# Teaching C++: Unique Challenges

Conrad Weisert, December, 1996



## 1. The Challenge

Teaching C++ is like teaching no other programming language or development tool. Among the complications in designing and presenting a C++ course are:

- Conveying the object-oriented paradigm and a new language at the same time.
- Integrating guidance on good practice with an intimidating amount of factual detail.
- Coping with a language so huge (with its essential standard library components) that no one expects to master all of it.
- Recognizing a 2-tiered programming milieu, in which a relatively small cadre of senior designer programmers prepares object-oriented *classes* to be used by a larger population of *client* programmers.
- Deciding which specialized areas are best deferred to short follow-up courses to be taken after the programmer has gained fluency in core C++ concepts and techniques.

A few C++/OOP courses, academic and commercial, are coping well. Many, however, are struggling to apply approaches that may have worked well for older languages but break down under the weight of these complications. Their alumni, for the most part, have wasted their time and money and are poorly prepared for the world of object-oriented software development.

## 2. Prerequisites

### *Targeting experienced programmers*

C++ is not a suitable first language for teaching computer programming to beginners. The neophyte must first grasp the stored-program concept exemplified by the procedural paradigm. That's followed by indoctrination into good programming style, with emphasis on minimizing repetition and on organizing programs into well-isolated modules with clean interfaces. Experience in testing non-trivial programs is also essential.

Pascal, PL/I, newer versions of Basic, and, of course, C are suited to providing such background in either a first programming course or in on-the-job experience.

### *What about Cobol?*

Programmers from a mainly Cobol background present a special challenge because of the quasi-structured traditions many programming groups observe. Although modern Cobol supports modular program structure, many Cobol textbooks and courses encourage a monolithic DATA DIVISION with every data item at global scope. Many other folklore traditions from early days remain in some organizations' programming standards and seriously limit program quality.

To cover the required *unlearning*, I add 6 hours to my C++ course when I give it to Cobol-oriented groups. In the extra sessions we examine module coupling and we debunk some Cobol traditions. After a couple of difficult experiences, I now avoid presenting C++ to mixed Cobol/non-Cobol groups.

### *Should we require C as prerequisite?*

After trying it both ways, I've concluded that we're better off assuming little if any background in C. I give, therefore, an integrated introduction to "C and C++".

Surprisingly many programmers who claim to know C are nevertheless shaky on crucial basics. Only if a student is a world-class C programmer, do I excuse him or her from the 5 hour quick survey of C that starts the course. By focusing on C's peculiar idioms and style rather than where the semicolons go, we hold the attention of mixed groups and rarely hear complaints of boredom from those who already knew some C.[1]

It would take a far bigger investment to send prospective C++ students to a separate prerequisite C course.

[1] I get an occasional complaint that the C survey portion goes too *fast* for someone who doesn't already know C, but those complaints turn out to be due to inadequate programming background in *any* procedural language. By finding that out early in the course we spare the unprepared student a lot of frustration.

In our quick survey, we omit or de-emphasize those ugly parts of C and its standard library that a C++ programmer can and should avoid.

### *Audience for the basic C++ course*

Our participants, then, are experienced professionals or advanced students who have been successfully programming in a language that supports structured coding constructs, a high degree of modularity, and data structuring. They'll already have a firm grasp of such basics as elementary data types, expressions, parameter passing, separate compilations, and structured flow control. They'll be able to digest some C and C++ equivalents quickly and to absorb even the detailed idiosyncrasies of C and C++ with little confusion.

## 3. Duration and schedule

### *No saturation scheduling*

I'm amazed and dismayed by the barrage of brochures for C++ tutorials or workshops that rely on saturation scheduling: all day long, day after day, for 3, 5, even 10 days. No matter how skillfully we present topics, no matter how well we structure workshop problems, the material is much too long and complex for students to grasp and retain in such a compressed schedule.

Many of these brochures flaunt the term *intensive*, as if that were a virtue. It isn't. C++/OOP is nothing like a foreign-language immersion course. Students gain little benefit from repetitive drill. Our goal isn't to have the students memorize syntax rules -- that's what manuals, textbooks, and help screens are for -- but rather to have them understand how to apply C++'s facilities well and how to avoid the pitfalls that C++ throws in their path. That requires time for study and reinforcement.

### *Session length and number of sessions*

A half-day ($3\frac{1}{2}$ hours with a short break in the middle) is a comfortable session. The students can absorb a reasonable amount of new material without straining their attention spans.

My basic C++ course[2] requires 6 half-day sessions to cover the basics. That's about the longest commitment most organizations can realistically make for an on-site course. In a longer course, we'd get dropouts due to competing pressures of meetings and other work.

### *Session spacing*

Ideally these half-day sessions should be spaced a week apart. That gives students time to keep up with the demands of their regular jobs, to read relevant textbook chapters, to work assigned problems at their own pace, and most important, to catch up if they start falling behind or have to miss a session.

A once-a-week schedule is ideal both for academic courses and for on-site presentations to an organization's local staff, but it may be impractical for public seminars or whenever the students have to travel from multiple locations. I've successfully done *twice*-a-week sessions, and that can work, too: preferably either Monday-Thursday or Tuesday-Friday.

But twice a week is the limit. I was recently asked to undertake a saturation schedule for a group that, so they claimed, was so desperate to become productive in C++ that they couldn't tolerate the slow pace of a twice-a-week schedule for a 6-session course. I compromised on "semi-saturation": We did a half-day session every day for eight[3] working days.

It didn't work. Although management had assured us that the participants would be dedicated to the course work, the demands of their regular jobs interfered not only with their study time outside the classroom but also with their attendance at scheduled class sessions. Out of 24 participants only about 10 were able to keep up with the course; the rest fell behind early and had no good way to catch up.

In that case, of course, there was no sensible reason why we couldn't have adopted a more reasonable schedule, since the participants were all at the same site. But what if geographic constraints prevail, either because students come from multiple sites or because the instructor has to travel a long distance?

### *Network of minicourses*

Although no compromise is entirely satisfactory, certain compromises are still far preferable to a saturation or semi-saturation schedule. Splitting the basic C++/OOP course into three or four mini-courses is one possibility. Each mini-course then consists of two or three sessions given close together, and the courses can be separated by a month or so to facilitate travel.

---

[2] "Making the Switch to C and C++ -- an accelerated course for experienced programmers". I cover the same material in more detail in the full-term academic version, which may be only the student's second programming course.

[3] Two sessions longer than the usual six, in anticipation of lack of study time, still not enough to compensate.

Of course, we can't break up a course arbitrarily; each mini-course must have definite objectives that leave the students prepared to undertake some real work. The disadvantages are the extended span of time needed to cover C++/OOP and the difficulty in sustaining continuity of participation.

## 4. Attendance and continuity

### *Cumulative content*

Most multiple-session courses are cumulative in content, and a C++ course is more cumulative than most. If a student misses a session or even a part of a session, subsequent material may make no sense at all.

### *Catching up*

This problem is often serious in on-site course, because of competition from meetings and allegedly urgent messages. After a bona fide and unforeseeable emergency, of course, we try to assist the participant in catching up. It's rarely possible, however, for a student to catch up on his own just by reading the textbook and class handouts. (If it were, then we wouldn't need classroom sessions.) In the classroom we discuss the pros and cons of alternative techniques and problem solutions. We may criticize misguided examples in the textbook. We explain the reasoning behind examples. We point out the crucial relationships among different concepts and techniques. Little of our classroom discussion is duplicated in the written material.

Some organizations record class sessions on videotape, and invite anyone who misses a session to view the tapes. That's better than nothing, but it rarely works well, partly because the students may lack time to view the tapes before the next class session and partly because of the unsuitability of the medium in low-budget productions. A three-hour class viewed from a single-camera vantage point with poor sound and illegible writing will hardly hold anyone's interest.

### *Taking attendance*

I've recently altered my view on taking attendance in non-academic courses and on what to do in the event of excessive absence. I used to assume that *mature* students were capable of judging for themselves whether they could afford to miss all or part of a session and would take responsibility for getting caught up. I was wrong.

