

Learning to Program: It Starts with Procedural

Conrad Weisert, June, 1997

Modern programming paradigms

A 1990's computer programmer confronts an intimidating range of programming paradigms, approaches to problem solving that are radically different from one another. Those paradigms include:

Paradigm	Exemplified by
Event-Driven	TurboVision [®]
Functional	Lisp
Logical	Prolog
Object-oriented	Smalltalk [®]
Procedural	Basic
Relational	SQL
Report	RPG
Spreadsheet	VisiCalc [®]
Visual	CA Realizer [®]

Some widely used programming languages combine two or three paradigms. C++, for example, is procedural, functional, and object-oriented, while *Visual Basic*[®] is event-driven, visual, and procedural.

Experts argue endlessly about the relative merits of the different paradigms for solving different classes of problem. Each has its strengths, and today's professional programmer needs to be aware of all of them and to have a solid command of two or three.

Learning and teaching programming

Similar arguments address teaching programming to beginners, but here the issue is clear cut: Students with no programming experience must start with *procedural* programming. The reasons are clear.

First, a specified sequence of steps is the *natural* approach to problem solving. Stored-program computers work that way internally. We follow manual procedures in our everyday lives. We express algorithms procedurally.

Second, procedural programming is embedded in the other paradigms. You can't write object-oriented, event-driven, or non-trivial functional programs without resorting to procedural code, often lots of it.

Third, a procedural program is well-suited to demonstrating and applying principles of good practice. Localized knowledge, cohesive modules, and clean interfaces have counterparts in other paradigms, but are easiest to grasp in a procedural program. Structured coding, relevant only to procedural programming, reinforces the student's appreciation of the value of source-code readability and maintainability.

Finally, a procedural program is entirely under the programmer's control. Students can be 100% responsible for their programs. They can neither blame some mysterious behind-the-scenes *engine* for their difficulties nor bluff a solution by trial-and-error flailing. Debugging strategy is straightforward.

So what's *wrong* with procedural programming?

The main drawback of procedural programming is that it breaks down when problems get very large. There are limits to the amount of detail one can cope with. Non-procedural programming can help the programmer compartmentalize and manage that detail.

Furthermore, some kinds of problem are most easily modeled by non-procedural techniques. A procedural solution may be natural, but a non-procedural one may be more direct. We can agree, then, that various forms of non-procedural programming are vastly more effective for many large real-world problems.

What about shortcuts?

If non-procedural programming offers such advantages, why can't beginning programmers start with it, picking up elements of procedural programming as they need them? Many courses attempt just that, e.g. "Visual Basic for Beginners". Results, however, keep confirming that this leads to shaky understanding. Students have only the vaguest understanding of what the computer is doing. They may memorize techniques that work for small common problems or for customizing a skeleton program, but they're helpless the first time they need to work out an *original* solution. After they finish the course, they go out into organizations and take unreasonably long to write ugly code that's unreliable and costly to maintain.

There is no shortcut. Programmers must master the procedural paradigm before undertaking the others, and that mastery much include a solid appreciation of established principles of good programming practice.

In an academic term that doesn't mean we devote an entire first course to procedural programming. I've had success conducting a 10-week first course¹ in which we study procedural programming (BASIC) in the first half and take up event-driven visual programming (Visual Basic) after the mid-term. When my students confront VB's bewildering array of controls,

¹ Syllabi are available from weisert@acm.com.

event-procedures, and properties-lists they already feel secure not only in their command of coding detail but also in their grasp of how everything fits together. They can then concentrate on what's *new* in event-driven visual programming.

I do the same thing in my non-academic courses. If a client asks me to give a Visual Basic course for beginners, I counter by proposing a sequence of two shorter courses¹, the first using BASIC². An interval of a month or so between the two courses is ideal, provided that the participants get assignments that maintain and reinforce the skills they learned in the first course.

That strategy applies to object-oriented programming, too. C++, in all its size and complexity, is utterly unsuited to learning beginning programming, but its C-like subset is a practical vehicle, especially for those likely to pursue serious full-time programming.

Which languages?

It really makes little difference which procedural language a programmer first sees, as long as the course focuses on mastering programming *concepts* rather than language details. At a minimum the chosen language must support:

- subroutines and functions with formal parameters,
- structured coding constructs,
- enough block structure to segregate local and global data.

Those requirements are met by Pascal, PL/I, C, and Ada, as well as by recent versions of BASIC and COBOL.

For students likely to go on with Visual Basic, the obvious choice is BASIC. For those likely to go on with C++ or Java, the choice is C. In those cases the first course can avoid those details of the procedural language rendered obsolete by the modern extensions, e.g. C's memory management and stream I-O functions or BASIC's type suffix and dot identifier separator.

These choices, however, aren't rigid. A programmer who has a thorough mastery of procedural programming concepts can make the switch to an unfamiliar procedural language very quickly. I'll teach C++ or VB to experienced programmers whose background is in any language that satisfies the above criteria.

What about machine language?

Although there's little excuse to program these days in a machine-dependent assembly language, showing students what happens in registers and memory can reinforce their grasp of the stored-program concept. If time permits, I spend the first couple of sessions in an academic course showing *machine-language* on a simple pseudo computer.

State of the art and a generation gap

The *structured revolution* of the 1970's was supposed to establish a systematic approach to developing high-quality software. Unfortunately, some of those lessons have been forgotten. Much of today's design and coding, even from respected vendors, is of appallingly poor quality. It's no wonder that many of today's mainstream software products are slow, full of bugs, and behind schedule.

A new generation of programmers has come on the scene since the structured revolution. Many of them are talented and creative and have a strong grasp of established principles. All too many others, however, have acquired little appreciation of either program quality or systematic development. They sit all day at their interactive graphic workbenches and churn out anything that eventually works. Some of them even assert that the principles of good practice established years ago are obsolete and don't apply to modern programming environments!

The challenge and the opportunity in education, both academic and commercial, is to instill in the student a deep appreciation of sound principles of high-quality design and coding. An introductory course in procedural programming gives us the best chance to get those principles across.

² The QBASIC[®] interpreter, included in some versions of MS-DOS, is well suited to such a course. Unlike its predecessors, it supports subroutines, functions, and structured coding, and its interpretive environment is congenial to neophytes' debugging sessions.