Programming in C++

Rules and Recommendations

Ellemtel Telecommunication Systems Laboratories
Box 1505
125 25 Älvsjö
Sweden
Tel: int + 46 8 727 30 00

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 rule 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.