1

A Tour of C++

The first thing we do, let´s
kill all the language lawyers.
– Henry VI, part II

This chapter is a quick tour of the major concepts and features of the C++ programming language. Its purpose is to give a fairly high-level acquaintance with C++. Detailed explanations of language features and techniques are postponed to later chapters. The discussion focuses on the language features supporting data abstraction and object-oriented programming, but it also briefly introduces the key features used for procedural programming.

1.1 Introduction

The C++ programming language is designed to

- be a better C
- support data abstraction
- support object-oriented programming

This chapter explains what this means without going into the finer details of the language definition:

§1.2 informally distinguishes ‘‘procedural programming,’’ ‘‘modular programming,’’ ‘‘data abstraction’’ and ‘‘object-oriented programming’’ from each other and presents the mechanisms that are essential for supporting these styles of programming. The styles of programming typically used in C are discussed under the headings ‘‘procedural programming’’ and ‘‘modular programming.’’ C++ is ‘‘a better C’’ because C++ provides better support for these styles of programming than C does. C++ does so without loss of generality or efficiency compared with C while remaining almost

completely a superset of C. The discussion of data abstraction and object-oriented programming equates ‘‘support for data abstraction’’ with the ability to define and use new types and equates ‘‘support for object-oriented programming’’ with the ability to express type hierarchies.

§1.3 presents the basic facilities for procedural and modular programming provided by C++. In particular, it introduces functions, pointers, iteration, input/output, and the notion of programs composed out of separately compiled modules. Chapters 2, 3, and 4 cover these facilities in detail.

§1.4 presents features needed to make data abstraction effective. In particular, it introduces classes, the simplest access control mechanism, constructors and destructors, operator overloading, user-defined type conversions, exception handling, and templates. Chapters 5, 7, 8, and 9 cover these facilities in detail.

§1.5 discusses facilities needed to support object-oriented programming. In particular, derived classes and virtual functions are introduced and some implementation issues are presented. Chapter 6 covers these facilities in detail.

§1.6 discusses some limits to perfection imposed on general-purpose programming languages in general, and C++ in particular, by efficiency demands, competing demands from diverse application areas, education, and the need to function in traditional compile and run-time environments.

Should you find any part of this chapter incomprehensible, you are strongly encouraged to read the corresponding chapters and then return to this overview with a more detailed understanding of the basic language features. The purpose of this chapter is to give you a general overview of C++, *not* to provide you with the information necessary to start programming in it.

1.2 Programming Paradigms

Object-oriented programming is a technique for programming – a paradigm for writing ‘‘good’’ programs for a set of problems. If the term ‘‘object-oriented programming language’’ means anything, it must mean a programming language that provides mechanisms that support the object-oriented style of programming well.

There is an important distinction here. A language is said to *support* a style of programming if it provides facilities that makes it convenient (reasonably easy, safe, and efficient) to use that style. A language does not support a technique if it takes exceptional effort or skill to write such programs; it merely *enables* the technique to be used. For example, you can write structured programs in Fortran and object-oriented programs in C, but it is unnecessarily hard to do so because these languages do not directly support those techniques.

Support for a paradigm comes not only in the obvious form of language facilities that allow direct use of the paradigm, but also in the more subtle form of compile-time and/or run-time checks against unintentional deviation from the paradigm. Type checking is the most obvious example of this; ambiguity detection and run-time

checks can be used to extend linguistic support for paradigms. Extra-linguistic facilities such as standard libraries and programming environments can also provide significant support for paradigms.

One language is not necessarily better than another because it possesses a feature the other does not. There are many examples to the contrary. The important issue is not so much what features a language possesses, but that the features it does possess are sufficient to support the desired programming styles in the desired application areas:

[1] All features must be cleanly and elegantly integrated into the language.
[2] It must be possible to use features in combination to achieve solutions that would otherwise have required extra, separate features.
[3] There should be as few spurious and ‘‘special-purpose’’ features as possible.
[4] A feature should be such that its implementation does not impose significant overheads on programs that do not require it.
[5] A user need only know about the subset of the language explicitly used to write a program.

The first principle is an appeal to aesthetics and logic, the next two are expressions of the ideal of minimalism, and the last two principles can be summarized as ‘‘what you don’t know won’t hurt you.’’

C++ was designed to support data abstraction and object-oriented programming in addition to traditional C programming techniques under these constraints. It was *not* meant to force any particular programming style upon all users.

Let’s now consider some programming styles and the key language mechanisms necessary for supporting them. The presentation of language features is not intended to be exhaustive.

1.2.1 Procedural Programming

The original (and probably still the most commonly used) programming paradigm is:

> *Decide which procedures you want;*
> *use the best algorithms you can find.*

The focus is on the processing, the algorithm needed to perform the desired computation. Languages support this paradigm by providing facilities for passing arguments to functions and returning values from functions. The literature related to this way of thinking is filled with discussion of ways of passing arguments, ways of distinguishing different kinds of arguments, different kinds of functions (procedures, routines, macros, ...), etc. Fortran is the original procedural language; Algol60, Algol68, Pascal, and C are later inventions in the same tradition.

A typical example of ‘‘good style’’ is a square root function. Given an argument, it produces a result. To do this, it performs a well understood mathematical computation:

```
double sqrt(double arg)
{
    // the code for calculating a square root
}

void some_function()
{
    double root2 = sqrt(2);
    // ...
}
```

The double slash, `//`, begins a comment that extends to the end of the line.

From the point of view of program organization, functions are used to create order in a maze of algorithms.

1.2.2 Modular Programming

Over the years, the emphasis in the design of programs has shifted away from the design of procedures and toward the organization of data. Among other things, this reflects an increase in program size. A set of related procedures with the data they manipulate is often called a *module*. The programming paradigm becomes:

> *Decide which modules you want;*
> *partition the program so that data is hidden in modules.*

This paradigm is also known as the ‘‘data-hiding principle.’’ Where there is no grouping of procedures with related data, the procedural programming style suffices. Also, the techniques for designing ‘‘good procedures’’ are now applied for each procedure in a module. The most common example of a module is the definition of a stack module. The main problems that have to be solved are:

[1] Provide a user interface for the stack (for example, functions `push()` and `pop()`).

[2] Ensure that the representation of the stack (for example, an array of elements) can be accessed only through this user interface.

[3] Ensure that the stack is initialized before its first use.

Modula-2 directly supports this notion; C merely enables its use. Here is a plausible external interface for a C-style stack module: