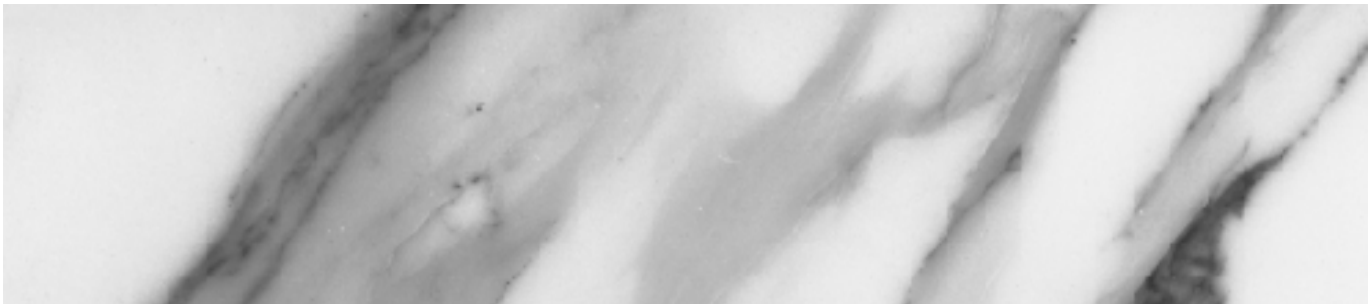
CORE C++

A Software Engineering Approach

Part 1

INTRODUCTION TO PROGRAMMING WITH C++



The first part of this book is about foundations of programming with C++. As everybody knows, C++ is an object-oriented language. But what does this mean? Why is using an object-oriented programming language better than using a traditional non-object-oriented language? What should you pay attention to while programming so that you reap the benefits of object orientation? Often, people take the object-oriented approach for granted, and this reduces the effectiveness of its use.

The first chapter answers these questions. It is all about breaking the program into parts. A large program has to be written as a collection of relatively independent, but communicating and cooperating, components. If, however, you break apart what should be kept together, you introduce excessive communications and dependencies between parts of the program, and the code becomes more difficult to reuse and maintain. If you leave together, in the same component, what can and should be broken into separate pieces, you wind up with complex and confusing code which is—guess what—difficult to reuse and maintain.

There is no magic in using objects. Using them in and of themselves brings no benefits. However, thinking about your program in terms of objects helps you to avoid these two dangers: breaking apart what should belong together and keeping together what should be put into separate parts. Chapter 1, "Object-Oriented Approach: What's So Good About It?" discusses these issues—it shows which problems should be solved with the use of the object-oriented approach and how the object-oriented approach solves these problems.

Chapter 2, "Getting Started Quickly: A Brief Overview of C++," gives you a brief introduction to the language, including objects. The introduction is high level only. (You have to read other chapters of the book to see the details.) Nevertheless, this chapter covers enough to enable you to write simple C++ programs and prepares you for the detailed study of the strong and weak features of C++.

Other chapters in Part 1 present the basic non-object-oriented features of the language. According to the promise I made in Chapter 1, I pay particular attention to writing reusable and maintainable code. For each C++ construct, I explain how to use and how not to use it. Even though I do not discuss objects yet, the presentation becomes quite complex, especially in Chapter 6, "Memory Management: The Stack and The Heap." After all, C++ is a complex language. Skip topics that you find obscure and come back to them later, when you have more time to concentrate on coding details.

OBJECT-ORIENTED APPROACH: WHAT'S SO GOOD ABOUT IT?

Topics in this Chapter

- The Origins of the Software Crisis
- Remedy 1: Eliminating Programmers
- Remedy 2: Improved Management Techniques
- Remedy 3: Designing a Complex and Verbose Language
- The Object-Oriented Approach: Are We Getting Something for Nothing?
- Characteristics of the C++ Programming Language
- Summary

Chapter 1

The object-oriented approach is sweeping all areas of software development. It opens new horizons and offers new benefits. Many developers take it for granted that these benefits exist and that they are substantial. But what are they? Do they come automatically, just because your program uses objects rather than functions?

In this chapter, I will first describe why we need the object-oriented approach. Those of you who are experienced software professionals, can skip this description and go on directly to the explanation of why the object-oriented approach to software construction is so good.

Those of you who are relatively new to the profession should read the discussion of the software crisis and its remedies to make sure you understand the context of the programming techniques I am going to advocate in this book. It should give you a better understanding of what patterns of C++ coding contribute to the quality of your program, what patterns inhibit quality, and why.

Given the abundance of low quality C++ code in industry, this is very important. Many programmers take it for granted that using C++ and its classes delivers all the advantages, whatever they are, automatically. This is not right. Unfortunately, most C++ books support this incorrect perception by concentrating on C++ syntax and avoiding any discussion of the quality of C++ code. When developers do not know what to aim for in C++ code, they wind up with object-oriented programs that are built the old way. These pro-

grams are no better than traditional C, PL/I (or whatever—insert your favorite language) programs and are as difficult to maintain.

The Origins of the Software Crisis

The object-oriented approach is yet another way to fight the so-called software crisis in industry: frequent cost overruns, late or canceled projects, incomplete system functionality, and software errors. The negative consequences of errors in software range from simple user inconvenience to not-so-simple economic losses from incorrectly recorded transactions. Ultimately, software errors pose dangers to human lives and cause mission failures. Correction of errors is expensive and often results in skyrocketing software costs.

Many experts believe that the reason for software crisis is the lack of standard methodology: The industry is still too young. Other engineering professions are much older and have established techniques, methodologies, and standards.

Consider, for example, the construction industry. In construction, standards and building codes are in wide use. Detailed instructions are available for every stage of the design and building process. Every participant knows what the expectations are and how to demonstrate whether or not the quality criteria have been met. Warranties exist and are verifiable and enforceable. Consumer protection laws protect the consumer from unscrupulous or inept operators.

The same is true of newer industries, like the automobile industry or electrical engineering. In all these areas of human endeavor we find industry-wide standards, commonly accepted development and construction methodologies, manufacturer warranties, and consumer protection laws. Another important characteristic of these established industries is that the products are assembled from ready-made components. These components are standardized, thoroughly tested, and mass-produced.

Compare this with the state of the software industry. There are no standards to speak of. Of course, professional organizations are trying to do their best, coming up with standards ranging from specification writing to software testing to user-computer interfaces. But these standards only scratch the surface—there are no software development processes and methodologies that would be universally accepted, enforced, and followed. Mass-market software warranties are a joke: The consumer is lucky if the manufacturer is responsi-

ble for the cost of distribution medium. Return policies are nonexistent: If you open the box, you forfeit your right to ever get your money back.

The products are crafted by hand. There are no ready-made, off-the-shelf components. There is no universally accepted agreement what the components and the products should do. In its legal suit against Microsoft, the United States government got into an argument over the definition of the *operating system* and its components—whether the browser is part of the operating system or just another application, like a word processor, spreadsheet, or appointment scheduler. The operating system is as important to the computer as the ignition system to the car (probably even more so). But could you imagine a legal argument over the composition of the ignition system? We all know that when the technology required it, a carburetor was part of the ignition system. When technology changed, it was eliminated without public discussion.

The young age of the software industry has definitely contributed to the situation. Hopefully, some elements of this dismal picture will disappear in the future. However, this young age did not prevent software industry from becoming a multibillion dollar one that plays a crucial role in the economy. The Internet changed the way we do commerce and search for information. It also changed the stock market landscape beyond recognition.

Doomsayers heralded the Year 2000 problem as a major menace to the economy. It is not important for the purposes of this discussion whether or not those fears were justified. What *is* important is that the software industry has matured enough in terms of sheer power. If a software problem can potentially disrupt the very fabric of the Western society, it means that the industry plays an important role in the society. However, its technology lagging behind other industries, mostly because of the nature of the software development process.

Very few software systems are so simple that one person can specify it, build it according to the specification, use it for its intended purpose, and maintain it when the requirements change or errors are discovered. These simple systems have a limited purpose and a relatively short time span. It is easy to throw them away and start from scratch, if necessary; the investment of time and money is relatively small and can easily be written off.

Most software programs exhibit quite different characteristics. They are complex and cannot be implemented by one person. Several people (often, many people) have to participate in the development process and coordinate their efforts. When the job is divided among several people, we try to make these parts of the software system independent from each other, so that the developers can work on their individual pieces independently.

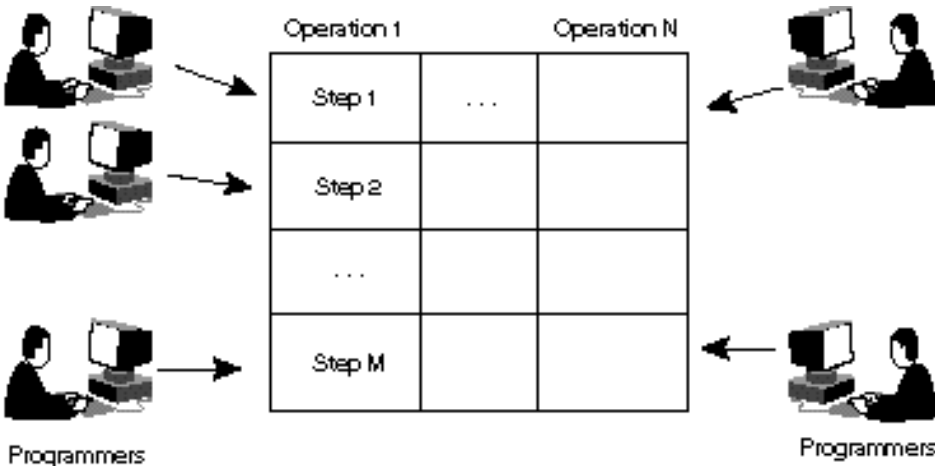
For example, we could break the functions of the software system into sep-

arate operations (place an order, add a customer, delete a customer, etc.). If those operations are too complex, implementing them by an individual programmer would take too long. So, we divide each operation into steps and substeps (verify customer, enter order data, verify customer credit rating, etc.) and assign each piece to an individual programmer for implementation (Figure 1–1).

The intent is to make system components independent from each other so that they can be developed by people working individually. But in practice, these separate pieces are not independent. After all, they are parts of the same system; so, they have to call each other, or work on shared data structures, or implement different steps of the same algorithm. Since the parts that different developers work on are not independent, the individual developers have to cooperate with each other: they write memos, produce design documents, send e-mail messages and participate in meetings, design reviews, or code walkthroughs. This is where the errors creep in—something gets misunderstood, something gets omitted, and something is not updated when related decisions are changed.

These complex systems are designed, developed, and tested over a long time. They are expensive. Some are very expensive. Many users depend on their operations. When requirements change, or errors or missing requirements are discovered, such systems cannot be replaced and thrown away—they often represent an investment too significant to be discarded.

Figure 1–1 Breaking the system into components.



These systems have to be maintained, and their code has to be changed. Changes made in one place in the code often cause repercussions in another place, and this requires more changes. If these dependencies are not noticed (and they *are* missed sometimes), the system will work incorrectly until the code is changed again (with further repercussions in other parts of the code). Since these systems represent a significant investment, they are maintained for a long time, even though the maintenance of these complex systems is also expensive and error-prone.

Again, the Year 2000 problem comes to mind. Many people are astonished by the fact that the programmers used only two last digits to represent the year. "In what world do these programmers live?" asks the public. "Don't they understand the implications of the switch from year 1999 to year 2000?" Yes, this is astonishing. But it is not the shortsightedness of the programmers that is astonishing, rather it is the longevity of the systems designed in the 1970s and 1980s. The programmers understood the implications of Year 2000 as well as any Y2K expert (or better). What they could not imagine in the 1970s and 1980s was that somebody would still be using their programs by the year 2000.

Yes, many organizations today pour exorbitant amounts of money into maintaining old software as if they are competing with others in throwing money away. The reason for this is that these systems are so complex that rebuilding them from scratch might be more expensive than continuing to maintain them.

This complexity is the most essential characteristic of most software systems. The problem domains are complex, managing the development process is complex, and the techniques of building software out of individual pieces manually are not adequate for this complexity.

The complexity of system tasks (this is what we call "the problem domain"), be it an engineering problem, a business operation, mass-marketed shrink-wrapped software, or an Internet application, makes it difficult and tedious to describe what the system should do for the users. The potential system users (or the marketing specialists) find it difficult to express their needs in a form that software developers can understand. The requirements presented by users that belong to different departments or categories of users often contradict each other. Discovering and reconciling these discrepancies is a difficult task. In addition, the needs of the users and marketers evolve with time, sometimes even in the process of formulating requirements, when the discussion of the details of system operations brings forth new ideas. This is why programmers often say that the users (and marketing specialists) do not know what they want. There are still few tools for capturing system requirements. This is why the requirements are usually produced as large volumes of text with drawings; this text is often poorly structured and is hard to comprehend; many statements in such requirements are vague, incomplete, contradictory, or open to interpretation.

The complexity of managing the development process stems from the need to coordinate activities of a large number of professionals, especially when the teams working on different parts of the system are geographically dispersed, and these parts exchange information or work on the same data. For example, if one part of the system produced data expressed in yards, the part of the system that uses this data should not assume that the data is expressed in meters. These consistency stipulations are simple, but numerous, and keeping them in mind is hard. This is why adding more people to a project does not always help. New people have to take over some of the tasks that the existing staff has been working on. Usually, the newcomers either take over some parts of the project that existing staff was supposed to work on later, or the parts of the project are further subdivided into subparts and are assigned to the newcomers.

The newcomers cannot become productive immediately. They have to learn about the decisions already made by the existing staff. The existing staff also slows down, because the only way for the newcomers to learn about the project is by talking to the existing staff and hence by distracting this staff from productive work.

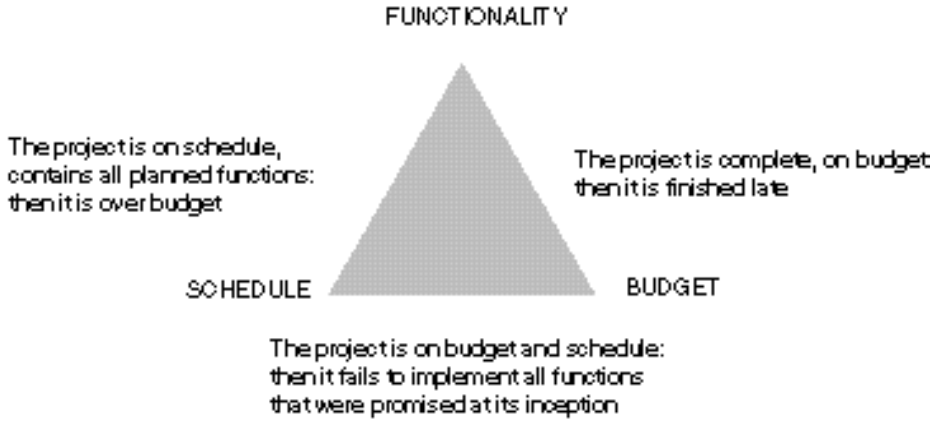


Figure 1–2 The mystery triangle of software projects.

Building software from individual pieces by hand adds to the problem: it is time consuming and prone to error. Testing is arduous, manual, and unreliable.

When I came to United States, my boss, John Convey, explained to me the situation in the following way. He drew a triangle where the vertices represented such project characteristics as schedule, budget, and system functionality (Figure 1–2). He said, "We cannot pull out all three. Something has to give in. If you implement all the system functionality on the budget, you will not be able to complete work on time, and you will ask for an extension. If you implement all functionality on schedule, chances are you will go over budget and will have to ask for more resources. If you implement the system on budget and on schedule (that does not happen often, but it is possible), well, then you will have to cut corners and implement only part of what you promised."

