;
      OldSymbol symbol;
};


// Modified class:
// The programmer has chosen to change the private data from an int
// to an enum. A user of the class 'Priority' does not have to change
// any code, since the enum return-value from the member function
// priority() is automatically converted to an int.

class Symbol {};
class NewSymbol : public Symbol {};
enum Priority { low, high, urgent };

class Priority
{
   public:
      // Interface intact through implicit cast, returns priority_data
      Priority priority();

      // Interface intact, object of new subclass to symbol returned
      class Symbol* getSymbol() const;
      // ...

   private:
      Priority priority_data; // New representation/name of internal data
      NewSymbol symbol;
};
```

7.2 Inline Functions

Rec. 29 Access functions are to be inline.

Rec. 30 Forwarding functions are to be inline.

Rec. 31 Constructors and destructors must not be inline.

The normal reason for declaring a function **`inline`** is to improve its performance.

Small functions, such as access functions, which return the value of a member of the class and so-called forwarding functions which invoke another function should normally be **`inline`**.

Correct usage of **`inline`** functions may also lead to reduced size of code.

Warning: functions, which invoke other **`inline`** functions, often become too complex for the complier to be able to make them **`inline`** despite their apparent smallness.

This problem is especially common with constructors and destructors. A constructor always invokes the constructors of its base classes and member data before executing its own code. Always avoid **`inline`** constructors and destructors!

7.3 Friends

Rec. 32 ***Friends*** of a class should be used to provide additional functions that are best kept outside of the class.

Operations on an object are sometimes provided by a collection of classes and functions.

A friend is a nonmember of a class, that has access to the nonpublic members of the class. Friends offer an orderly way of getting around data encapsulation for a class. A friend class can be advantageously used to provide functions which require data that is not normally needed by the class.

Suppose there is a list class which needs a pointer to an internal list element in order to iterate through the class. This pointer is not needed for other operations on the list. There may then be reason, such as obtaining smaller list objects, for an list object not to store a pointer to the current list element and instead to create an iterator, containing such a pointer, when it is needed.

One problem with this solution is that the iterator class normally does not have access to the data structures which are used to represent the list (since we also recommend private member data).

By declaring the iterator class as a friend, this problem is avoided without violating data encapsulation.

Friends are good if used properly. However, the use of many friends can indicate that the modularity of the system is poor.

7.4 *const* Member Functions

Rule 23 A member function that does not affect the state of an object (its instance variables) is to be declared **`const`**.

Rule 24 If the behaviour of an object is dependent on data outside the object, this data is not to be modified by const member functions.

Member functions declared as **`const`** may not modify member data and are the only functions which may be invoked on a **`const`** object. (Such an object is clearly unusable without **`const`** methods). A **`const`** declaration is an excellent insurance that objects will not be modified (mutated) when they should not be. A great advantage that is provided by C++ is the ability to overload functions with respect to their const-ness. (Two member functions may have the same name where one is const and the other is not).

Non-const member functions are sometimes invoked as so-called 'lvalues[1]' (as a location value at which a value may be stored). A const member function may never be invoked as an 'lvalue'.

The behaviour of an object can be affected by data outside the object. Such data must not be modified by a const member function.

At times, it is desirable to modify data in a const object (such a having a cache of data for performance reasons). In order to avoid explicit type conversions from a const type to a non-const type, the only way is to store the information outside the object. (See example 55). This type of data should be seen as external data which does not affect the behaviour of the class.

Exception to Rule 23: No exceptions.

Exception to Rule 24: No exceptions.

Example 23 const-declared access functions to internal data in a class

```
class SpecialAccount : public Account
{
   public:
      int insertMoney();
      // int getAmountOfMoney();  No! Forbids ANY constant object to
      //                          access the amount of money.
      int getAmountOfMoney() const;   // Better!
      // ...
   private:
      int moneyAmount;
};
```

1. See, for example, page 25 in ref. [1]: The Annotated C++ Reference Manual – Bjarne Stroustrup/Margareth Ellis[ARM].

Example 24 **Overloading an operator/function with respect to const-ness**

```
#include <iostream.h>
#include <string.h>
static unsigned const cSize = 1024;
class InternalData {};

class Buffer
{
   public:
      Buffer( char* cp );

      // Inline functions in this class are written compactly so the example
      // may fit on one page. THIS is NOT to be done in practice (See Rule 21).

      // A. non-const member functions: result is an lvalue
      char& operator[]( unsigned index ) { return buffer[index]; }
      InternalData& get() { return data; }

      // B. const member functions: result is not an lvalue
      char operator[]( unsigned index ) const { return buffer[index]; }
      const InternalData& get() const { return data; }

   private:
      char buffer[cSize];
      InternalData data;
};

inline Buffer::Buffer( char* cp )
{
   strncpy( buffer , cp , sizeof( buffer ) );
}

main()
{
   const Buffer cfoo = "peter";  // This is a constant buffer
   Buffer foo = "mary";          // This buffer can change

   foo[2]='c';                   // calls char& Buffer::operator[](unsigned)
   cfoo[2] = 'c'                 // ERROR: cfoo[2] is not an lvalue.

   // cfoo[2] means that Buffer::operator[](unsigned) const is called.

   cout << cfoo[2] << ":" << foo[2] << endl; // OK! Only rvalues are needed

   foo.get() = cfoo.get();
   cfoo.get() = foo.get();       // ERROR: cfoo.get() is not an lvalue
}
```

7.5 Constructors and Destructors

Rule 25 A class which uses “new” to allocate instances managed by the class,[1] must define a ***copy constructor***.

Rule 26 All classes which are used as base classes and which have virtual functions, must define a virtual destructor.

Rec. 33 Avoid the use of global objects in constructors and destructors.

A copy constructor is recommended to avoid surprises when an object is initialized using an object of the same type. If an object manages the allocation and deallocation of an object on the heap (the managing object has a pointer to the object to be created by the class’ constructor), only the value of the pointer will be copied. This can lead to two invocations of the destructor for the same object (on the heap), probably resulting in a run-time error.[2]

The corresponding problem exists for the assignment operator (‘`=`’). See 7.6: Assignment Operators.

If a class, having virtual functions but without virtual destructors, is used as a base class, there may be a surprise if pointers to the class are used. If such a pointer is assigned to an instance of a derived class and if `delete` is then used on this pointer, only the base class’ destructor will be invoked. If the program depends on the derived class’ destructor being invoked, the program will fail.[3]

In connection with the initialization of statically allocated objects, it is not certain that other static objects will be initialized (for example, global objects).[4] This is because the order of initialization of static objects which is defined in various compilation units, is not defined in the language definition. There are ways of avoiding this problem[5], all of which require some extra work.

You must know what you are doing if you invoke virtual functions from a constructor in the class. If virtual functions in a derived class are overridden, the original definition in the base class will still be invoked by the base class’ constructor. Override, then, does not always work when invoking virtual functions in constructors. See Example 30.

Exception to Rule 25: Sometimes, it is desired to let objects in a class share a data area. In such a case, it is not necessary to define a copy constructor. Instead, it is necessary to make sure that this data area is not deallocated as long as there are pointers to it.

Exception to Rule 26: No exceptions.

1. i.e. instances bound to member variables of pointer or reference type that are deallocated by the object.
2. See Example 25 and Example 26.
3. See Example 27.
4. i.e. the static object which was declared external. See Example 28.
5. See Example 29.

Example 25 **Definition of a “dangerous” class not having a copy constructor**

```
#include <string.h>

class String
{
   public:
      String( const char* cp = "");    // Constructor
      ~String();                       // Destructor
      // ...
   private:
      char* sp;
      // ...
};

String::String(const char* cp) : sp( new char[strlen(cp)] )     // Constructor
{
   strcpy(sp,cp);
}

String::~String()    // Destructor
{
   delete sp;
}

// "Dangerous" String class
void
main()
{
   String w1;
   String w2 = w1;
   // WARNING: IN A BITWISE COPY OF w1::sp,
   // THE DESTRUCTOR FOR W1::SP WILL BE CALLED TWICE:
   // FIRST, WHEN w1 IS DESTROYED; AGAIN, WHEN w2 IS DESTROYED.
}
```

Example 26 “Safe” class having copy constructor and default constructor

```
#include <string.h>

class String
{
   public:
      String( const char* cp = "");   // Constructor
      String( const String& sp );     // Copy constructor
      ~String();                      // Destructor
      // ...
   private:
      char* sp;
      // ...
};

String::String( const char* cp ) : sp( new char[strlen(cp)] )    // Constructor
{
   strcpy(sp,cp);
}

String::String( const String& stringA ) : sp( new char[strlen(stringA.sp)] )
{
   strcpy(sp,stringA.sp);
}

String::~String()     // Destructor
{
   delete sp;
}

// “Safe” String class
void
main()
{
   String w1;
   String w2 = w1;   // SAFE COPY: String::String( const String& ) CALLED.
}
```

Example 27 Definitions of classes not having virtual destructors

```
class Fruit
{
   public:
      ~Fruit();     // Forgot to make destructor virtual!!
      // ...
};

class Apple : public Fruit
{
   public:
      ~Apple();     // Destructor
      // ...
};

// "Dangerous" usage of pointer to base class

class FruitBasket
{
   public:
      FruitBasket();      // Create FruitBasket
      ~FruitBasket();     // Delete all fruits
      // ...
      void add(Fruit*);   // Add instance allocated on the free store
      // ...
   private:
      Fruit* storage[42]; // Max 42 fruits stored
      int numberOfStoredFruits;
};

void
FruitBasket::add(Fruit* fp)
{
   // Store pointer to fruit
   storage[numberOfStoredFruits++] = fp;
}

FruitBasket::FruitBasket() : numberOfStoredFruits(0)
{
}

FruitBasket::~FruitBasket()
{
   while (numberOfStoredFruits > 0)
   {
     delete storage[--numberOfStoredFruits]; // Only Fruit::~Fruit is called !!
   }
}
```

Example 28 Dangerous use of static objects in constructors

```
// Hen.hh
class Egg;

class Hen
{
   public:
      Hen();     // Default constructor
      ~Hen();    // Destructor
      // ...
      void makeNewHen(Egg*);
      // ...
};

// Egg.hh

class Egg { };

extern Egg theFirstEgg; // defined in Egg.cc

// FirstHen.hh

class FirstHen : public Hen
{
   public:
      FirstHen();    // Default constructor
      // ...
};

extern FirstHen theFirstHen; // defined in FirstHen.cc

// FirstHen.cc

FirstHen theFirstHen; // FirstHen::FirstHen() called

FirstHen::FirstHen()
{
   // The constructor is risky because theFirstEgg is a global object
   // and may not yet exist when theFirstHen is initialized.
   // Which comes first, the chicken or the egg ?

   makeNewHen(&theFirstEgg);
}
```

Example 29 **One way of ensuring that global objects have been initialized**

```
// WARNING!!! THIS CODE IS NOT FOR BEGINNERS !!!

// PortSetup.hh

class PortSetup
{
   public:
      PortSetup();         // Constructor: initializes flag
      void foo();          // Only works correctly if flag is 42
   private:
      int flag;            // Always initialized to 42
};

extern PortSetup portSetup;  // Must be initialized before use

// Create one instance of portSetupInit in each translation unit
// The constructor for portSetupInit will be called once for each
// translation unit. It initializes portSetup by using the placement
// syntax for the ”new” operator.

static
class PortSetupInit
{
   public:
      PortSetupInit();           // Default constructor
   private:
      static int isPortSetup;
} portSetupInit;

// PortSetup.cc

#include "PortSetup.hh"
#include <new.h>

// ...

PortSetupInit::PortSetupInit()   // Default constructor
{
   if (!isPortSetup)
   {
      new (&portSetup) PortSetup;
      isPortSetup = 1;
   }
}
```

Example 30 Override of virtual functions does not work in the base class' constructors

```
class Base
{
   public:
      Base();        // Default constructor
      virtual void foo() { cout << "Base::foo" << endl; }
      // ...
};

Base::Base()
{
   foo();     // Base::foo() is ALWAYS called.
}

// Derived class overrides foo()
class Derived : public Base
{
   public:
      virtual void foo() { cout << "Derived::foo" << endl; } //foo is overridden
      // ...
};

main()
{
   Derived d;    // Base::foo() called when the Base-part of
                 // Derived is constructed.
}
```

7.6 Assignment Operators

Rule 27 A class which uses “new” to allocate instances managed by the class,[1] must define an ***assignment operator***.

Rule 28 An assignment operator which performs a destructive action must be protected from performing this action on the object upon which it is operating.

Rec. 34 An assignment operator ought to return a ***const*** reference to the assigning object.

An assignment is not inherited like other operators. If an assignment operator is not explicitly defined, then one is automatically defined instead. Such an assignment operator does not perform bit-wise copying of member data; instead, the assignment operator (if defined) for each specific type of member data is invoked. Bit-wise copying is only performed for member data having primitive types.

One consequence of this is that bit-wise copying is performed for member data having pointer types. If an object manages the allocation of the instance of an object pointed to by a pointer member, this will probably lead to problems: either by invoking the destructor for the managed object more than once or by attempting to use the deallocated object. See also Rule 25.

If an assignment operator is overloaded, the programmer must make certain that the base class’ and the members’ assignment operators are run.

A common error is assigning an object to itself (**`a = a`**). Normally, destructors for instances which are allocated on the heap are invoked before assignment takes place. If an object is assigned to itself, the values of the instance variables will be lost before they are assigned. This may well lead to strange run-time errors. If **`a = a`** is detected, the assigned object should not be changed.

If an assignment operator returns “void”, then it is not possible to write **`a = b = c`**. It may then be tempting to program the assignment operator so that it returns a reference to the assigning object. Unfortunately, this kind of design can be difficult to understand. The statement **`(a = b) = c`** can mean that **`a`** or **`b`** is assigned the value of **`c`** before or after **`a`** is assigned the value of **`b`**. This type of code can be avoided by having the assignment operator return a **`const`** reference to the assigned object or to the assigning object. Since the returned object cannot be placed on the left side of an assignment, it makes no difference which of them is returned (that is, the code in the above example is no longer correct).

Exception to Rule 27: Sometimes, it is desirable to allow objects in a class to share a data area. In such cases, it is not necessary to define an assignment operator. Instead, it is necessary to make sure that the shared data area is no deallocated as long as there are pointers to it.

Exception to Rule 28: No exceptions.

---

1. i.e. instances bound to member variables of pointer or reference type that are deallocated by the object.

Example 31 **Incorrect and correct return values from an assignment operator**

```
void
MySpecialClass::operator=( const MySpecialClass& msp );      / Well ...?

MySpecialClass&
MySpecialClass::operator=( const MySpecialClass& msp );      // No!

const MySpecialClass&
MySpecialClass::operator=( const MySpecialClass& msp );      // Recommended
```

Example 32 **Definition of a class with an overloaded assignment operator**

```
class DangerousBlob
{
   public:
      const DangerousBlob& operator=( const DangerousBlob& dbr );
      // ...
   private:
      char* cp;
};

// Definition of assignment operator

const DangerousBlob&
DangerousBlob::operator=( const DangerousBlob& dbr )
{
   if ( this != &dbr )    // Guard against assigning to the “this” pointer
   {
      delete cp;         // Disastrous if this == &dbr
   }
   // ...
}
```

7.7 Operator Overloading

Rec. 35 Use operator overloading sparingly and in a uniform manner.

Rec. 36 When two operators are opposites (such as **==** and **!=**), it is appropriate to define both.

Operator overloading has both advantages and disadvantages. One advantage is that code which uses a class with overloaded operators can be written more compactly (more readably). Another advantage is that the semantics can be both simple and natural. One disadvantage in overloading operators is that it is easy to misunderstand the meaning of an overloaded operator (if the programmer has not used natural semantics). The extreme case, where the plus-operator is re-defined to mean minus and the minus-operator is re-defined to mean plus, probably will not occur very often, but more subtle cases are conceivable.

Designing a class library is like designing a language! If you use operator overloading, use it in a uniform manner; do not use it if it can easily give rise to misunderstanding.

If the operator **!=** has been designed for a class, then a user may well be surprised if the operator **==** is not defined as well.

7.8 Member Function Return Types

Rule 29 A public member function must never return a non-const reference or pointer to member data.

Rule 30 A public member function must never return a non-const reference or pointer to data outside an object, unless the object shares the data with other objects.

By allowing a user direct access to the private member data of an object, this data may be changed in ways not intended by the class designer. This may lead to reduced confidence in the designer's code: a situation to be avoided.

A worse risk is having pointers which point to deallocated memory. Rule 29 and Rule 30 attempt to avoid this situation.

Note that we do ***not*** forbid the use of protected member functions which return a ***const*** reference or pointer to member data. If protected access functions are provided, the problems described in 7.1 are avoided.

Exception to Rule 29: No exceptions.

Exception to Rule 30: No exceptions.

Example 33 Never return a non-const reference to member data from a public function.

```
class Account
{
   public:
      Account( int myMoney ) : moneyAmount( myMoney ) {};
      const int& getSafeMoney() const { return moneyAmount; }
      int& getRiskyMoney() const { return moneyAmount; }       // No!
      // ...
   private:
      int moneyAmount;
};

Account myAcc(10);   // I'm a poor lonesome programmer a long way from home

myAcc.getSafeMoney() += 1000000;  // Compilation error: assignment to constant

myAcc.getRiskyMoney() += 1000000; // myAcc::moneyAmount = 1000010 !!
```

7.9 Inheritance

Rec. 37 Avoid inheritance for ***parts-of*** relations.

Rec. 38 Give derived classes access to class type member data by declaring protected access functions.

A common mistake is to use ***multiple inheritance*** for ***parts-of*** relations (when an object consists of several other objects, these are inherited instead of using instance variables. This can result in strange class hierarchies and less flexible code. In C++ there may be an arbitrary number of instances of a given type; if inheritance is used, direct inheritance from a class may only be used once[1].

A derived class often requires access to base class member data in order to create useful member functions. The advantage in using protected member functions is that the names of base class member data are not visible in the derived classes and thus may be changed. Such access functions should only return the values of member data (read-only access). This is best done by simply invoking **`const`** functions for the member data.

The guiding assumption is that those who use inheritance know enough about the base class to be able to use the private member data correctly, while not referring to this data by name. This reduces the coupling between base classes and derived classes.

1. As opposed to the language Eiffel, in which multiple, repeated inheritance is permitted.

8 Class Templates

Rec. 39 Do not attempt to create an instance of a class template using a type that does not define the member functions which the class template, according to its documentation, requires.

Rec. 40 Take care to avoid multiple definition of overloaded functions in conjunction with the instantiation of a class template.

It is not possible in C++ to specify requirements for type arguments for class templates and function templates. This may imply that the type chosen by the user, does not comply with the interface as required by the template. For example, a class template may require that a type argument have a comparison operator defined.

Another problem with type templates can arise for overloaded functions. If a function is overload, there may be a conflict if the element type appears explicitly in one of these. After instantiation, there may be two functions which, for example, have the type **`int`** as an argument. The compiler may complain about this, but there is a risk that the designer of the class does not notice it. In cases where there is a risk for multiple definition of member functions, this must be carefully documented.

Example 34 Problem when using parameterized types (Cfront 3.0 or other template compiler)

```
template <class ET>
class Conflict
{
   public:
      void foo( int a );
      void foo( ET a );     // What if ET is an int or another integral type?
                            // The compiler will discover this, but ...
};
```

9 Functions

Unless otherwise stated, the following rules also apply to member functions.

9.1 Function Arguments

Rule 31 Do not use unspecified function arguments (ellipsis notation).

Rec. 41 Avoid functions with many arguments.

Rec. 42 If a function stores a pointer to an object which is accessed via an argument, let the argument have the type pointer. Use reference arguments in other cases.

Rec. 43 Use constant references (**`const &`**) instead of call-by-value, unless using a pre-defined data type or a pointer.

The best known function which uses unspecified arguments is **`printf()`**. The use of such functions is not advised since the strong type checking provided by C++ is thereby avoided. Some of the possibilities provided by unspecified function arguments can be attained by overloading functions and by using default arguments.

Functions having long lists of arguments look complicated, are difficult to read, and can indicate poor design. In addition, they are difficult to use and to maintain.

By using references instead of pointers as function arguments, code can be made more readable, especially within the function. A disadvantage is that it is not easy to see which functions change the values of their arguments. Member functions which store pointers which have been provided as arguments should document this clearly by declaring the argument as a pointer instead of as a reference. This simplifies the code, since it is normal to store a pointer member as a reference to an object.

One difference between references and pointers is that there is no null-reference in the language, whereas there is a null-pointer. This means that an object must have been allocated before passing it to a function. The advantage with this is that it is not necessary to test the existence of the object within the function.

C++ invokes functions according to call-by-value. This means that the function arguments are copied to the stack via invocations of copy constructors, which, for large objects, reduces performance. In addition, destructors will be invoked when exiting the function. **`const &`** arguments mean that only a reference to the object in question is placed on the stack (call-by-reference) and that the object's state (its instance variables) cannot be modified. (At least some **`const`** member functions are necessary for such objects to be at all useful).

Exception to Rule 31: No exceptions.

Example 35 References instead of pointers

```
// Unnecessarily complicated use of pointers
void addOneComplicated( int* integerPointer )
{
   *integerPointer += 1;
}

addOneComplicated( &j );

// Write this way instead:
void addOneEasy( int& integerReference )
{
   integerReference += 1;
}

addOneEasy( i );
```

Example 36 Different mechanisms for passing arguments

```
// a. A copy of the argument is created on the stack.
//    The copy constructor is called on entry,
//    and the destructor is called at exit from the function.
//    This may lead to very inefficient code.

void foo1( String s );
String a;
foo1( a );     // call-by-value

// b. The actual argument is used by the function
//    and it can be modified by the function.

void foo2( String& s );
String b;
foo2( b );    // call-by-reference

// c. The actual argument is used by the function
//    but it cannot be modified by the function.

void foo3( const String& s );
String c;
foo3( c );    // call-by-constant-reference

// d. A pointer to the actual argument is used by the function.
//    May lead to messy syntax when the function uses the argument.

void foo4( const String* s );
String d;
foo4( &d );    // call-by-constant-pointer
```

9.2 Function Overloading

Rec. 44 When overloading functions, all variations should have the same semantics (be used for the same purpose).

Overloading of functions can be a powerful tool for creating a family of related functions that only differ as to the type of data provided as arguments. If not used properly (such as using functions with the same name for different purposes), they can, however, cause considerable confusion.

Example 37 Example of the proper usage of function overloading

```
class String
{
   public:                                  // Used like this:
      // ...                                // String x = "abc123";
      int contains( const char c );         // int i = x.contains( 'b' );
      int contains( const char* cs );       // int j = x.contains( "bc1" );
      int contains( const String& s );      // int k = x.contains( x );
      // ...
};
```

9.3 Formal Arguments

Rule 32 The names of ***formal arguments*** to functions are to be specified and are to be the ***same*** both in the function declaration and in the function definition.

The names of formal arguments may be specified in both the function declaration and the function definition in C++, even if these are ignored by the compiler in the declaration. Providing names for function arguments is a part of the function documentation. The name of an argument may clarify how the argument is used, reducing the need to include comments in, for example, a class definition. It is also easier to refer to an argument in the documentation of a class if it has a name.

Exception to Rule 32: No exceptions!

Example 38 Declaration of formal arguments

```
int setPoint( int, int );        // No !
int setPoint( int x, int y );    // Good

int
setPoint( int x, int y )
{
   // ...
}
```

9.4 Return Types and Values

Rule 33 Always specify the return type of a function explicitly.

Rule 34 A public function must never return a reference or a pointer to a local variable.

Functions, for which no return type is explicitly declared, implicitly receive `int` as the return type. This can be confusing for a beginner, since the compiler gives a warning for a missing return type. Because of this, functions which return no value should specify `void` as the return type.

If a function returns a reference or a pointer to a local variable, the memory to which it refers will already have been deallocated, when this reference or pointer is used. The compiler may or may not give a warning for this.

Exception to Rule 33: No exceptions!

Exception to Rule 34: No exceptions!

Example 39 Functions which return no value should be specified as having the return type void.

```
void
strangeFunction( const char* before, const char* after )
{
   // ...
}
```

9.5 Inline Functions

Rule 35 Do not use the preprocessor directive **`#define`** to obtain more efficient code; instead, use inline functions.

Rec. 45 Use **`inline`** functions when they are really needed.

See also 7.2.

`Inline` functions have the advantage of often being faster to execute than ordinary functions. The disadvantage in their use is that the implementation becomes more exposed, since the definition of an **`inline`** function must be placed in an include file for the class, while the definition of an ordinary function may be placed in its own separate file.

A result of this is that a change in the implementation of an **`inline`** function can require comprehensive re-compiling when the include file is changed. This is true for traditional file-based programming environments which use such mechanisms as **`make`** for compilation.

The compiler is not compelled to actually make a function inline. The decision criteria for this differ from one compiler to another. It is often possible to set a compiler flag so that the compiler gives a warning each time it does not make a function inline (contrary to the declaration). “Outlined inlines” can result in programs that are both unnecessarily large and slow.

It may be appropriate to separate inline definitions from class definitions and to place these in a separate file.

Exception to Rule 35: No exceptions

Example 40 Inline functions are better than macros

```
// Example of problems with #define “functions”
#define SQUARE(x) ((x)*(x))
int a = 2;
int b = SQUARE(a++);  // b = (2 * 3) = 6

// Inline functions are safer and easier to use than macros if you
// need an ordinary function that would have been unacceptable for
// efficiency reasons.
// They are also easier to convert to ordinary functions later on.
inline int square( int x )
{
   return ( x * x );
};

int c = 2;
int d = square( c++ );   // d = ( 2 * 2 ) = 4
```

9.6 Temporary Objects

Rec. 46 Minimize the number of temporary objects that are created as return values from functions or as arguments to functions.

Temporary objects are often created when objects are returned from functions or when objects are given as arguments to functions. In either case, a constructor for the object is first invoked; later, a destructor is invoked. Large temporary objects make for inefficient code. In some cases, errors are introduced when temporary objects are created. It is important to keep this in mind when writing code. It is especially inappropriate to have pointers to temporary objects, since the lifetime of a temporary object is undefined. (See 18.7).

Example 41 Temporary objects and one way of eliminating them

```
class BigObject  { double big[123456]; };

// Example of a very inefficient function with respect to temporary objects:
BigObject
slowTransform( BigObject myBO )
{
   // When entering slowTransform(), myBO is a copy of the function argument
   // provided by the user. -> A copy constructor for BigObject is executed.
   // ... Transform myBO in some way
   return myBO;   // Transformed myBO returned to the user
}
// When exiting slowTransform(), a copy of myBO is returned to the
// user -> copy-constructor for BigObject is executed, again.

// Much more efficient solution:
BigObject&
fastTransform( BigObject& myBO )
{
   // When entering fastTransform(), myBO is the same object as the function
   // argument provided by the user. -> No copy-constructor is executed.

   // Transform myBO in some way
   return myBO;   // Transformed myBO is returned to the user.
}
// When exiting fastTransform(), the very same myBO is returned
// to the user. -> No copy constructor executed.

void main()
{
   BigObject BO;
   BO = slowTransform( BO );
   BO = fastTransform( BO );   // Same syntax as slowTransform() !!
}
```

9.7 General

Rec. 47 Avoid long and complex functions.

Long functions have disadvantages:

1. If a function is too long, it can be difficult to comprehend. Generally, it can be said that a function should not be longer than two pages, since that is about how much that can be comprehended at one time.
2. If an error situation is discovered at the end of an extremely long function, it may be difficult for the function to clean up after itself and to "undo" as much as possible before reporting the error to the calling function. By always using short functions, such an error can be more exactly localized.

Complex functions are difficult to test. If a function consists of 15 nested if statements, then there are 2**15 (or 32768) different branches to test in a single function.

10 Constants

Rule 36 Constants are to be defined using **`const`** or **`enum`**; never using #**`define`**.

Rule 37 Avoid the use of numeric values in code; use symbolic values instead.

The preprocessor performs a textual substitution for macros in the source code which is then compiled. This has a number of negative consequences. For example, if a constant has been defined using #**`define`**, the name of the constant is not recognized in many debuggers. If the constant is represented by an expression, this expression may be evaluated differently for different instantiations, depending on the scope of the name. In addition, macros are, at times, incorrectly written.

Numerical values in code (“Magic Numbers”) should be viewed with suspicion. They can be the cause of difficult problems if and when it becomes necessary to change a value. A large amount of code can be dependent on such a value never changing, the value can be used at a number of places in the code (it may be difficult to locate all of them), and values as such are rather anonymous (it may be that every ‘2’ in the code should not be changed to a ‘3’).

From the point of view of portability, absolute values may be the cause of more subtle problems. The type of a numeric value is dependent on the implementation. Normally, the type of a numeric value is defined as the smallest type which can contain the value.

Exception to Rule 36: No exceptions.

Exception to Rule 37: Certain numerical values have a well established and clear meaning in a program. For example, ’`1`’ and ’`0`’ are often used to represent ‘true’ and ‘false’ respectively. These may be used directly in code without being considered to be “Magic”.

Example 42 Different ways of declaring constants.

