Part I

Preliminaries

Chapters 1 through 4 present an introduction to C++ that provides the basis for understanding the rest of the material in this book. This part also provides professional programmers with insight into how their managers and technical leaders view life. This material is intended to help developers understand how their organization works so they can participate more fully in the decision-making process.

Chapter 1

Introduction

FAQ 1.01

What is the purpose of this chapter?

To explain what the book is all about, how it is related to the electronic FAQ and the first edition, and what conventions are used.

This chapter discusses the purpose of the book and the conventions it follows. This chapter also discusses our approach to FAQs and why you should buy this book if you have the first edition or have access to the electronic FAQ.

FAQ 1.02

What are C++ FAQs?

Frequently Asked Questions that should be asked about object-oriented programming and C++.

Each FAQ provides specific guidance in the form of in-depth answers. Many FAQs also provide a complete, working program that illustrates the principles espoused by the FAQ. The word FAQs is pronounced like “facts.”

These FAQs aren’t necessarily questions people have asked; rather, they are the questions people should ask. Although we never say it publicly, most of these FAQs are based on dumb things we see people do on a fairly regular basis. We got tired of explaining the same fundamental notions over and over again and decided to write them down in this book.

On the other hand, you have taken a step toward OO and C++ competence by purchasing this guidebook; now take the next step by reading and understanding its message.

FAQ 1.03

Who is the target audience for this book?

Professional software developers.

This book is aimed at developers including programmers, architects, and designers. It is a fine way for the experienced programmer to learn object-oriented C++. This book is not for beginners who are just learning to program since it assumes previous programming background. Familiarity with C wouldn't hurt but is not absolutely necessary.

FAQ 1.04

Is this a book about C++ per se?

This is a C++ book with a twist.

This book focuses on the object-oriented aspects of C++. Thus, whenever you see the word "C++," you should assume that the words "object-oriented" are present (and we'll occasionally inject the words "object-oriented" as a reminder to the reader).

This book focuses on practical ways to use C++; it does not explore all of the dark corners of the language beloved by "language lawyers." In this way, this book is not the traditional C++ book written from the perspective of the language and stressing the syntax and features of C++ in all their gory detail. Instead, this book concentrates on the key aspects of C++ (such as its OO features) and how to apply them effectively. Another reason for this approach is that the language is so large that it is hard for developers to understand what is relevant and how to apply it.

In this vein, one of the main contributions of this book is to focus on the moral use of C++ rather than simply describing the legal use of C++. In this context, using C++ morally means adhering to a programming discipline (i.e., a subset of all possible combinations of all the constructs of C++) that is relatively risk-free (whereas using C++ legally simply refers to any use of the language that the compiler accepts). We have found that many of the problems that developers run into stem from trying to combine

C++ features in incompatible and seemingly random ways; therefore using C++ morally is vital to using C++ effectively.

This book also tries to bridge the gap between software architecture and OO design and C++ programming (see Chapter 4).

FAQ 1.05

Why do developers need a guidebook for C++ and OO technology?

Learning to use C++ and OO properly is a long journey with many pitfalls.

Because of the sophistication and complexity of C++, developers need a road map that shows how to use the language properly. For example, *inheritance* is a powerful facility that can improve the clarity and extensibility of software, but it can also be abused in ways that result in expensive design errors.

The field of object-oriented technology is large, evolving, and heterogeneous. Under these circumstances, a guidebook is essential. These FAQs cover the latest innovations so that you don't have to stumble around for years learning the same lessons others have already learned. The FAQs also expose incorrect and questionable practices.

To be effective, programmers need to understand the language features and how the features of the language can be combined. For example, pointer arithmetic and the *is-a* conversion (see FAQ 2.24) are both useful, but combining them has some subtle edge effects that can cause big problems; see FAQ 8.16. Similar comments apply when combining overloading and overriding (FAQ 29.02), overriding and default parameters, abstract base classes and assignment (FAQ 24.05), and so on. So it is not enough to understand each feature of C++.

FAQ 1.06

What kind of guidance is given in the answers to these FAQs?

Explanations of language features, directions for using these features properly, and guidelines indicating programming practices to avoid.

The FAQs can be divided into roughly three categories:

1. FAQs that explain what a particular language feature is and how to use it in compliance with C++ semantics.

2. FAQs that explain how to use C++ properly. Some of these answers deal with only a single language feature, while others explain how to use several different language features in concert. Combining language features allows sophisticated designs that can simultaneously satisfy multiple technical requirements and business goals.
3. FAQs that expose poor programming practices. These show design and programming practices that are legal in C++ but should be avoided because they can lead to programs that are bug-ridden, hard to comprehend, expensive to maintain, difficult to extend, and lacking reuse value.

FAQ 1.07

What is the electronic FAQ and why buy this book when the electronic FAQ is free?

The electronic FAQ is a set of C++ questions and answers, originally prepared and distributed on the Internet by Marshall Cline. The Internet version is currently updated and distributed by Marshall and is available through the news group comp.lang.c++. This book has substantially more material than the electronic FAQ.