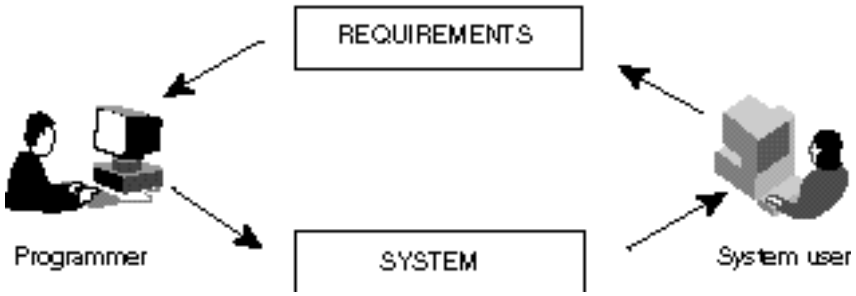
The problems shown in the triangle have plagued the software industry for a long time. Initial complaints about the software crisis were voiced in 1968. The industry developed several approaches to the problem. Let us take a brief look at a list of potential remedies.

Remedy 1: Eliminating Programmers

In the past, hardware costs dominated the cost of computer systems; software costs were relatively small. The bottleneck in system development seemed to be in communication between the programmers and software users, who tried to explain to the programmers what their business or engineering applications had to accomplish.

The programmers just could not get it right because they were trained as mathematicians, not in business, engineering, and so forth. They did not know business and engineering terminology. On the other hand, business and engineering managers did not know design and programming terminology; hence, when the programmers tried to explain what they understood about the requirements, communication breakdown would occur.

Figure 1–3 Communication breakdown between the user and the developer.



Similarly, the programmers often misunderstood the users' objectives, assumptions, and constraints. As a result, the users were getting not exactly what they wanted.

A good solution to the software crisis at that time seemed to be to get rid of programmers. Let the business and engineering managers write applications directly, without using programmers as intermediaries. However, the programmers at that time were using machine and assembly languages. These languages required intimate familiarity with the computer architecture and with the instruction set and were too difficult for managers and engineers who were not trained as programmers.

To implement this solution, it was necessary to design programming languages that would make writing software faster and easier. These languages should be simple to use, so that engineers, scientists, and business managers would be able to write programs themselves instead of explaining to the programmers what should be done.

FORTRAN and COBOL are the languages that were initially designed so that scientists, engineers, and business managers could write programs without communicating with the programmers.

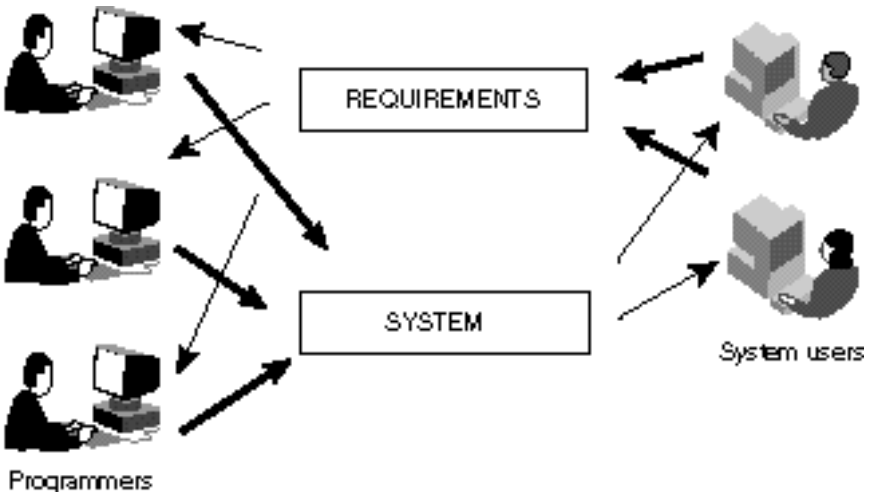
This approach worked fine. Many scientists, engineers, and business managers learned how to program and wrote their programs successfully. Some experts predicted that the programming profession would disappear soon. But this approach worked fine only for small programs that could be specified, designed, implemented, documented, used, and maintained by one person. It worked for programs that did not require cooperation of several (or many) developers and did not have to live through years of maintenance. The development of such programs did not require cooperation of developers working on different parts of the program.

Actually, Figure 1–3 is correct for small programs only. For larger programs, the picture is rather like Figure 1–4. Yes, communication problems between the user and the developers are important, but the communication problems between developers are much more important. It is communication between developers that cause misunderstandings, incompatibilities and errors regardless of who these developers are—professional programmers, professional engineers, scientists, or managers.

Even Figure 1–4 is an oversimplification. It shows only a few users, who specify the requirements and evaluate the effectiveness of the system. For most software projects, there are many users (marketing representative, salespeople) who specify the system, and more than one person who evaluates it (often, this is not the same person). Inconsistencies and gaps in specifying what the system should do (and in evaluating how well it does it) add to the communication problems among developers. This is especially true when a new system should perform some of its functions similarly to an existing system. This often leads to different interpretations among developers.

Another attempt to get rid of programmers was based on the idea of using *superprogrammers*. The idea is very simple. If ordinary programmers cannot create parts of the program so that these parts fit together without errors, let us find a capable individual who is so bright that he (or she) can develop the program alone. The superprogrammers' salaries have to be higher than the salaries of ordinary programmers, but they would be worth it. When the same

Figure 1–4 Communication breakdown between program developers.



person creates different parts of the same program, compatibility problems are less likely, and errors are less frequent and can be corrected quicker.

In reality, the superprogrammer could not work alone—there was too much mundane work that could be performed by ordinary people with smaller salaries. So, the superprogrammers had to be supported by technicians, librarians, testers, technical writers, and so on.

This approach met with limited success. Actually, each development project was an unqualified success—produced on schedule, under budget, and with complete functionality despite that pessimistic model on Figure 1–2. However, communication between the superprogrammer and the supporting cast was limited by the ordinary human capabilities of the supporting cast.

Also, the superprogrammers were not available for long-term maintenance; they either moved on to other projects, were promoted to managerial positions and stopped coding, or they moved on to other organizations in search of other challenges. When ordinary maintenance programmers were maintaining the code created by a superprogrammer, they had as much trouble as with the maintenance of code written by ordinary programmers, or even more trouble because superprogrammers tend to produce terse documentation: to a superprogrammer, even a complex system is relatively simple, and hence it is a waste to provide it with a lengthy description.

Nowadays, very few people promise that we will learn how to produce software systems without programmers. The industry turned to the search for the techniques that would produce high-quality programs with the use of people with ordinary capabilities. It found the solution in the use of management techniques.

Remedy 2: Improved Management Techniques

Since hardware costs continue to plummet, it is the cost of software development and maintenance that dominates the cost of computer systems rather than hardware cost. An expensive software system represents a significant investment that cannot be discarded easily and rewritten from scratch. Hence, expensive systems are maintained longer even though they are more expensive to maintain.

Continuing increase in hardware power opens new horizons; this entails further increases in code complexity and software costs (both for development and for maintenance).

This changes priorities in the software development process. Since the hopes for resolving this problem with the help of a few exceptionally bright individuals were dashed, the industry turned to methods of managing communication among ordinary individuals—users and developers and, especially, managing developers working on different parts of the project.

To facilitate the communications between users and developers, the industry employed the following two management techniques:

- the *waterfall* method (partitioning the development process into separate distinct stages)
- *rapid prototyping* (partial implementation for users' earlier feedback)

The Waterfall Method

There are several variations of the waterfall approach used in managing programming projects. They all include breaking the development process into sequential stages. A typical sequence of stages might include requirement definition, systems analysis, architectural design, detailed design, implementation and unit testing, integration testing, acceptance testing, and maintenance. Usually, a separate specialized team of developers performs each stage. After a period of trial use and the review of utility of the system, a new (or amended) set of requirements could be defined, and the sequence of steps might be repeated.

Transitions between stages are reflected in the project schedule as milestones related to a production of specific documents. The documents developed during each stage are ideally used for two purposes: for feedback from the previous stage to evaluate correctness of the development decisions and as an input document for the next stage of the project. This can be done either informally, by circulating the document among interested parties, or formally, by running design reviews and walkthrough meetings with representatives of each development team and the users.