```
// Constants using macros
#define BUFSIZE 7              // No type checking

// Constants using const
const int bufSize = 7;         // Type checking takes place

// Constants using enums
enum SIZE { BufSize = 7 };     // Type checking takes place
```

Example 43 Declaration of `const` defined in another file

```
extern const char constantCharacter;
extern const String fileName;
```

11 Variables

Rule 38 Variables are to be declared with the smallest possible ***scope***.

Rule 39 Each variable is to be declared in a ***separate declaration statement.***

Rule 40 Every variable that is declared is to be ***given a value*** before it is used.

Rule 41 If possible, always use initialization instead of assignment.

A variable ought to be declared with the smallest possible scope to improve the readability of the code and so that variables are not unnecessarily allocated. When a variable that is declared at the beginning of a function is used somewhere in the code, it is not easy to directly see the type of the variable. In addition, there is a risk that such a variable is inadvertently hidden if a local variable, having the same name, is declared in an internal block.

Many local variables are only used in special cases which seldom occur. If a variable is declared at the outer level, memory will be allocated even if it is not used. In addition, when variables are initialized upon declaration, more efficient code is obtained than if values are assigned when the variable is used.

A variable must always be initialized before use. Normally, the compiler gives a warning if a variable is undefined. It is then sufficient to take care of such cases. Instances of a class are usually initialized even if no arguments are provided in the declaration (the empty constructor is invoked). To declare a variable that has been initialized in another file, the keyword **`extern`** is always used.

By always initializing variables, instead of assigning values to them before they are first used, the code is made more efficient since no temporary objects are created for the initialization. For objects having large amounts of data, this can result in significantly faster code.

Exception to Rule 38: No exceptions.

Exception to Rule 39: No exceptions.

Exception to Rule 40: No exceptions.

Exception to Rule 41: In certain special cases, a variable is assigned the value of a complicated expression; it may then be unnecessary to give the variable an initial value. See Example 44.

Example 44 Initialization instead of Assignment

```
// Do not do this!
// int i;
// ... 1022 lines of code
// i = 10;

int j = 10;                    // Better

class Special                  // Array of this class is used to initialize
{                              // MyClass::complicated
   public:
      Special();               // Default constructor
      int isValid() const;
      int value() const;
};

const int Magic = 1066;
Special specialInit[Magic];

class MyClass
{
   public:
      MyClass( const char* init );   // Constructor
      // ...
   private:
      String privateString;
      int complicated;
};

// Do not do this! Inefficient code.
// Empty constructor + assignment operator called for privateString
//
// MyClass::MyClass( const char* init )
// {
//    privateString = init;
//    ...
// }

MyClass::MyClass( const char* init ) : privateString( init )    // Better
{
   // Special case - complicated expression

   for( int i = 0; i < Magic; i++ )        // No! You should enclose "for"
      if ( specialInit[i].isValid() )      // loops in braces! See Rec. 25!
      {
         complicated = specialInit[i].value();
         break;
      }
}
```

12 Pointers and References

Rule 42 Do not compare a pointer to NULL or assign NULL to a pointer; use 0[1] instead.

Rec. 48 Pointers to pointers should whenever possible be avoided.

Rec. 49 Use a **`typedef`** to simplify program syntax when declaring function pointers.

According to the ANSI-C standard, NULL is defined either as **`(void*)0`** or as **`0`**. If this definition remains in ANSI-C++, problems may arise. If NULL is defined to have the type **`void*`**, it cannot be assigned an arbitrary pointer without an explicit type conversion. For this reason, we recommend comparisons with 0 at least until the ANSI-C++ committee has made a decision.

Pointers to pointers normally ought not be used. Instead, a class should be declared, which has a member variable of the pointer type. This improves the readability of the code and encourages data abstraction. By improving the readability of code, the probability of failure is reduced. One exception to this rule is represented by functions which provide interfaces to other languages (such as C). These are likely to only allow pre-defined data types to be used as arguments in the interface, in which case pointers to pointers are needed. Another example is the second argument to the **`main`** function, which must have the type `char*[]`[2].

A function which changes the value of a pointer that is provided as an argument, should declare the argument as having the type reference to pointer (e.g. `char*&`). See Rec. 42!

`typedef` is a good way of making code more easily maintainable and portable. See chapter 18.1, Port.Rec.1. Another reason to use **`typedef`** is that the readability of the code is improved. If pointers to functions are used, the resulting code can be almost unreadable. By making a type declaration for the function type, this is avoided.

Function pointers can be used as ordinary functions; they do not need to be dereferenced[3].

Exception to Rule 42: No exceptions.

Example 45 Different comparisons of pointers

```
char* sp = new char[100];
if ( !sp )         cout << "New failed!" << endl;   // No!
if ( sp == 0 )     cout << "New failed!" << endl;   // Best
if ( sp == NULL ) cout << "New failed!" << endl;   // ERROR sometimes !!!
```

1. An intensive debate about this has been raging in the “news” group `comp.lang.c++`. Future changes in this recommendation may occur.
2. This is equivalent to `char**`.
3. See Example 46.

Example 46 Pointers to pointers are often unnecessary[1]

```
#include <iostream.h>

void print_mij(int** m, int dim1, int dim2)
{
   for (int i = 0; i < dim1; i++)
   {
      for (int j = 0; j < dim2; j++ )
         cout << " " << ((int*)m)[i*dim2+j];
      cout << endl;
   }
}

// Could be written as:

class Int_Matrix
{
   public:
      Int_Matrix(int dim1, int dim2);
      int value(int,int) const;
      int dim1() const;
      int dim2() const;
   // ..
};

void print_Mij(Int_Matrix m)
{
   for (int i = 0; i < m.dim1(); i++)
   {
      for (int j = 0; j < m.dim2(); j++ )
         cout << " " << m.value(i,j);
      cout << endl;
   }
}
```

1. This example is, in part, taken from [3]: The C++ Programming Language, Second Edition – Bjarne Stroustrup.

Example 47 Complicated declarations

```
// func1 is a function: int -> (function : const char* -> int)
// i.e. a function having one argument of type int and returning
// a pointer to a function having one argument of type const char*
// and returning an int.

int (*func1(int))(const char*);

// func1 of the same type as func2

typedef int FTYPE(const char*);
FTYPE* func2(int);

int (*(*func1p)(int))(const char*) = func2;

// Realistic example from signal.h

void (*signal(int,void (*)(int)))(int);
```

Example 48 Syntax simplification of function pointers using a typedef

```
#include <math.h>

// Ordinary messy way of declaring pointers to functions:
// double ( *mathFunc ) ( double ) = sqrt;

// With a typedef, life is filled with happiness (chinese proverb):
typedef double MathFuncType( double );
MathFuncType* mathFunc = sqrt;

void
main()
{
   // You can invoke the funktion in an easy or complicated way
   double returnValue1 = mathFunc( 23.0 );     // Easy way
   double returnValue2 = ( *mathFunc )( 23.0 ); // No! Correct, but complicated
}
```

13 Type Conversions

Rule 43 Never use ***explicit type conversions (casts)***.

Rule 44 Do not write code which depends on functions that use ***implicit type conversions***.

Rule 45 Never convert pointers to objects of a derived class to pointers to objects of a virtual base class.

Rule 46 Never convert a **`const`** to a non-**`const`**.

A type conversion may either be explicit or implicit, depending on whether it is ordered by the programmer or by the compiler. Explicit type conversions (casts) are used when a programmer want to get around the compiler's typing system; for success in this endeavour, the programmer must use them correctly. Problems which the compiler avoids may arise, such as if the processor demands that data of a given type be located at certain addresses or if data is truncated because a data type does not have the same size as the original type on a given platform. Explicit type conversions between objects of different types lead, at best, to code that is difficult to read.

Explicit type conversions (casts) can be motivated if a base class pointer to a derived class pointer is needed. This happens when, for example, a heterogeneous container class is used to implement a container class to store pointers to derived class objects. This new class can be made "type-safe" if the programmer excludes other objects than derived class pointers from being stored. In order for this implementation to work, it is necessary that the base class pointers are converted to derived class pointers when they are removed from the heterogeneous container class.

The above reason for using explicit casts will hopefully disappear when templates are introduced into C++.

It is sometimes said that explicit casts are to object-oriented programming, what the **`goto`** statement was to structured programming.

There are two kinds of implicit type conversions: either there is a conversion function from one type to another, written by the programmer, or the compiler does it according to the language standard. Both cases can lead to problems.

C++ is lenient concerning the variables that may be used as arguments to functions. If there is no function which exactly matches the types of the arguments, the compiler attempts to convert types to find a match. The disadvantage in this is that if more than one matching function is found, a compilation error will be the result. Even worse is that existing code which the compiler has allowed in other contexts, may contain errors when a new implicit type conversion is added to the code. Suddenly, there may be more than one matching function[1].

Another unpredictable effect of implicit type conversions is that temporary objects are created during

1. See Example 53!

the conversion[1]. This object is then the argument to the function; not the original object. The language definition forbids the assignment of temporary objects to non-constant references, but most compilers still permit this. In most cases, this can mean that the program does not work properly. Be careful with constructors that use only one argument, since this introduces a new type conversion which the compiler can unexpectedly use when it seems reasonable in a given situation.

Virtual base classes give rise to other type conversion problems. It is possible to convert a pointer, to an instance of a class which has a virtual base class, to a pointer to an object of that virtual base class. The opposite conversion is not allowed, i.e. the type conversion is not reversible. For this reason, we do not recommend the conversion of a derived class pointer to a virtual base class pointer.

In order to return a non-const temporary object, it sometimes happens that an explicit type conversion is used to convert const member data to non-const. This is bad practice that should be avoided, primarily because it should be possible for a compiler to allocate constants in ROM (Read Only Memory)[2].

Exception to Rule 43: An explicit type conversion (cast) is preferable to a doubtful implicit type conversion.
Explicit type conversions may be used to convert a pointer to a base class to a pointer of a derived class within a type-safe container class that is implemented using a heterogeneous container class.
Explicit type conversion must be used to convert an anonymous bit-stream to an object. This situation occurs when unpacking a message in a message buffer. Generally, explicit type conversions are needed for reading an external representation of an object.

Exception to Rule 44: At times it is desirable to have constructors that use only one argument. By performing an explicit type conversion, the correctness of the code does not depend on the addition. See the Exception to Rule 22!

Exception to Rule 45: If a virtual base class is to contain a pure virtual function which converts a virtual base class pointer to a derived class pointer, this can be made to work by defining the function in the derived class. Note that this implies that all derived classes must be known in the virtual base class. See Example 52!

Exception to Rule 46: No exceptions. Use pointers to data allocated outside the class, when necessary. See Example 54 and Example 55.

1. See Example 51!
2. See Example 54 and Example 55.

Example 49 **Constructors with a single argument that may imply dangerous type conversions**