The problem is that management has *assigned* people to take the course. Participants, even senior professionals, may feel that they don't have the option of dropping out of the course after falling behind. Students who miss one or two sessions[4] return, because they have no choice, and sit through the later sessions utterly uncomprehending. They may intend to study the materials on their own, to talk to fellow students, or to confer with the instructor, but they just keep falling farther behind.

I've begun keeping track of students' comings and goings. After a student misses a crucial topic I may:

- bar him from returning to later sessions unless he can show that he's prepared for what comes next,
- exclude him from participating in course evaluation exercises at the end.

Management doesn't always let me take such drastic action. If not, I can at least consult my own list of who was present for each topic, if someone later expresses dissatisfaction with the course.

## 5. Hands-on problem solving

### *An integral part of the course*

Students will retain the concepts and techniques they've learned only if they immediately reinforce them through actual use. C++ is full of details that appear plausible in a textbook or in a classroom discussion, but that turn out to be elusive a week later at the computer.

At the end of each session I assign a problem that draws on the concepts and techniques we've just discussed. If the students turn in their solutions at the next session, I examine them and return them with comments at the following session.

In an academic course, homework is mandatory and counts toward a grade. In an on-site course or a public seminar, on the other hand, the instructor can't compel students to do the homework. In presenting C++ in different kinds of organizations I've found that:

- Some participants, but not all, are eager to do the assignments and to demonstrate their new skills.
- With weak management support we usually get about 20% of the assignments turned in, while with strong support we get 75% or better.

---

[4] The same management who assign students to attend the course often demand their attendance at competing meetings! If not, they at least foster an environment where competing activities take priority over class attendance.

- Those who elect not to do the assignments (or whose work assignments don't allow enough time) end up forgetting most of the content. They get only a broad *C++ appreciation* out of their investment in attending.

Although I always make the purpose and value of the assignments clear at the start and in the printed syllabus, we rarely get 100%.

### *Inside or outside the classroom?*

A desktop computer in front of each student has now become common in some classrooms. Having students follow along is helpful in teaching such interactive skills as spreadsheet navigation, but it actually inhibits learning in an advanced programming course.

Very little in C++/OOP lends itself to short bursts of computer use. Getting started with the editor-compiler environment or seeing the effect of a particular program change may be useful, but our emphasis is on *problem solving*, not facility in interacting with the software. Problem solving requires concentration in a calm, reasonably private setting.

Nor do we favor scheduled in-class workshop problems. They put undue pressure on slower students, and often encourage sloppy practices. Then after the workshop segment ends, the computers remain a distraction from the presentations and class discussions.

### *Prepared examples and customizing exercises*

Since it's impractical for the students to develop self-contained examples of larger C++ classes, I distribute examples both for in-class code walkthroughs and as (optional) building blocks for the assigned exercises. This also gives the students early appreciation for component reuse.

We must not, however, tempt the student into trial-and-error *customization* in lieu of original work. Any handout components prepared in advance must stop well short of solving the assigned problems.

## 6. Optional and follow-on topics

### *The size and complexity issue*

C++ is, by a big margin, the largest and most complicated language[5] in the history of programming. A C++ practitioner is expected to command an enormous and growing repertoire of concepts and techniques. Few C++ programmers will ever master the whole thing, nor will they need to.

That presents quite a challenge to the course designer. Where do we draw the line between our basic C++/OOP course and optional topics? Can we structure a network or *curriculum* of shorter specialized courses to meet the needs of programmers working on different kinds of application?

We really have little choice but to limit the *basic* C++/OOP course to:

- the fundamentals of the object-oriented paradigm,
- the C++ techniques that support that paradigm, and
- the C/C++ features needed to work on typical non-specialized applications.

We can illustrate these concepts and techniques with lots of practical examples, but we must resist any temptation to pile on additional details, however interesting.

### *Frameworks and GUI classes*

Some course purveyors try to integrate basic C++/OOP instruction with MFC[6] or some other set of graphic user interface tools or event-driven *application frameworks*, usually vendor-specific.