For example, the requirement definition process produces the requirements document used as a feedback to the project originators or user representatives and as an input document for the systems analysts. Similarly, the systems analysis stage produces the detailed system specification used as a feedback to the users and as an input document for the design stages. This is the ideal. In practice, people who should provide the feedback might have

other pressing responsibilities and might devote only limited time to providing the feedback. This undermines the whole idea of quality control built into the process.

In addition, the further the project proceeds, the more difficult it becomes to get meaningful feedback from the users : The vocabulary becomes more and more computer oriented, the charts and diagrams use notation that is unfamiliar to the users, and design reviews often degenerate into a rubber stamp.

The advantage of this approach is its well-defined structure with clearly defined roles of each developer and specific deliverables at each milestone. A number of methods and tools exist for project planning and evaluating the duration and cost of different stages. This is especially important for large projects when we want to ensure that the project is moving in the right direction. The experience accumulated in one project helps in planning for subsequent similar projects.

The disadvantage is its excessive formalism, the possibility to hide from personal responsibility behind the group process, inefficiency, and the time lag of the feedback mechanism.

Rapid Prototyping

The rapid prototyping method takes the opposite approach. It eliminates the formal stages in favor of facilitating the feedback from the users. Instead of producing the formal specification for the system, the developers produce a system prototype that can be demonstrated to the users. The users try the prototype and provide the developers with much earlier and more specific feedback than in the waterfall approach. This sounds great, but is not easy to do for a large system—producing a rapid prototype might not be rapid at all and could easily approach the complexity and expense of producing the whole system. The users who should try the prototype might be burdened with other, more direct responsibility. They might lack skills in operating the system, they might lack skills in systematic testing, and they might lack skills (or time) in providing the feedback to the developers.

This approach is most effective for defining system user interface: menus, dialog boxes, text fields, control buttons, and other components of the human-computer interactions. Often, organizations try to combine both approaches. This works; often, it works well, but it does not eliminate the problem of communication among developers working on different parts of the system.

To improve communication among developers, a number of formal “structured” techniques were developed and tried with different degrees of success.

For writing system requirements and specifications, structured English (or whatever language is spoken by the developers) is used to facilitate understanding of the problem description and identification of the parts of the problem. For defining the general architecture and specific components of the system, structured design became popular with conjunction with such techniques as data flow diagrams and state transition diagrams. For low-level design, different forms of flowcharts and structured pseudocode were developed to facilitate understanding of algorithms and interconnections among parts of the program. For implementation, the principles of structured programming were used. Structured programming limited the use of jumps within the program code and significantly contributed to the ease of understanding code (or at least significantly decreased the complexity of understanding code).

It is not necessary to describe each of these techniques here. These formal management and documentation techniques are very helpful. Without them, the situation would be worse. However, they are not capable of eliminating the crisis. Software components are still crafted by hand, and they are connected through multiple interconnections and dependencies. The developers have difficulties documenting these interconnections so that those who work on other parts of the system would understand the mutual expectations and constraints. The maintenance programmers also have difficulties understanding complex (and poorly documented) interconnections.

As a result, the industry turned to techniques that alleviate the effects of interconnections. We are currently witnessing a shift from methodologies that allow us to write software faster and easier to methodologies that support writing understandable software. This is not a paradox—it is a shift in attitude toward program quality.

Remedy 3: Designing a Complex and Verbose Language

Earlier programming languages, such as FORTRAN, COBOL, APL, Basic, or even C were designed to facilitate the work of writing code. These programming languages were relatively small, terse, and easy to learn. Beating around the bush in writing code was frowned on, while succinct programming expressions were viewed as a manifestation of superb programming skills.

Lately, there's been a clear shift in programming language design. Modern languages, such as Ada, C++, and Java use the opposite approach. These lan-

guages are huge and difficult to learn. Programs written in these languages are invariably longer than similar programs written in more traditional languages. The programmers are burdened with definitions, declarations, and other descriptive elements of code.

This verboseness contributes to code consistency. If the programmer uses inconsistent code in different parts of the program, the compiler discovers that and forces the programmer to eliminate the inconsistency. With older languages, the compiler would assume that inconsistency was introduced intentionally to achieve some purpose known to the programmer. Language designers and compiler writers liked to say that "We do not want to second-guess the programmer." With such permissible languages, finding an error often required elaborate runtime testing and might result in errors escaping the hunt altogether. Modern languages treat code inconsistency as a syntax error and force the programmer to eliminate it even before the program has a chance to run. This is an important advantage, but it makes writing code much more difficult.

Another advantage of this verboseness is that code expresses the intent of the programmer better. With traditional languages, the maintenance programmer often had to guess what the code designer meant. Detailed comments in source code were needed to help the code reader, but the code designers often did not have the time (or skills) to comment code adequately. Modern languages allow the code designer to make the code more self-documented. "Redundant" declarations reduce the need for comments in code and make it easier for the maintenance programmer to understand the intent of the code. This is a new tendency in industry, and we will see specific coding examples that support this approach.

These modern languages are both huge and complex; they are, of course, too large and too complex for managers or scientists or engineers to use. These languages are designed for professional programmers who are trained in the art of partitioning the software system into cooperating parts without excessive interconnections among parts (and excessive shared knowledge among developers). The basic unit of modularization in older languages was a function. They provided no means to indicate in the program code that certain functions were logically closer to each other than to other functions. New languages also use functions as units of program modularization, but they also give the programmer the means to aggregate functions together. In Ada, this means of aggregation is called *package*. Ada package can contain data, and package functions can operate on that data, but there is only one instance of that data in the Ada program. C++ and Java make the next step: their unit of aggregation, class, allows the programmer to combine functions and data so that the program can use any number of data instances, objects.

However, the use of modern programming languages does not create any advantages in and of itself. Using these languages, you can write programs that are as bad as programs written in any traditional language, with multiple links between parts of the program, with obscure code that needs extensive documentation, so that the maintainer's focus of attention is spread over different levels of computations. This is where the object-oriented approach comes into play. In the next section, I will discuss why the object-oriented approach is so good.

The Object-Oriented Approach: Are We Getting Something for Nothing?

Everybody is excited about the object-oriented approach (well, almost everybody). Almost everybody knows that this approach is better than what preceded it (even though not everybody knows why). Those who are not excited about the object-oriented approach do not really doubt its effectiveness. They doubt whether the organizational changes are worth the trouble: expense of training the developers and the users; efforts to produce new standards, guidelines, and documentation; project delays due to learning; and assimilation of new technology with all related mistakes and false starts.

The risks are significant, but so are the rewards (or so we think). The major boost for the object-oriented approach comes from the availability and broad acceptance of languages that support objects; C++ is without a doubt the most significant factor here.

Is the object-oriented approach just a new buzzword? Will it be replaced by something else in due time? Does it have real advantages? Are there any drawbacks or tradeoffs to be made?

Let us be frank: There is no reason why the object-oriented approach (and the use of classes in C++) is advantageous for writing small programs. The advantages of the object-oriented approach outweigh drawbacks for large complex programs only.

There are two components to the program complexity:

- the complexity of the application itself (what the application does for its users)
- the complexity of the program implementation (introduced by the decisions made by designers and programmers while implementing the program)

We cannot control the complexity of application—it is defined by the goal of the program. We cannot expect that in future applications that complexity will somewhat decrease; if anything, it is going to increase with further increase in hardware power that puts more and more complex tasks within our reach.

It is the second component of complexity that we should control. Every time we decide that part of the work should be done in one unit of the program and another part of the work should be done in another unit of the program, we introduce additional complexity of cooperation, coordination, and communications among these units. This risk is particularly high when we tear apart the actions that should belong together and put them in different units of the program. Doing that creates significant additional complexity in the program.

Why would one want to tear apart what should belong together? Nobody does this intentionally or maliciously. However, often it is not easy to recognize which things belong together and which do not. To recognize these situations, we have to learn how to evaluate the quality of design. And, by the way, what is design?

What Does the Designer Do?