```
class String
{
   public:
      String( int length );     // Allocation constructor
      // ...
};

// Function that receives an object of type String as an argument
void foo( const String& aString );

// Here we call this function with an int as argument
int x = 100;
foo( x );       // Implicit conversion: foo( String( x ) );
```

Example 50 **A use of implicit type conversion**

```
// String.hh

class String
{
   public:
      String( char* cp );              // Constructor
      operator const char* () const;  // Conversion operator to const char*
      // ...
};

void foo( const String& aString );
void bar( const char* someChars );

// main.cc

main()
{
   foo( "hello" );        // Implicit type conversion char* -> String
   String peter = "pan";
   bar( peter );          // Implicit type conversion String -> const char*
}
```

Example 51 When implicit type conversion gives unpleasant results

```
// This function looks bulletproof, but it isn't.
// Newer versions of compilers should flag this as an error.
void
mySwap( int& x, int& y )
{
   int temp = x;
   x = y;
   y = temp;
}

int i = 10;
unsigned int ui = 20;
mySwap( i, ui );      // What really happens here is:
                      // int T = int( ui );    // Implicit conversion
                      // mySwap( i, T );       // ui is of course not changed!
                      // Fortunately, the compiler warns for this !
```

Example 52 Conversion of derived class pointer to a virtual base class pointer is irreversible

```
class VirtualBase
{
   public:
      virtual class Derived* asDerived() = 0;
};

class Derived : virtual public VirtualBase
{
   public:
      virtual Derived* asDerived();
};

Derived*
Derived::asDerived()
{
   return this;
}

void
main()
{
   Derived d;
   Derived* dp = 0;
   VirtualBase* vp = (VirtualBase*)&d;

   dp = (Derived*)vp; // ERROR! Cast from virtual base class pointer
   dp = vp->asDerived(); // OK! Cast in function asDerived
}
```

Example 53 Addition which leads to a compile-time error

```
// String.hh

class String
{
   public:
      String( char* cp );               // Constructor
      operator const char* () const;    // Conversion operator to const char*
      // ...
};

void foo( const String& aString );
void bar( const char* someChars );

// Word.hh

class Word
{
   public:
      Word( char* cp );   // Constructor
      // ...
};

// Function foo overloaded

void foo( const Word& aWord );

// ERROR: foo( "hello" ) MATCHES BOTH:
// void foo( const String& );
// AND void foo( const Word& );

//main.cc

main()
{
   foo( "hello" );         // Error ambiguous type conversion !
   String peter = "pan";
   bar( peter );           // Implicit type conversion String -> const char*
}
```

Example 54 For more efficient execution, remove const-ness when storing intermediate results

```
// This is code is NOT recommended

#include <math.h>

class Vector
{
   public:
      Vector(int, const int []);  // Constructor
      double length() const;      // length = sqrt(array[1]*array[1] + ... )
      void set(int x, int value);
      // ...
   private:
      int size;
      int* array;
      double lengthCache;    // to cache calculated length
      int hasChanged;        // is it necessary to re-calculate length ?
};

double
Vector::length() const
{
   if (hasChanged) // Do we need to re-calculate length
   {
      ((Vector*)this)->hasChanged=0; // No! Cast away const
      double quadLength = 0;
      for ( int i = 0; i < size; i++ )
      {
         quadLength += pow(array[i],2);
      }
      ((Vector*)this)->lengthCache = sqrt(quadLength); // No! Cast away const
   }
   return lengthCache;
}

void
Vector::set( int nr, int value )
{
   if ( nr >= size ) error( "Out Of Bounds");
   array[nr]=value;
   hasChanged = 1;
}
```

Example 55 Alternative to removing const-ness for more efficient execution

```
// This is code is safer than Example 54 but could be inefficient

#include <math.h>

class Vector
{
   public:
      Vector(int, const int []);  // Constructor
      double length() const;      // length = sqrt(array[1]*array[1] + ... )
      void set(int x, int value);
      // ...
   private:
      int size;
      int* array;
      double* lengthCache;   // to cache length in
      int* hasChanged;       // is it necessary to re-calculate length ?
};

Vector::Vector(int sizeA, const int arrayA[])
: size(sizeA), array( new int[sizeA] ),
  hasChanged(new int(1)), lengthCache(new double)
{
   for ( int i = 0; i < size; i++ )
   {
      array[i] = arrayA[i];
   }
}

Vector::~Vector()    // Destructor
{
   delete array;
   delete hasChanged;
   delete lengthCache;
}

// Continue on next page !
```

```
double
Vector::length() const
{
   if (hasChanged)     // Do we need to re-calculate length ?
   {
      *hasChanged=0;
      double quadLength = 0;
      for ( int i = 0; i < size; i++ )
      {
         quadLength += pow(array[i],2);
      }
      *lengthCache = sqrt(quadLength);
   }
   return lengthCache;
}

void
Vector::set( int nr, int value )
{
   if ( nr >= size ) error( "Out Of Bounds");
   array[nr]=value;
   *hasChanged = 1;
}
```

14 Flow Control Structures

Rule 47 The code following a **`case`** label must ***always*** be terminated by a **`break`** statement.

Rule 48 A **`switch`** statement must ***always*** contain a **`default`** branch which handles unexpected cases.

Rule 49 Never use **`goto`**.

Rec. 50 The choice of loop construct (**`for`**, **`while`** or **`do-while`**) should depend on the specific use of the loop.

Rec. 51 Always use **`unsigned`** for variables which cannot reasonably have negative values.

Rec. 52 Always use inclusive lower limits and exclusive upper limits.

Rec. 53 Avoid the use of **`continue`**.

Rec. 54 Use **`break`** to exit a loop if this avoids the use of flags.

Rec. 55 Do not write logical expressions of the type **`if(test)`** or **`if(!test)`** when **`test`** is a pointer.

Each loop construct has a specific usage. A **`for`** loop is used only when the loop variable is increased by a constant amount for each iteration and when the termination of the loop is determined by a constant expression. In other cases, **`while`** or **`do-while`** should be used. When the terminating condition can be evaluated at the beginning of the loop, **`while`** should be used; **`do-while`** is used when the terminating condition is best evaluated at the end of the loop.

`Goto` breaks the control flow and can lead to code that is difficult to comprehend. In addition, there are limitations for when **`goto`** can be used. For example, it is not permitted to jump past a statement that initializes a local object having a destructor.

Variables representing size or length are typical candidates for **`unsigned`** declarations. By following this recommendation some unpleasant errors can be avoided.

It is best to use inclusive lower and exclusive upper limits. Instead of saying that **`x`** is in the interval **`x>=23`** and **`x<=42`**, use the limits **`x>=23`** and **`x<43`**. The following important claims then apply:

- The size of the interval between the limits is the difference between the limits.
- The limits are equal if the interval is empty.
- The upper limit is never less than the lower limit.

By being consistent in this regard, many difficult errors will be avoided.

If the code which follows a **`case`** label is not terminated by **`break,`** the execution continues after the next **`case`** label. This means that poorly tested code can be erroneous and still seem to work.

`continue` can be used to exit from loops. However, code may be more comprehensible by using an **`else`** clause instead.

C++ has a very loose and, simultaneously, very free way of determining if an expression is true or false. If an expression is evaluated as 0, it is false; otherwise, it is considered to be true.

We do not recommend logical tests such as "**`if(pointer)`**" if "**`pointer`**" is a variable of pointer-type. The only reason is readablity; many programmers find it difficult to read such code.

Consider the scope within which an iteration variable is visible. A variable that is declared within a 'for' statement is currently only visible in the nearest enclosing block. The standardization committee for C++ is however discussing a language modification regarding this point. No decision has yet been made. Still, this problem is avoided if the control structure is encapsulated in a compound statement.

Exception to Rule 47: When several **`case`** labels are followed by the same block of code, only one **`break`** statement is needed. Also, other statements than **`break`** may be used to exit a **`switch`** statement, such as **`return`**.

Exception to Rule 48: No exceptions.

Exception to Rule 49: For extremely time-critical applications or for fault handling, **`goto`** may be permitted. Every such usage must be carefully motivated.

Example 56 Problem using unsigned loop variables

```
for( unsigned int i = 3; i >= 0; --i )
{
   // This loop will never terminate, since i cycles through:
   // 3, 2, 1, 0, 4294967295, 4294967294, etc ... on a SparcStation
   // Note that this example does not follow the rules: i >= 0
   // in the for statement. See next example !
}
```

Example 57 Visibility of variable declared in 'for' loop

```
for ( int index = 0; index < 10; index++ )
{
   cout << index;
}

int index = 3;     // ERROR, THIS IS AN ILLEGAL RE-DECLARATION OF index
                   // BECAUSE index IS DECLARED IN BLOCK-SCOPE.
```

Example 58 **Dangerous switch/case statement**

```
switch ( tag )
{
   case A:
   {
      // Do something
      // Next statement is a call to foo() inside next case
   }

   case B:
   {
      foo();
      // Do something else
      break;      // Now we leave the switch-statement
   }

   default:
   {
      // If no match in above cases, this is executed
      exit( 1 );
   }
}
```

Example 59 **Good and bad ways of setting limits for loop variables**

```
int a[10];
int ten = 10;
int nine = 9;

// Good way to do it:
for( int i = 0; i < ten; i++ )        // Loop runs 10-0=10 times
{
   a[i] = 0;
}

// Bad way to do it:
for( int j = 0; j <= nine; j++ )      // Loop runs 10 times, but 9-0=9 !!!
{
   a[j] = 0;
}
```

Example 60 **Using break to exit a loop, no flags are needed.**

```
do                                      // This way:
{
   if ( Something )
   {
      // Do something
      break;
   }
} while( someCondition );


int endFlag = 0;                        // Is better than this:
do
{
   if ( /* Something */ )
   {
      // Do something
      endFlag = 1;
   }
} while( someCondition && !endFlag );
```

Example 61 **By using an extra 'else' clause, `continue` is avoided and the code can be comprehended.**

```
while( /* Something */ )                // This way is more clear
{
   if( /* Something */ )
   {
      // Do something
   }

   else
   {
      // Do something else
   }
}


while( /* Something */ )                // Than using continue
{
   if( /* Something */ )
   {
      // Do something
      continue;                        // No !
   }
   // Do something else
}
```

15 Expressions

Rec. 56 Use parentheses to clarify the order of evaluation for operators in expressions.

There are a number of common pitfalls having to do with the order of evaluation for operators in an expression. Binary operators in C++ have associativity (either leftwards or rightwards) and precedence. If an operator has leftwards associativity and occurs on both sides of a variable in an expression, then the variable belongs to the same part of the expression as the operator on its left side.

In doubtful cases, parentheses ***always*** are to be used to clarify the order of evaluation.

Another common mistake is to confuse the assignment operator and the equality operator. Since the assignment operator returns a value, it is entirely permitted to have an assignment statement instead of a comparison expression. This, however, most often leads straight to an error.

C++ allows the overloading of operators, something which can easily become confusing. For example, the operators << (shift left) and >> (shift right) are often used for input and output. Since these were originally bit operations, it is necessary that they have higher priority than relational operators. This means that parentheses must be used when outputting the values of logical expressions.

Example 62 Problem with the order of evaluation

```
// Interpreted as ( a<b ) < c, not ( a<b ) && ( b<c )
if ( a < b < c )
{
   // ...
}

// Interpreted as a & ( b < 8 ), not ( a & b ) < 8
if ( a & b < 8 )
{
   // ...
}
```

Example 63 When parentheses are recommended

```
int i = a >= b && c < d && e + f <= g + h;                      // No!
int j = ( a >= b ) && ( c < d ) && (( e + f ) <= ( g + h ));   // Better
```

16 Memory Allocation

Rule 50 Do not use **`malloc`**, **`realloc`** or **`free`**.

Rule 51 Always provide empty brackets ("**`[]`**") for **`delete`** when deallocating arrays.

Rec. 57 Avoid global data if at all possible.

Rec. 58 Do not allocate memory and expect that someone else will deallocate it later.

Rec. 59 Always assign a new value to a pointer that points to deallocated memory.

In C++ data can be allocated statically, dynamically on the stack, or dynamically on the heap. There are three categories of static data: global data, global class data, and static data local to a function.

In C **`malloc`**, **`realloc`** and **`free`** are used to allocate memory dynamically on the heap. This may lead to conflicts with the use of the **`new`** and **`delete`** operators in C++.

It is dangerous to:

1 invoke **`delete`** for a pointer obtained via **`malloc/realloc`**,

2 invoke **`malloc/realloc`** for objects having constructors,

3 invoke **`free`** for anything allocated using **`new`**.

Thus, avoid whenever possible the use of **`malloc`**, **`realloc`** and **`free`**.

If an array **`a`** having a type **`T`** is allocated, it is important to invoke **`delete`** in the correct way. Only writing **`delete a;`** will result in the destructor being invoked only for the first object of type **`T`**. By writing **`delete [m] a;`** where **`m`** is an integer which is greater than the number of objects allocated earlier, the destructor for **`T`** will be invoked for memory that does not represent objects of type **`T`**. The easiest way to do this correctly is to write **`delete [] a;`** since the destructor will then be invoked only for those objects which have been allocated earlier.

Static data can cause several problems. In an environment where parallel threads execute simultaneously, they can make the behaviour of code unpredictable, since functions having static data are not reentrant.

One difference between ANSI-C and C++ is in how constants are declared. If a variable is declared as a constant in ANSI-C, it has the storage class **`extern`** (global). In C++, however, it normally has the storage class **`static`** (local). The latter means that a new instance of the constant object is created each time a file includes the file which contains the declaration of the object, unless the variable is explicitly declared extern in the include file.

An extern declaration in C++ does not mean that the variable is initialized; there must be a definition for this in a definition file. Static constants that are defined within a class are always external and must always be defined separately.

It may, at times, be tempting to allocate memory for an object using **new**, expecting someone else to deallocate the memory. For instance, a function can allocate memory for an object which is then returned to the user as the return value for the function. There is no guarantee that the user will remember to deallocate the memory and the interface with the function then becomes considerably more complex.

Pointers that point to deallocated memory should either be set to 0 or be given a new value to prevent access to the released memory. This can be a very difficult problem to solve when there are several pointers which point to the same memory, since C++ has no garbage collection.

Exception to Rule 50: No exceptions.

Exception to Rule 51: No exceptions.

Example 64 Right and wrong ways to invoke delete for arrays with destructors

```
int n = 7;
T* myT = new T[n]; // T is a type with defined constructors and destructors
// ...
delete myT;         // No! Destructor only called for first object in array a
delete [10] myT;    // No! Destructor called on memory out of bounds in array a
delete [] myT;      // OK, and always safe!
```

Example 65 Dangerous memory management

```
String myFunc( const char* myArgument )
{
   String* temp = new String( myArgument );
   return *temp;
   // temp is never deallocated and the user of myFunc cannot deallocate
   // because a temporary copy of that instance is returned.
}
```

17 Fault Handling

Rec. 60 Make sure that fault handling is done so that the transfer to exception handling (when this is available in C++) may be easily made.

Rec. 61 Check the fault codes which may be received from library functions even if these functions seem foolproof.

In November 1990, the ANSI C++ committee accepted a proposal for exception handling which is described in chapter 15 of ref. 1. When designing fault handling in code, it is appropriate to consider being able to make a smooth transfer to exception handling. For example, instead of using ordinary fault codes, which may necessitate a lot of re-programming when exception handling is available, a call can be made to a function **`void fault(const char*)`** which sends a fault message (somewhere) and then terminates execution in some way.

System functions (those which are specific to UNIX) ought to be used with care if the code is to be portable. If such functions are used, the possible fault codes that may be received should be carefully checked.

Two important characteristics of a robust system are that all faults are reported and, if the fault is so serious that continued execution is not possible, the process is terminated. In this way, the propagation of faults through the system is avoided. It is better to have a process crash, than to spread erroneous information to other processes. In achieving this goal, it is important to always test fault codes from library functions. The opening or closing of files may fail, allocation of data may fail, etc. One test too many is better than one test too few. Our own functions should preferably not return fault codes, but should instead take advantage of exception handling.

Example 66 Future exception handling in C++

```
// The top function where we catch exceptions thrown in called functions
int f()
{
   // We suspect that something can go wrong when function g() is called.
   // Therefore, we enclose the call in a try block.
   try
   {
      return g();       // This is the try block
   }

   // If any exceptions, having a given type, were thrown when g()
   // was executing, they are caught in these two catch blocks.
   catch ( int x )      // catches int
   {
      cerr << "Number " << x << " happened !" << endl;
      return x;
   }

   catch ( char* x )    // catches char*
   {
      // Respond in some other way
   }

   // Anything else that is thrown, is thrown up to the function that calls f()
}

// This function has no try or catch block. When the exception is thrown
// in function h(), it is thrown up to the function f().
int g()
{
   return h();
}

extern int somethingIsVeryWrongAndICannotHandleThisAnyMore();

int h()
{
   // Here we find out that something went wrong, and throw an exception
   if (somethingIsVeryWrongAndICannotHandleThisAnyMore())
   {
      // In this case, we throw an int as exception, but almost any object
      // can be thrown. See Errata for “The Annotated C++ Reference Manual”
      // section 15.7.
      throw 2;
   }

   // Keep on trucking if all is OK
}
```

18 Portable Code

18.1 Data Abstraction

Port. Rec. 1 Avoid the direct use of pre-defined data types in declarations.

An excellent way of transforming your world to a “vale of tears” is to directly use the pre-defined data types in declarations. If it is later necessary, due to portability problems, to change the return type of a function, it may be necessary to make change at a large number of places in the code. One way to avoid this is to declare a new type name using classes or **`typedef`**s to represent the types of variables used. In this way, changes can be more easily made. This may be used to give data a physical unit, such as kilogram or meter. Such code is more easily reviewed. (For example, when the code is functioning poorly, it may be noticed that a variable representing meters has been assigned to a variable representing kilograms). It should be noted that a **`typedef`** does not create a new type, only an alternative name for a type. This means that if you have declared **`typedef int Error`**, a variable of the type **`Error`** may be used anywhere that an **`int`** may be used.

See also chapter 12, Rec. 49!

Example 67 Type declarations using typedef

```
// Instead of:
long int time;
short int mouseX;
char* menuName;

// Use (for example):
typedef long int TimeStamp;
typedef short int Coordinate;
class String { /* ... */ };

// and:
TimeStamp time;
Coordinate mouseX;
String menuName;
```

18.2 Sizes of Types

Port. Rec. 2 Do not assume that an **`int`** and a **`long`** have the same size.

Port. Rec. 3 Do not assume that an **`int`** is 32 bits long (it may be only 16 bits long).

Port. Rec. 4 Do not assume that a **`char`** is **`signed`** or **`unsigned`**.

Port. Rec. 5 Always set **`char`** to **`unsigned`** if 8-bit ASCII is used.

In the definition of the C++ language, it has not yet been decided if a **`char`** is **`signed`** or **`unsigned`**. This decision has instead been left to each compiler manufacturer. If this is forgotten and this characteristic is exploited in one way or another, some difficult bugs may appear in the program when another compiler is used.

If 8-bit ASCII is used (as is quite likely in the future) and comparisons are made of two characters, it is important that **`unsigned char`** is used.

18.3 Type Conversions

Port. Rec. 6 Be careful not to make type conversions from a “shorter” type to a “longer” one.

Port. Rec. 7 Do not assume that pointers and integers have the same size.

Port. Rec. 8 Use explicit type conversions for arithmetic using signed and unsigned values.

A processor architecture often forbids data of a given size to be allocated at an arbitrary address. For example, a word must begin on an “even” address for MC680x0. If there is a pointer to a **`char`** which is located at an “odd” address, a type conversion from this **`char`** pointer to an **`int`** pointer will cause the program to crash when the **`int`** pointer is used, since this violates the processor’s rules for alignment of data.

18.4 Data Representation

Port. Rec. 9 Do not assume that you know how an instance of a data type is represented in memory.

Port. Rec. 10 Do not assume that **`long`**s, **`float`**s, **`double`**s or **`long double`**s may begin at arbitrary addresses.

The representation of data types in memory is highly machine-dependent. By allocating data members to certain addresses, a processor may execute code more efficiently. Because of this, the data structure that represents a class will sometime include holes and be stored differently in different process architectures. Code which depends on a specific representation is, of course, not portable.

See18.3 for explanation of Port. Rec. 10.

18.5 Underflow/Overflow

Port. Rec. 11 Do not depend on underflow or overflow functioning in any special way.

18.6 Order of Execution

Port. Rec. 12 Do not assume that the operands in an expression are evaluated in a definite order.

Port. Rec. 13 Do not assume that you know how the invocation mechanism for a function is implemented.

Port. Rec. 14 Do not assume that an object is initialized in any special order in constructors.

Port. Rec. 15 Do not assume that static objects are initialized in any special order.

If a value is modified twice in the same expression, the result of the expression is undefined except when the order of evaluation is guaranteed for the operators that are used.

The order of initialization for static objects may present problems. A static object may not be used in a constructor, if it is not initialized until after the constructor is run. At present, the order of initialization for static objects, which are defined in different compilation units, is not defined. This can lead to errors that are difficult to locate (see Example 69). There are special techniques for avoiding this. See Example 29!

Example 68 Do not depend on the order of initialization in constructors.

```
#include <iostream.h>
class X
{
   public:
      X(int y);
   private:
      int i;
      int j;
};

inline X::X(int y) : j(y), i(j)    // No! j may not be initialized before i !!
{
   cout << "i:" << i << " & " << "j:" << j << endl;
}

main()
{
   X x(7);        // Rather unexpected output: i:0 & j:7
}
```

Example 69 Initialization of static objects

```
// Foo.hh

#include <iostream.h>
#include <string.h>

static unsigned int const Size = 1024;

class Foo
{
   public:
      Foo( char* cp );                // Constructor
      // ...
   private:
      char buffer[Size];
      static unsigned counter;        // Number of constructed Foo:s
};

extern Foo foo_1;
extern Foo foo_2;

// Foo1.cc
#include "Foo.hh"

unsigned Foo::counter = 0;
Foo foo_1 = "one";

//Foo2.cc
#include "Foo.hh"

Foo foo_2 = "two";

Foo::Foo( char* cp )      // Irrational constructor
{
   strncpy( buffer, cp, sizeof(buffer) );
   foos[counter] = this;
   switch ( counter++ )
   {
      case 0:
      case 1:
         cout << ::foo_1.buffer << "," << ::foo_2.buffer << endl;
         break;
      default:
         cout << "Hello, world" << endl;
   }
}
// If a program using Foo.hh is linked with Foo1.o and Foo2.o, either
// ,two      or  one,     is written on standard output depending on
// one,two       one,two  the order of the files given to the linker.
```

18.7 Temporary Objects

Port. Rec. 16 Do not write code which is dependent on the lifetime of a temporary object.

Temporary objects are often created in C++, such as when functions return a value. Difficult errors may arise when there are pointers in temporary objects. Since the language does not define the life expectancy of temporary objects, it is never certain that pointers to them are valid when they are used.

One way of avoiding this problem is to make sure that temporary objects are not created. This method, however, is limited by the expressive power of the language and is not generally recommended.

The C++ standard may someday provide an solution to this problem. In any case, it is a subject for lively discussions in the standardization committee.

Example 70 Difficult error in a string class which lacks output operator

```
class String
{
   public:
      operator const char*() const;     // Conversion operator to const char*
      friend String operator+( const String& left, const String& right );
      // ...
};