This book and the electronic FAQ were inspired by a seemingly unquenchable thirst among C++ developers for more and better information about C++ through comp.lang.c++. Addison-Wesley decided to provide an expanded form of that information in book format.

This book covers a broader range of topics and goes into greater depth than the electronic FAQ. It provides deeper coverage of the key points with extensive new examples.

Most of the programming examples are working, stand-alone programs, complete with their own `main()`, all necessary `#include` files, and so on. All examples have been compiled directly from the source text of the book; those that are complete programs have also been run.

FAQ 1.08

Why should you buy this edition if you already have a copy of the first edition?

new

Because the world has changed and you want to keep up with technology.

The OO world and the C++ language have changed significantly in the last few years. There are new language constructs such as Run Time Type Identification (RTTI) and namespaces. The Standard Template Library (STL) is a massive addition to the C++ body of essential knowledge. Design notation has apparently standardized on the Unified Modeling Language (UML). Java, CORBA, and ActiveX are now topics that every C++ developer needs to understand. The goal of this second edition is to bring you up to speed on all of these new developments while still keeping the pithy style and FAQ format that was so well received in the first edition.

Finally, the second edition is much more self-contained than the first, with lots of syntax and semantics. We've appreciated all your comments and suggestions and have tried to accommodate them wherever possible.

FAQ 1.09

What conventions are used in this book?

The undecorated word *inheritance* means "public inheritance." Private or protected inheritance is referred to explicitly.

Similarly the undecorated term *derived class* means "public derived class." Derived classes produced via private or protected inheritance are explicitly designated "private derived class" or "protected derived class," respectively.

The class names `Base` and `Derived` are used as hypothetical class names to illustrate the general relationship between a base class and one of its (publicly) derived classes.

The term *out-lined function* indicates a function that is called via a normal `CALL` instruction. In contrast, when an *inlined function* is invoked, the compiler inserts the object code for that function at the point-of-call.

The term *remote ownership* is used when an object contains a pointer to another object that the first object is responsible for deleting. The default destruction and copy semantics for objects that contain remote ownership are incorrect, so explicit controls are needed.

To allow compilation while simplifying the presentation to the reader, examples that use the standard library have a line that says `using namespace std;`. This dumping of the entire standard namespace is acceptable as a short-term conversion technique or as a pedagogical aid, but its use in production systems is controversial. Most authorities recommend introducing class names as needed or using the `std::` qualifier.

The term *OO* is used as an abbreviation for "object-oriented."

The term *method* is used as a synonym for "member function."

`NULL` is used rather than 0 to make the code more readable. Organizational standards and guidelines should be consulted before the reader continues this practice.

The term *C programming language* refers to the ISO version of C.

The compiler is assumed (per the C++ Standard) to insert an implicit `return 0;` at the end of `main()`.

The intrinsic data type `bool` is used, which has literal values `true` and `false`. For compilers that don't have a built-in `bool` type, insert the following at the beginning of each example:

```
typedef char bool; const bool false = 0; const bool true = 1;
```

The expression `new MyClass`, where `MyClass` is some type, is assumed to throw an exception if it runs out of memory—it never returns `NULL`. Most compilers implement this correctly, but some do not.

Most examples use `protected:` data rather than `private:` data. In the real world, this is appropriate for most developers and most applications, but framework developers probably should not use `protected:` data, since this would create a data coupling between the derived classes and the `protected:` data of the base class. In general, framework developers should use `private:` data with `protected:` access functions.

Type names (names of classes, structs, unions, enums, and typedefs) start with a capital letter; preprocessor symbols are all capitals; all other identifiers start with a lowercase letter. Data member names and class-scoped enumerations end with a single underscore.

It is assumed that the file extensions `.cpp` and `.hpp` are appropriate. Some compilers use a different convention.

Universal Modeling Language (UML) notation is used to express design relationships.

The following priorities were used in designing the examples: (1) unity of purpose, (2) compactness, and (3) self-contained functionality. In other words, each example demonstrates one basic point or technique, is as short as possible, and, if possible, is a complete, working program. The examples are not intended for plug-in reuse in industrial-strength settings because balancing the resultant (subtle) tradeoffs would conflict with these priorities.

To avoid complicating the discussions with finding the optimal balance between the use of `virtual` and `inline` for member functions, `virtual` is used more often than strictly necessary (see FAQ 21.15). To achieve compactness, some member functions are defined in the class body even if they wouldn't normally be `inline` or even if moving them down to the bottom of a header file would improve specification (see FAQ 6.05). Uncalled functions are often left undefined. Some functions that are called are also undefined, since compactness is a higher priority than self-contained functionality. Also for compactness, examples are not wrapped in preprocessor symbols that prevent multiple expansions (see FAQ 2.16).

The examples put the `public:` part at the beginning of the class rather than at the end of the class. This makes it easier for those who simply want to use the class as opposed to those who want to go in and change the internal implementation of the class. This is normally the right tradeoff since a class is normally used a lot more often than it is changed.

It is assumed that the C++ compiler and standard library are both compliant with the Standard and work correctly. In the real world, this is probably not a safe assumption, and you should be cautious.