Many software professionals think that design is about deciding what the program should do, what functions it should perform for the user, what data it has to produce, what data it needs to do the job. Others add such tasks as deciding what algorithms should be used to do the job, what the user interface should look like, what performance and reliability should we expect, and so on. This is not design. This is analysis.

Design comes later, when we already know what functions the program should perform for the user, its input and output data, data transformation algorithms, user interface, and so on. Design, in general, is a set of the following decisions:

- What units a program will consist of. When you design a software program, this decision produces a set of functions, classes, files, or other units the program might consist of.
- How these units will be related to each other (who uses whom). This design decision describes which unit calls the services of which other unit and what data these units exchange to support the services.

- What the responsibility of each individual unit is going to be. It is while making this set of design decisions that one could go wrong and break apart what should belong together. But this observation is too general to be useful in practice. You need specific design techniques catered to specific units of modularity.

Come to think of it, this is true not only of software design, but of any creative activity. Be it a musical composition, a book, a letter to a friend, or a painting, we have to decide what components the product should have, how these components are related to each other, and what the role of each component in achieving the common goal is. The more complex the task, the more important design is for the quality of the result. A simple e-mail message does not need careful planning. A software user manual does.

Structured design uses functions as units of modularity. The designer identifies the tasks that the program should perform, partitions these tasks into subtasks, steps, substeps, and so on, until these steps are small enough; after that, each step is implemented as a separate, presumably independent function.

Data-centered design assigns modules so that each module is responsible for processing a specific element of input or output data. The designer identifies the data that the program uses as its input and the data that the program produces as its output. Then the designer breaks complex data into components until processes that are needed to produce output data from input data are small enough. After that, each data transformation process is implemented as a separate, presumably independent, function.

There are numerous variations of these techniques catered toward different types of applications: database applications, interactive applications, real-time, and so on.

All high-level programming languages provide functions, procedures, subroutines, or other similar units of modularity (e.g., COBOL paragraphs) and support these modular design techniques. These methodologies are useful, but they do not eliminate the problem of design complexity: The number of interconnections between program modules remains high, because the modules are linked through data. References to data make the module code obscure. The designers (and then maintainers) have too many factors to consider, and they make errors that are hard to find and correct.

Software designers use several criteria that allow them to minimize complexity and mutual dependencies between cooperating parts of the code. Traditional software quality criteria are cohesion and coupling. Modern object-oriented criteria are information hiding and encapsulation.

Design Quality: Cohesion

Cohesion describes relatedness of the steps that the designer puts into the same module. When a function performs one task over one computational object, or several related steps directed toward a specific goal, we say that this function has good cohesion. When a function performs several unrelated tasks over one object, or even several tasks over several objects, we say that the function has poor, weak, low cohesion.

High-cohesion functions are easy to name; we usually use composite terms that contain one active verb for the action and one noun for the object of the action, for example, `insertItem`, `findAccount` (if the name is honest, which is not always the case). For low-cohesion functions, we use several verbs or nouns, for example, `findOrInsertItem` (unless, of course, we want to hide the designer's failure and we just call the function `findItem` that finds the item in a collection or inserts the item if it is not found).

The remedy for poor cohesion (as for any design flaw) is redesign. Redesign means changing the list of parts (functions) and their responsibilities. In the case of poor cohesion, it means breaking the function with weak cohesion into several cohesive functions.

This approach works most of the time but should not be taken to the extreme. Otherwise, it will result in too many small functions, and the designer (and the maintainer) winds up with a larger number of things to remember (function names and their interfaces).

This is why cohesion is not used alone; it is not a very strong criterion. It needs other criteria to complement it. Make sure, however, that you consider it when you evaluate design alternatives.

Design Quality: Coupling

The second traditional criterion, coupling, describes the interface between a function (a server, a called function) and the calling functions (server's clients). The clients supply the server function with input data values. For example, the function `processTransaction` (a client) might call the function `findItem` (a server) and pass the item ID and the text of an error message as input data to `findItem`. The server depends on the correctness of its input data to produce correct results (e.g., find the item, display the right error message).

The clients depend on results produced by the server. For example, the function `findItem` might produce for its client (`processTransaction`) the

flag that says whether the item was found or not and the index of the item if it was found. This represents the server output. The total number of elements in server input and output represents the measure of coupling. We try to minimize coupling by reducing the number of elements in the function interface.

The criterion of *coupling* is more powerful than cohesion is. It is very sensitive to design decisions when the designer tears apart operations that should belong together. Almost invariably, these design decisions result in extra communication among modules and in additional coupling. For example, transaction error processing should be done in one place and should not be torn between `processTransaction` and `findItem`.

The solution to excessive coupling is, of course, redesign: reconsidering the decisions as to what function does what. If part of the operation on data is done in one function, and another part of the operation is performed in another function, the designer should consider allocating both parts to the same function. This decreases complexity of design without increasing the number of functions in the program. For example, moving error processing from `findItem` to `processTransaction` eliminates the need for passing the text of error message to `findItem` and passing the flag back to `processTransaction`.

At this stage I would like to make sure that when you look at C++ source code, you analyze it from the point of view of cohesion, coupling, and tearing apart what should belong together.

Design Quality: Binding Together Data and Functions

What contributions does the object-oriented approach make to these quality criteria? Remember, improving software quality does not mean making code aesthetically nicer, because aesthetics does not reduce complexity of code. Improving quality means making program modules more independent, making code more self-documented, and the intent of the designer easily understood.

The object-oriented approach is based on binding together data and operations. We will spend a good deal of time discussing the syntax for doing that. It

is important, however, before we look at the object syntax, to understand what this syntax tries to accomplish and why it is so expedient to use this syntax.

Why is binding together data and operations beneficial? The problem with the functional approach to the program design is that the "independent" functions are connected with other functions through data. For example, one function can set a value of a variable, and another function might use its value (`findItem` sets the value of the index, and `processTransaction` uses that index). This creates interdependencies between these two functions (and probably between other functions, too).

One solution to this problem is to merge these two functions. When it works, this solution has to be used. But it does not work all the time. Often, we call the server function repeatedly, and probably from different client functions. Eliminating the server function (`findItem`) does not make sense.

In addition, some other functions might set and use that value (the item index might be used in functions `deleteItem`, `updateItem`, and so on). In a small program, it is not difficult to trace all instances of accessing and modifying the value of a variable and find the source of the problem if something is done incorrectly. In a large program, this is more difficult, especially for the maintainer who does not completely understand the intent of the designer. Even the original designer, who returns to the program after several weeks or months of interruption, often feels at a loss in understanding the program and locating the functions that use a particular value.

It would be beneficial to indicate the set of functions that access and modify a particular variable by placing functions together in the source code. This would help the maintainer (and the designer when he or she returns to the program) to understand the intent of the designer at the time of writing the program. Many software designers do that because they understand the importance of the self-documenting characteristics of the code.

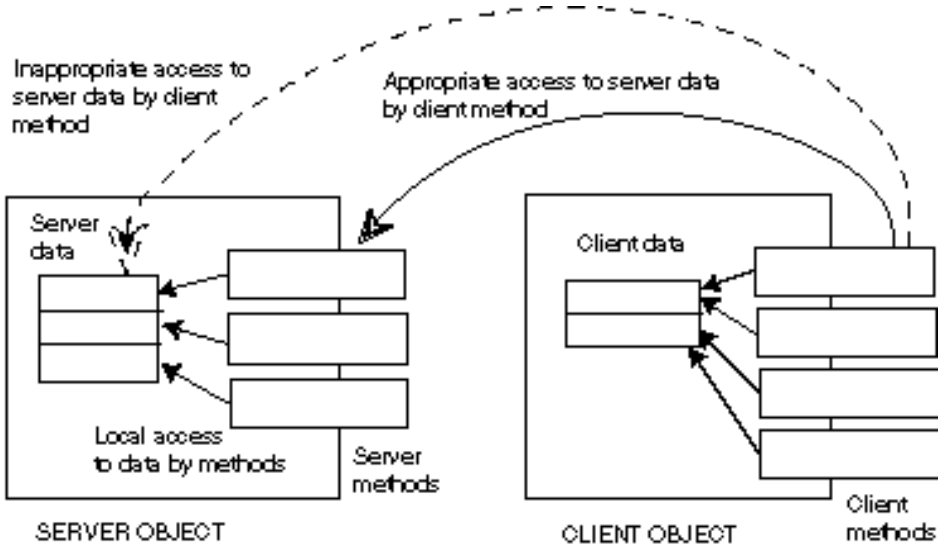
However, this is often hard to do. Functions are often placed in the source code so that they are easy to locate, in alphabetical order. Even when they are grouped according to the variables they access, there is no assurance that all relevant functions are indeed grouped together. When the designer (or the maintainer) needs a quick fix to some problem and adds a function that accesses this variable at some other place in the program, this is not a syntax error. The compiler will accept the program. The execution results will be correct. And the future maintainer might never notice that additional function. With functional programming, it is not easy to make sure that all functions that access or modify particular data are listed in the same place.