That's way too much for one course. What typically happens is that the student acquires some facility in plugging pieces of C++ code into predefined structures, but without gaining a firm grasp of the underlying concepts. He or she may be temporarily effective in throwing together programs that fit the patterns, but may well be unable to do original C++ design work or to confront new patterns.

An MFC (or other vendor's equivalent) course is fine, but only after the students have gotten a solid grounding in C++ and OOP concepts. Those who've mastered the basics of C++ may not even need a follow-on course, but can pick up what they have to know on their own from a book or vendor's manual.

### *Databases and I-O*

C++'s lack of support for *persistent objects*, i.e. permanent files, remains an obstacle to its use in business applications. Organizations who need to manipulate databases or permanent files (who doesn't?) can choose among three ways of circumventing this limitation:

---

[5] We count as *language* those standard library components deemed essential for typical application development.

[6] Microsoft's proprietary class library, mainly for *Windows* programming.

- They can develop special-purpose I-O interface components using some variant of RTTI[7] to reconstitute objects upon input.
- They can use an *object-oriented database* product.
- They can treat *relational tables* as objects,[8] rather than the data those tables describe.

Given the lack of a standard approach, there's no point in trying to cover these techniques in a general course. One or more short follow-ups may be appropriate.

### *Container classes*

This topic includes both what we used to call "array handling" and, for more sophisticated needs, "dynamic data structures".

The Standard Template Library (STL) and its vendor-specific predecessors make the use of containers more accessible to rank and file programmers, but at the cost of a huge addition to the programmer's repertoire. The entire STL is not only far too big and sophisticated to include in a basic C++ course, but even too big and diverse for one follow-on "Mastering the STL" course. We should introduce STL components not as ends in themselves but as we need them to convey algorithms and techniques.

Since arrays and other simple containers are essential to non-trivial C++ programs, our basic course must cover them thoroughly. We can't leave our students to struggle with C's primitive array handling, so we introduce a few straightforward array templates that are both more useful (in everyday applications) and simpler than their STL counterparts. We then explain what the STL is and point the interested student toward references.

### *String handling*

Nearly all C++ textbooks and nearly all C++ courses introduce a string class, if only to illustrate memory management (and to liberate the programmer from C's crude string-handling), but then they don't do much with it. Many programmers lack experience and confidence in routinely manipulating text, and we see the result in cumbersome, user-unfriendly programs.

I introduce string classes, too, but once the student sees how they work, we go on to examine their *use* in straightforward parsing and scanning functions. A two-session follow-up course is available for those who want to go deeper into this important topic.

## 7. Course emphasis and style

### *More than just how it works*

The worst fault a C++ course can have is to focus exclusively on the *rules*, but many courses do just that. They explain every detail of every language feature drawing little distinction between important and unimportant or between good and bad. Such courses may help the student pass a certification examination, but they won't help him or her to develop high-quality software. The value of any programming course lies in conveying a solid grasp of problem solving and *good programming practice*. Because of the vast range of choices in C++ that need is especially acute here.

I don't worry if my students haven't memorized every detail of C++ syntax and semantics; they can always look something up when they need it. If they understand the purpose of each feature, the relationships among features, and how to use C++ *well*, they will have gotten value from the course.

### *Having fun*

Despite the daunting difficulties, a good C++ course ought to be an enjoyable and satisfying experience. More than most other languages, C++ lets us exercise creative design over a wide range of levels.

I've noted a gratifying enthusiasm among my C++ students (at least those who attended regularly and did the assignments). It's *fun* to design and build classes and other object-oriented constructs and then see them exhibit the desired behavior. It's fun to debate the pros and cons of alternative approaches to some problem.

The instructor should share in this enthusiasm. Teaching C++ is an enjoyable experience. I've had fun with each C++ class I've conducted, both commercial and academic. I've enjoyed each group of students, and I maintain contact with a number of them in an informal C++ network.

Teaching C++ is also an educational experience. I learn something new about the subject matter each time. Given the pace of growth in the language, in related libraries, in add-on products, and in object-oriented methodology, I expect that to continue indefinitely.

---

[7] Run-time type identification.

[8] That's the *quasi object-oriented* approach taken in some popular high-level application development tools.