String a = “This may go to ”;
String b = "h***!";
   // The addition of a and b generates a new temporary String object.
   // After it is converted to a char* by the conversion operator, it is
   // no longer needed and may be deallocated. This means that characters
   // which are already deallocated are printed to cout -> DANGEROUS!!
cout << a + b;
```

18.8 Pointer Arithmetic

Port. Rec. 17 Avoid using shift operations instead of arithmetic operations.

Port. Rec. 18 Avoid pointer arithmetic.

Pointer arithmetic can be portable. The operators == and != are defined for all pointers of the same type, while the use of the operators <, >, <=, >= are portable only if they are used between pointers which point into the same array.

19 References

[1] The Annotated C++ Reference Manual – Bjarne Stroustrup/Margareth Ellis[ARM]

– Addison Wesley 1990, ISBN 0-201-51459-1

This book forms the basis of the work in the ANSI-C++ committee.

[2] C++ Primer, Second Edition – Stanley B. Lippman

– Addison Wesley 1991, ISBN 0-201-54848-8

Very good for learning the basics of C++.

[3] The C++ Programming Language, Second Edition – Bjarne Stroustrup

– Addison Wesley 1991, ISBN 0-201-53992-6

This second edition has been completely updated with the current (and future) language definition. It will most certainly become a standard reference book.

[4] Advanced C++ Programming Styles and Idioms – James O. Coplien

– Addison Wesley 1992, ISBN 0-210-54855-0

Possibly the most advanced book on how to use C++. Contains many tricks and tips.

[5] Object-oriented Software Construction – Bertrand Meyer

– Prentice Hall 1988, ISBN 0-13-629049-3 or 0-13-629031-0 PBK

Somewhat of a classic work. Examples are written in Eiffel.

[6] Data Abstraction and Object-Oriented Programming in C++ – Keith E. Gorlen, Sanford M. Orlow, Perry S. Plexico

– John Wiley & Sons 1990, ISBN 0 471 92346 X pbk or 0 471 92751 1

The book that describes the class library NIH. Includes many good examples.

[7] Object-Oriented Design with Applications – Grady Booch

– Benjamin/Cummings 1991, ISBN 0-8053-0091-0

Treats the design and implementation of software in various object-oriented languages.

[8] Recommended C Style and Coding Standards

– Bell Labs, Zoology Computer Systems University of Toronto, CS University of Washington, November 18, 1989.

A collection of rules for programming in C. Contains a good section on portability.

[9] A Guide to Natural Naming – Daniel Keller

– ETH, Projekt-Zentrum IDA, CH-8092 Zurich, Switzerland

A guide on how to choose good names for functions and variables. Not adapted to object-oriented programming.

[10] Advanced C++ – Jonathan E. Shopiro

Binder with material from course held in Lund (Sweden) from June 4 to June 7, 1991. Filled with warnings and tips.

[11] Objektorienterad programmering och biblioteksuppbyggnad i C++ – Martin Carrol

Material from course held in Stockholm (Sweden) on April 18, 1991. Presents useful viewpoints on problems which may arise when designing a class library.

[12] Automatic Detection of C++ Programming Errors: Initial Thoughts on a lint++ – Scott Myers/Moises Lejter

– Usenix C++ Conference Proceedings, Spring 91

Article which describes some programming rules for C++.

[13] Code-Style Prescriptions – Carl R. Dickler

– Unix Review, 9(9), 1991, pages 41-45

Article which describes a number of programming rules for C and which discusses why programming rules are needed.

20 Summary of Rules

Rule 0 Every time a rule is broken, this must be clearly documented.

Rule 1 Include files in C++ always have the file name extension ".hh".

Rule 2 Implementation files in C++ always have the file name extension ".cc".

Rule 3 Inline definition files always have the file name extension ".icc".

Rule 4 Every file that contains source code must be documented with an introductory comment that provides information on the file name and its contents.

Rule 5 All files must include copyright information.

Rule 6 All comments are to be written in English.

Rule 7 Every include file must contain a mechanism that prevents multiple inclusions of the file.

Rule 8 When the following kinds of definitions are used (in implementation files or in other include files), they must be included as separate include files:

- classes that are used as base classes,
- classes that are used as member variables,
- classes that appear as return types or as ***argument types*** in function/member function prototypes.
- function prototypes for functions/member functions used in inline member functions that are defined in the file.

Rule 9 Definitions of classes that are only accessed via pointers (`*`) or references (`&`) shall not be included as include files.

Rule 10 **Never** specify relative UNIX names in **`#include`** directives.

Rule 11 Every implementation file is to include the relevant files that contain:

- declarations of types and functions used in the functions that are implemented in the file.
- declarations of variables and member functions used in the functions that are implemented in the file.

Rule 12 The identifier of every globally visible class, enumeration type, type definition, function, constant, and variable in a class library is to begin with a prefix that is unique for the library.

Rule 13 The names of variables, constants, and functions are to begin with a lowercase letter.

Rule 14 The names of abstract data types, structures, **`typedefs`**, and enumerated types are to begin with an uppercase letter.

Rule 15 In names which consist of more than one word, the words are written together and each word that follows the first is begun with an uppercase letter.

Rule 16 Do not use identifiers which begin with one or two underscores ('_' or '__').

Rule 17 A name that begins with an uppercase letter is to appear directly after its prefix.

Rule 18 A name that begins with a lowercase letter is to be separated from its prefix using an underscore ('_').

Rule 19 A name is to be separated from its suffix using an underscore ('_').

Rule 20 The public, protected, and private sections of a class are to be declared in that order (the public section is declared before the protected section which is declared before the private section).

Rule 21 No member functions are to be defined within the class definition.

Rule 22 Never specify public or protected member data in a class.

Rule 23 A member function that does not affect the state of an object (its instance variables) is to be declared **`const`**.

Rule 24 If the behaviour of an object is dependent on data outside the object, this data is not to be modified by const member functions.

Rule 25	A class which uses “new” to allocate instances managed by the class, must define a ***copy constructor***.
Rule 26	All classes which are used as base classes and which have virtual functions, must define a virtual destructor.
Rule 27	A class which uses “new” to allocate instances managed by the class, must define an ***assignment operator***.
Rule 28	An assignment operator which performs a destructive action must be protected from performing this action on the object upon which it is operating.
Rule 29	A public member function must never return a non-const reference or pointer to member data.
Rule 30	A public member function must never return a non-const reference or pointer to data outside an object, unless the object shares the data with other objects.
Rule 31	Do not use unspecified function arguments (ellipsis notation).
Rule 32	The names of formal arguments to functions are to be specified and are to be the same both in the function declaration and in the function definition.
Rule 33	Always specify the return type of a function explicitly.
Rule 34	A public function must never return a reference or a pointer to a local variable.
Rule 35	Do not use the preprocessor directive **`#define`** to obtain more efficient code; instead, use inline functions.
Rule 36	Constants are to be defined using **`const`** or **`enum`**; never using #**`define`**.
Rule 37	Avoid the use of numeric values in code; use symbolic values instead.
Rule 38	Variables are to be declared with the smallest possible ***scope***.
Rule 39	Each variable is to be declared in a separate declaration statement.
Rule 40	Every variable that is declared is to be given a value before it is used.
Rule 41	If possible, always use initialization instead of assignment.
Rule 42	Do not compare a pointer to NULL or assign NULL to a pointer; use 0 instead.
Rule 43	Never use ***explicit*** type conversions (casts).
Rule 44	Do not write code which depends on functions that use implicit type conversions.
Rule 45	Never convert pointers to objects of a derived class to pointers to objects of a virtual base class.
Rule 46	Never convert a **`const`** to a non-**`const`**.
Rule 47	The code following a **`case`** label must always be terminated by a **`break`** statement.
Rule 48	A **`switch`** statement must always contain a **`default`** branch which handles unexpected cases.
Rule 49	Never use **`goto`**.
Rule 50	Do not use **`malloc`**, **`realloc`** or **`free`**.
Rule 51	Always provide empty brackets (“**`[]`**”) for **`delete`** when deallocating arrays.

21 Summary of Recommendations

Rec. 1 Optimize code only if you know that you have a performance problem. Think twice before you begin.

Rec. 2 If you use a C++ compiler that is based on Cfront, always compile with the +w flag set to eliminate as many warnings as possible.

Rec. 3 An include file should not contain more than one class definition.

Rec. 4 Divide up the definitions of member functions or functions into as many files as possible.

Rec. 5 Place machine-dependent code in a special file so that it may be easily located when porting code from one machine to another.

Rec. 6 Always give a file a name that is unique in as large a context as possible.

Rec. 7 An include file for a class should have a file name of the form <class name> + extension. Use uppercase and lowercase letters in the same way as in the source code.

Rec. 8 Write some descriptive comments before every function.

Rec. 9 Use **`//`** for comments.

Rec. 10 Use the directive **`#include "filename.hh"`** for user-prepared include files.

Rec. 11 Use the directive **`#include <filename.hh>`** for include files from libraries.

Rec. 12 Every implementation file should declare a local constant string that describes the file so the UNIX command **`what`** can be used to obtain information on the file revision.

Rec. 13 Never include other files in an ".icc" file.

Rec. 14 Do not use typenames that differ only by the use of uppercase and lowercase letters.

Rec. 15 Names should not include abbreviations that are not generally accepted.

Rec. 16 A variable with a large scope should have a long name.

Rec. 17 Choose variable names that suggest the usage.

Rec. 18 Write code in a way that makes it easy to change the prefix for global identifiers.

Rec. 19 Encapsulate global variables and constants, enumerated types, and typedefs in a class.

Rec. 20 Always provide the ***return type*** of a function explicitly.

Rec. 21 When declaring functions, the leading parenthesis and the first argument (if any) are to be written on the same line as the function name. If space permits, other arguments and the closing parenthesis may also be written on the same line as the function name. Otherwise, each additional argument is to be written on a separate line (with the closing parenthesis directly after the last argument).

Rec. 22 In a function definition, the ***return type*** of the function should be written on a separate line directly above the function name.

Rec. 23 Always write the left parenthesis directly after a function name.

Rec. 24 Braces ("{}") which enclose a block are to be placed in the same column, on separate lines directly before and after the block.

Rec. 25 The flow control primitives **`if`**, **`else`**, **`while`**, **`for`** and **`do`** should be followed by a block, even if it is an empty block.

Rec. 26 The dereference operator '**`*`**' and the address-of operator '**`&`**' should be directly connected with the type names in declarations and definitions.

Rec. 27 Do not use spaces around '**`.`**' or '**`->`**', nor between unary operators and operands.

Rec. 28 Use the c++ mode in GNU Emacs to format code.

Rec. 29 Access functions are to be inline.

Rec. 30	Forwarding functions are to be inline.
Rec. 31	Constructors and destructors must not be inline.
Rec. 32	Friends of a class should be used to provide additional functions that are best kept outside of the class.
Rec. 33	Avoid the use of global objects in constructors and destructors.
Rec. 34	An assignment operator ought to return a ***const*** reference to the assigning object.
Rec. 35	Use operator overloading sparingly and in a uniform manner.
Rec. 36	When two operators are opposites (such as **`==`** and **`!=`**), it is appropriate to define both.
Rec. 37	Avoid inheritance for parts-of relations.
Rec. 38	Give derived classes access to class type member data by declaring protected access functions.
Rec. 39	Do not attempt to create an instance of a class template using a type that does not define the member functions which the class template, according to its documentation, requires.
Rec. 40	Take care to avoid multiple definition of overloaded functions in conjunction with the instantiation of a class template.
Rec. 41	Avoid functions with many arguments.
Rec. 42	If a function stores a pointer to an object which is accessed via an argument, let the argument have the type pointer. Use reference arguments in other cases.
Rec. 43	Use constant references (**`const &`**) instead of call-by-value, unless using a pre-defined data type or a pointer.
Rec. 44	When overloading functions, all variations should have the same semantics (be used for the same purpose).
Rec. 45	Use **`inline`** functions when they are really needed.
Rec. 46	Minimize the number of temporary objects that are created as return values from functions or as arguments to functions.
Rec. 47	Avoid long and complex functions.
Rec. 48	Pointers to pointers should whenever possible be avoided.
Rec. 49	Use a **`typedef`** to simplify program syntax when declaring function pointers.
Rec. 50	The choice of loop construct (**`for`**, **`while`** or **`do-while`**) should depend on the specific use of the loop.
Rec. 51	Always use **`unsigned`** for variables which cannot reasonably have negative values.
Rec. 52	Always use inclusive lower limits and exclusive upper limits.
Rec. 53	Avoid the use of **`continue`**.
Rec. 54	Use **`break`** to exit a loop if this avoids the use of flags.
Rec. 55	Do not write logical expressions of the type **`if(test)`** or **`if(!test)`** when **`test`** is a pointer.
Rec. 56	Use parentheses to clarify the order of evaluation for operators in expressions.
Rec. 57	Avoid global data if at all possible.
Rec. 58	Do not allocate memory and expect that someone else will deallocate it later.
Rec. 59	Always assign a new value to a pointer that points to deallocated memory.
Rec. 60	Make sure that fault handling is done so that the transfer to exception handling (when this is available in C++) may be easily made.
Rec. 61	Check the fault codes which may be received from library functions even if these functions seem foolproof.

22 Summary of Portability Recommendations

Port. Rec. 1	Avoid the direct use of pre-defined data types in declarations.
Port. Rec. 2	Do not assume that an **`int`** and a **`long`** have the same size.
Port. Rec. 3	Do not assume that an **`int`** is 32 bits long (it may be only 16 bits long).
Port. Rec. 4	Do not assume that a **`char`** is **`signed`** or **`unsigned`**.
Port. Rec. 5	Always set **`char`** to **`unsigned`** if 8-bit ASCII is used.
Port. Rec. 6	Be careful not to make type conversions from a “shorter” type to a “longer” one.
Port. Rec. 7	Do not assume that pointers and integers have the same size.
Port. Rec. 8	Use explicit type conversions for arithmetic using signed and unsigned values.
Port. Rec. 9	Do not assume that you know how an instance of a data type is represented in memory.
Port. Rec. 10	Do not assume that **`long`**s, **`float`**s, **`double`**s or **`long double`**s may begin at arbitrary addresses.
Port. Rec. 11	Do not depend on underflow or overflow functioning in any special way.
Port. Rec. 12	Do not assume that the operands in an expression are evaluated in a definite order.
Port. Rec. 13	Do not assume that you know how the invocation mechanism for a function is implemented.
Port. Rec. 14	Do not assume that an object is initialized in any special order in constructors.
Port. Rec. 15	Do not assume that static objects are initialized in any special order.
Port. Rec. 16	Do not write code which is dependent on the lifetime of a temporary object.
Port. Rec. 17	Avoid using shift operations instead of arithmetic operations.
Port. Rec. 18	Avoid pointer arithmetic.