The object-oriented approach resolves this problem by binding together data values and functions that access and modify these values. C++ combines

data and operations in larger units called *classes*. We do not tear related things apart, we put them together, reducing the number of things to remember about other parts of the program. Data can be made private, assuring that only the functions that belong to the class can access these particular values. Hence, the knowledge of the designer (that is, the list of functions that access data) is expressed explicitly in the syntactic unit: class description. The maintenance programmer is assured that no other function accesses or modifies these values.

Figure 1–5 shows a relationship between two objects, a server and a client object. Each object consists of data, methods, and the border. Everything that is inside the border is private and is not available to other parts of the program. Everything that is outside of the border is public and is available to other parts of the program. Data are inside the border and are invisible from outside. Methods (functions) are partially inside and partially outside of the border. The outside part represents the method interface known to the clients. The inside part represents the code implementation hidden from the client code.

Figure 1–5 The relationship between server and client objects.



design. When the client method needs the server data to do its job, it does not specify the names of server data, because that data is private and is not available outside of the server class. Instead, the client method calls the server method that accesses the server data. Since the server method implementation is inside the server, it has no difficulty accessing private server data.

Design Quality: Information Hiding and Encapsulation

When we decide what data and functions should belong together, we should choose among a large number of alternatives. Some alternatives are clearly not good, while some others are better. Choosing which one to implement can be a chore.

The criterion of encapsulation requires us to combine data and operations so that the client code would be able to do its job by calling server functions without explicitly mentioning the names of server data components.

The major advantage of this approach is that the list of functions that access server data is immediately available to the maintainer in the form of class description.

Another advantage is that the client code contains less data-related manipulation and hence becomes self-documented. Let us say that the application has to set the variables describing a customer—first and last name, middle initial, street address, city, state, zipcode, social security number, and so on—16 values in all. If the client code is written using the traditional approach, the designer writes the client code with 16 assignment statements. Now the maintainer has to decide:

- whether or not all components of the customer description are assigned values
- whether or not the values assigned here, only to components of the customer description or some other data, are also handled

It might take some time and effort to answer both questions, and this contributes to complexity.

If this code is written using the object-oriented approach, the data is private, and the client code cannot mention the names of variables describing the customer—name, address, and so on. Instead, the client code calls an access function such as `setCustomerData`. This passes the designer's intent to the maintainer immediately.

The criterion of information hiding requires us to combine data and operations and distribute responsibilities among operations so that the client code will be independent from the data design.

For example, we might not need to check the validity of the state and zip code values and check whether they are consistent. It is appropriate to push this job down to the server rather than to the client. If the assignment of zip codes to states changes, or if a city secedes from the state and becomes a separate state, we will have to change only the server and not the client code. Had we assigned this responsibility to clients, we would have to change each place zip code is used.

The important advantage of the object-oriented approach over the traditional approach is the narrow scope of modification changes. Let us say that we switch from a five-digit zip code to a nine-digit zip code. With the traditional approach, all client code has to be inspected, because the customer data can be handled anywhere. If in some places you miss adding the code that handles additional digits, it is not a syntax error, and it has to be discovered during regression testing.

With the object-oriented approach, you have to change the functions that handle customer zip code, and the client code that calls these functions is not affected. So, the major advantages of encapsulation and information hiding are:

- There is a clear indication what functions access specified data by binding together data and functions in class descriptions.
- The meaning of client code is clear from function names, not from the interpretation of a large number of low-level computations and assignments.
- When the data representation changes, the class access functions change, but the range of changes in the client code is narrow.

Design Issue: Name Conflicts

Less crucial but still very important is the issue of name conflicts in a large program. The names of the functions must be unique in a C++ program. If one designer chose the function name `findItem`, no other designer can use this name for another function in the same program. At first glance it looks like a simple problem. So, anybody who wants to use this name will use a slightly modified name, for example, `findInventoryItem` or `findAccountingItem`.

For many developers, the need to come up with unique names is an annoying restriction. But this is not just a problem of creativity in naming. This is the problem of excessive communication between developers.

Let us say you are a member of a 20-person development team, and you want to name the function you are working on `findItem`. Let us say there are three other developers on the team that write client code that will call this function. With the object-oriented approach, only these three developers should know that you are writing a `findItem` function. All others do not have to learn about that and can concentrate on other things.

With the traditional approach, all 20 developers have to be notified about your decision. Of course, they will not devote all their working time to learning the name of your function. But still, all of them will have to know the names of your functions and the functions designed by all other developers. Notice that you also have to learn (or keep handy) the list of all function names that all other developers are designing, even though you do not use them. Many organizations develop sophisticated standards for function names to avoid name conflicts and spend significant resources training developers to use these standards, enforcing compliance, and managing change when the standards have to evolve.

This can easily become a chore that channels human attention from other issues. After all, the power of our attention is limited, and the thinner it is spread, the higher is the likelihood of errors. The object-oriented approach alleviates this problem. It allows you to use the same function name for as many functions as you want if these functions belong to different classes. Hence, only those developers who actually call your functions should know about your names. All others can devote their energy to other issues.

Design Issue: Object Initialization

Another less crucial but still important issue is object initialization. In the traditional approach, computational objects are initialized explicitly in the client

code. For example, the client code must explicitly initialize the components of the customer description. With the object-oriented approach, a call to `setCustomerData` takes care of that. Still, the client code has to call this function. If this function is not called, this is not a syntax error but a semantic error, and it has to be discovered at run time during testing. If customer processing requires system resources (files, dynamic memory, etc.), these resources also have to be returned by the client code explicitly, for example, by calling the appropriate function.

The object-oriented approach allows us to do these operations implicitly. When the client code creates an object by passing the necessary initializing data (we will see the syntax for doing that later), there is no need to explicitly call initializing functions. At the end of computations, resources allocated to the object can be returned without explicit action of the client code.

of the object-oriented approach. When reading the rest of the book, please come back to this chapter often to make sure that the trees of syntax do not hide from your view a beautiful forest of object-oriented programming.

What Is an Object, Anyway?

In object-oriented programming, we design a program as a set of cooperating objects rather than as a set of cooperating functions. An *object* is a combination of data and behavior. As a programmer, you might be familiar with other terms for data, such as data fields, data members, or attributes. We often refer to object behavior using such terms as functions, member functions, methods, or operations.

Data characterizes the state of an object. When similar objects can be described in the terms of the same data and operations, we generalize the idea of the object in the concept of the class. A class is not an object. A class is a description of common properties (data and operations) of objects that belong to that class. A class does not live in the memory of the computer during program execution. An object does. Every object of a specific class has all the data fields defined for the class. For example, each `InventoryItem` might have an i.d. number, item description, quantity on hand, purchase price, retail price,

and so on. We describe these common properties in the definition of class `InventoryItem`. During the program execution, objects of the class `InventoryItem` are created and allocated memory for their data fields. These objects can change independently of each other. When the values of data fields change we say that the state of the object changes.

All objects of the same class are characterized by the same behavior. Object operations (functions, methods, or operations) are described in the class definition together with data. Each object of a specific class can perform the same set of operations. These operations are performed on behalf of other objects in the program. Usually, these are operations over object data; these operations can retrieve the values of data fields, or they can assign new values to data fields, or compare values, print values and so on. For example, an inventory item object can have, among others, such functions that set retail price to a given value or compare the item's i.d. number with the given number.

A computer program can have more than one object of the same kind. The term object is used to describe each such object instance because an object is an instance of a class. Some people also use the term object to describe the group of objects of the same kind. More often, however, it is the term class that is used to describe the set of potential object instances of the same kind. Each object of the same class has its own set of data fields, but the corresponding fields in different objects have the same names. For example, two inventory item objects might have the same (or different) values of retail price; the i.d. number will probably be different for different inventory item objects. All objects of the same class can perform the same operations, that is, they respond to the same function calls. All objects of the same kind have the same properties (data and operations). When we call an object function that changes the state of the object or retrieves information about the state of the object, we say that we are sending a message to the object.

This is a very important detail. In a C++ program, a function call usually involves two objects. One object sends a message (calls the function), another object receives the message (sometimes we say it is a target of the message). We will call the object that sends the message the *client object*; we will call the target of the message the *server object*; although this terminology sounds similar to the client-server terminology used for the client-server computer system architecture, it means different things. Actually, these terms were used in object-oriented programming much earlier than the first client-server systems became popular. We are going to spend a lot of time talking about client-server relationships among objects in a C++ program.

As you are going to see later, the objects in a C++ program look syntactically very much like ordinary variables—integers, characters, and floating

point numbers. They are allocated memory pretty much the way ordinary variables are: allocated on the stack or on the heap (again, we will see the details later). A C++ class is a syntactic extension of what other languages call structures or records that combine data components together. The C++ class includes both data declarations and function declarations.

So, when the client code needs to use the object, like comparing an item i.d. number with a given value or setting the value of the item retail price, it does not mention the names of the object data fields. Instead, it calls the functions provided by the object and these functions do the job for the client; they compare item i.d. numbers and set the value of the retail price.

Advantages of Using Objects

Although it does not sound like a big deal—the client code mentions the names of object functions rather than the names of object data fields—the difference is significant. Experience shows that the design of data structures is more volatile and amenable to change than is the design of operations. By using the names of functions rather than the names of data fields, we insulate the client code from potential changes in the server object design. By doing so, we improve the maintainability of the program, and this is one of the important goals of the object-oriented approach.

Also, when the client code calls a server function, for example, `compareID`, the intent of the client designer is immediately clear to the maintainer. When the client code retrieves and manipulates IDs of the objects, the meaning of the code has to be deduced from the meaning of elementary operations rather than from function names.

In summary, the goal of the object-oriented approach is the same as for other software development methodologies: to improve software quality as perceived by the ultimate user (program functionality and total development and maintenance costs).

Proponents of the object-oriented technology hope that the object-oriented approach allows us to reduce complexity of program code. With less complexity to deal with, we hope to decrease the number of errors in software and to increase productivity during development and maintenance.

Software complexity can be reduced by partitioning programs into relatively independent parts that can be understood in isolation, with few references to other parts of the program. When we use classes as program units, we have a chance to decrease interconnections between the parts. We pay for that by increasing interconnections between parts of the class; class member

functions operate on the same data. But this is fine—a class is usually developed by the same person, and it is interpersonal communication that is prone to omissions, inconsistencies, and misunderstanding. Reduced dependencies between classes reduce coordination among team members assigned to these classes, and the number of errors.

Unlike interconnected parts, independent classes can be easier to reuse in other contexts; this improves productivity during development of the system and possibly productivity during development of other software systems.

Interconnected parts have to be studied together, which is slow and prone to error. Independent classes can be easier to understand: This improves productivity during program maintenance.

The object-oriented technology is not without its risks and costs. Developers, users, and managers—all have to be trained, and training costs are significant. Object-oriented projects seem to take longer than traditional projects do, especially during the first phases of the project, analysis and design. Object-oriented programs contain more lines of code than do traditional programs. (Do not be scared—I am talking about lines of source code, not about the size of object code—the size of executable code really does not depend on the development methodology.) Most important, the languages that support object-oriented programming (especially C++) are complex. Using them entails certain risk that the benefits will not be realized, that the object-oriented program will be larger, more complex, slower, and more difficult to maintain than the traditional program. Hopefully, this book will teach you not only how to use C++ but also what to avoid. The proper use of this powerful and stimulating language will help you realize the promise of the object-oriented technology.

Characteristics of the C++ Programming Language

C++ is a superset of the C programming language. C itself is a descendant of several generations of early languages; it was created and implemented with conflicting goals in mind. This is why C++ contains features that are inconsistent and sometimes irritating. In this section, I will briefly review the major characteristics of C and then will show how C++ uses this “inheritance” to achieve its goals.

C Objectives: Performance, Readability, Beauty, and Portability

The first goal of C was to give software developers a performance-oriented systems programming language. This is why C and C++ do not support run-time checking for errors that could cause incorrect program behavior but could be found by the programmer during testing. This is why C and C++ contain low-level operators that emulate assembly language instructions and allow the programmer to control computer's resources, such as registers, ports, and flag masks.

has spared you hundreds of hours of debugging anguish of assembly language programming.

The second goal of C was to give software developers a high-level language suitable for implementing complex algorithms and sophisticated data structures. This is why C and C++ allow the programmer to use loops, conditional statements, functions, and procedures. This is why C and C++ support processing of different data types, arrays, structures, and dynamic memory management. (If you are not comfortable with these terms, do not worry; this will not prevent you from mastering C++ using this book.) These features support code readability and maintainability.

The third goal of C was to allow software developers to write source code that is elegant and aesthetically pleasing. It is not exactly clear what "elegant" and "aesthetically pleasing" meant; it probably meant different things to different people, but the consensus was that if the program is succinct, terse, and puts a lot of processing into a few lines of well-designed code than it is elegant and aesthetically pleasing. As a consequence of this approach, the language affords the programmers significant "freedom" in writing code without flagging this code as syntactically incorrect. We are going to see more of this issue later.

The fourth goal of C was to support program portability at the level of the source code; this is why C and C++ executable object code is not expected to run under different operating systems or on different hardware platforms. However, the source code is expected to be compiled by different compilers

or for different platforms without changes, and it should run exactly the same way without modifications.

The first three goals were achieved reasonably well, even though they were somewhat conflicting. C was used for the development of the UNIX operating system, which gradually became very popular and was implemented on a large number of hardware platforms, including multiuser environments (mainframes, minicomputers, PC servers) and single-user environments (PCs). C was also used for implementing system utilities, database systems, word processors, spreadsheets, editors, and numerous applications.

The conflict between readability and succinct expressiveness (programming aesthetics) was not resolved. Those developers who valued readability learned how to use C to produce readable code. Those developers who valued expressiveness learned how to use C to produce succinct code and even had competitions for writing the most obscure and expressive code.

The fourth goal, the portability of C code was also met, but with significant reservations. That is, the language itself is portable: If the language statements are compiled under different operating systems or on different hardware platforms, the program will execute in exactly the same way. The catch is that any real C program contains much more than C language statements: it contains numerous calls to library functions.

The implicit goal of the C language design was to create a small language. Initially, it had only 30 keywords. If you compare it with COBOL or PL/I, the difference is staggering. As a result, the language is very small. It does not have the exponentiation operation, it cannot compare or copy text, and it does not include input or output operations. However, you can do all of that (and much more) using library functions that come with the compiler. The language designers felt that it was up to compiler vendors to decide what library functions should be used to compare or copy text, to perform input or output operations, and so on.

This was not a good idea. It was at odds with the idea of source code portability. If different compilers and different platforms were using different library functions, then the program could not be ported to a different platform without modifications to library function calls. It could not even be recompiled on the same platform using a different compiler. Also, this approach was at odds with the idea of "programmer portability." A programmer who learned one library had to be retrained to be able to use another library.

This is no small matter, and vendors of compilers for different platforms recognized its importance and developed the "standard" library that programmers could use on different machines without making too many modifications to program code. "Too many" means that some modifications had to

be made. Because of the lack of one strong center for standardization, several versions of UNIX were developed, and libraries of different compiler vendors behaved somewhat differently on different machines under different operating systems.

American National Standards Institute (ANSI) spearheaded the standardization effort and codified ANSI C in 1983–1989 with the goal of promoting portability. The ANSI version of the language also incorporates some new ideas, but does it with backward compatibility, so that the legacy C code could be recompiled by new compilers.

Today, even though C source code is mostly portable, problems do exist, and porting a program to a different machine or operating system might require changes. Skills of C programmers are also mostly portable; and programmers can move from one development environment to another with little retraining (but some training might be needed).

The C designers also did not see the "catch" in introducing diverse libraries. We pay for that "flexibility" with increased costs of code porting and programmer retraining. The experience that the industry accumulated dealing with these issues is one of the reasons why Java designers pay so much attention to enforcing uniform standards. The Java language is extremely unforgiving and flags many C idioms as syntax errors. The issue of backward compatibility with C has a relatively low priority in Java design. Clearly, Java designers did not want to sign on that dotted line.

C++ Objectives: Classes with Backward Compatibility with C

One C++ design goal was to expand the power of C by supporting the object-oriented approach to programming. Here, "expand" should be taken literally. C++ is designed for 100% backward compatibility with C: Every legal C program is a legal C++ program. (Actually, there are some exceptions, but they are not important.) Hence, C++ shares with C all its design features, good or bad (until death do us part).

Similar to C, C++ is token oriented and case sensitive. The compiler breaks the source code into component words regardless of their position on the line, and elements of the program code should not be in specific columns (e.g., they have to be in FORTRAN or COBOL). The C++ compiler ignores all white space between tokens, and the programmers can use white space to format code for readability. Case sensitivity helps avoid name conflicts but can result in errors if the programmer (or the maintainer) does not pay (or does not have time to pay) attention to subtle differences in capitalization.

Similar to C, C++ only has a few basic numeric data types, fewer than in other modern languages. To add insult to injury, some of these basic data types have different ranges on different machines. To make matters even more confusing, the programmers can use so-called modifiers that change the legal range of values acceptable on a given machine. This has implications both for software portability and maintainability.

To compensate for the scarcity of built-in data types, C++ supports data aggregation into composite types: arrays, structures, unions, and enumerations; data aggregates can be combined to form other aggregates. This feature is also borrowed from C.

C++ supports a standard set of flow control constructs: sequential execution of statements and function calls, iterative execution of statements and blocks of statements (`for`, `while`, `do` loops), decision-making (`if`, `switch` constructs), jumps (`break`, `continue`, and, yes, there is the `goto` statement too). This set of control constructs is the same as in C, but there are some differences in the use of `for` loops.

Similar to C, C++ is a block-structured language: Unnamed blocks of code can be nested to any depth, and variables defined in inner blocks are not visible to the outer blocks. This allows programmers who write the inner blocks to use any names for local variables without the fear of conflict (and need for coordination) with the names defined by the programmers who write the outer blocks.

On the other hand, a C (and C++) function (i.e., a named block) cannot be nested inside another function, and the function names must be unique in the program. This is a serious limitation. It increases pressure on coordination among programmers during development and makes maintenance more difficult. C++ partially corrects this problem by introducing the class scope. Class methods (that is, functions defined inside the class) have to be unique within the class only, not within the program.

C++ functions can be called recursively in exactly the same way as C functions can. Older languages did not support recursion because recursive algorithms represent a miniscule fraction of all algorithms. Naïve use of recursion can waste both time and space during execution. However, a few algorithms where recursion is useful really benefit from it, and recursion is a standard feature in modern programming languages (but not in scripts).

Exactly as in C, C++ functions can be placed in one file or in several source files. These files can be compiled and debugged separately, thus enabling different programmers to work on different parts of the project independently. Compiled object files can be linked together later to produce an executable object file. This is important for labor division in large projects.

Very much like C, C++ is a strongly typed language: It is an error to use a value of one type where a value of another type is expected, for example, in expressions or in passing arguments to a function. Today, this is a common principle of programming language design. Many data type errors that would manifest themselves at run time can be flagged at compile time. It saves the time spent on testing and debugging.

Even more than C, C++ is a weakly typed language (yes, it is both a strongly typed and weakly typed language). Conversions among numeric types are done silently both in expressions and in parameter passing. This is an important departure from modern language design and prone with errors that cannot be discovered at compile time. In addition, C++ supports conversions between related classes. On the one hand, this allows us to use a nice programming technique called polymorphism. On the other hand, this feature prevents the compiler from catching substitutions made inadvertently.

C++ inherits from C support for the use of pointers for three purposes: a) passing parameters from a calling function to a called function, b) dynamic memory allocation from the computer heap (for dynamic data structures), and c) manipulating arrays and array components. All techniques for using pointers are prone to error; these errors are particularly difficult to discover, localize, and correct.

Very much like C, C++ is designed for efficiency: Array bounds are checked neither at compile time nor at run time. It is up to the programmer

to maintain program integrity and avoid corruption of memory if an invalid index is used. This is a popular source of errors in C/C++ programs.

Similar to C, C++ is designed for writing succinct and terse code: It gives a special meaning to punctuation and operator characters such as asterisks, plus signs, equal signs, braces, brackets, commas, and so forth. These symbols can have more than one meaning in a C++ program: The meaning depends on context, and this makes learning and using C++ more difficult than learning and using other languages.

C++ adds a number of new features to C. The most important feature is support for objects. C++ expands C structures to classes; they bind together data and functions as a unit of code. Classes encourage information hiding by localizing data representation within their borders so that the data components are not available outside of the class. Classes support encapsulation by providing access functions (methods) that are called by client code. The use of class scope reduces name conflicts in C++ programs.

Classes facilitate hierarchical approach to design so that higher-level classes reuse lower-level classes. Class composition and class inheritance allow the programmers to implement complex models of the real world and manipulate program components easily.

There are a number of other features in the language that help the designer express code intent in the code itself, not in comments.

However, the C++ language, similar to C, was designed for an experienced programmer. The compiler does not try to second-guess the programmer, assuming that he or she knows what he or she is doing. (And we do not always, do we?) It is important to know what we are doing. If the developer is not careful, a C++ program can be quite complex and intimidating to a reader and difficult to modify and maintain. Type conversions, pointer manipulation, array processing, and parameter passing are also frequent sources of obscure errors in C++ programs.

Hopefully, sound software engineering practices recommended in this book will help you to understand what you are doing and to avoid pitfalls of unnecessary complexity.

Summary

In this chapter, we looked at different solutions to the software crisis. The use of object-oriented languages seems to be the most effective way of avoiding budget overruns, missed deadlines, and scaled-down releases. However, writ-

ing programs in object-oriented languages is more difficult than writing them in traditional procedural languages. Used correctly, object-oriented languages facilitate the reading of a program, not the writing of it. Actually, this is very good. After all, we write source code only once, when we type it in. We read code many, many times over—when we debug it, test it, and maintain it.

As an object-oriented language, C++ allows you to bind together data and functions in a new syntactic unit, class, which extends the concept of type. Using C++ classes, you write the program as a set of cooperating objects rather than as a set of cooperating functions. Using classes promotes modularization, and code design with high cohesion and low coupling. Classes support encapsulation, class composition, and inheritance. This contributes to code reuse and maintainability. The use of classes eliminates naming conflicts and contributes to understanding code.

It is important to learn how to use C++ correctly. The indiscriminate use of the features that C++ inherited from C could easily eliminate all advantages of object-oriented programming. Our discussion of these features was by necessity cursory and introductory. Later in the book, I will try to do my best to show you how to make the best use of C++ features. This presentation will be full of specific technical details. As I mentioned earlier, I think it is a good idea to come back to this chapter from time to time to make sure that you do not lose sight of powerful object-oriented ideas behind low-level syntactical details.

In the next chapter, we will start our road toward the correct use of C++ by reviewing its basic program structure and most important programming constructs.