Structured COBOL

MIKE MURACH

Structured COBOL

Mike Murach

SRA

SCIENCE RESEARCH ASSOCIATES, INC.
Chicago, Palo Alto, Toronto, Henley-on-Thames, Sydney, Paris
A Subsidiary of IBM

Director/author:	Mike Murach
Technical advisor:	Paul Noll
Writer/editor:	Judy Taylor
Writer/programmer:	Doug Lowe
Book designer:	Michael Rogondino
Production coordinator:	Debbie Lowe
Illustrator:	Steve Ehlers
Acquisition editor:	Phil Gerould
Project editor:	Ron Lewton

We gratefully acknowledge the following for their permission to reprint the materials listed:

Courtesy of International Business Machines Corporation: figures 1-1, 1-7, 5-17, 10-2, 10-5, 10-7, 10-11, A-1, A-2, and A-3.

Courtesy of The Burroughs Corporation: figure 1-3.

Courtesy of Honeywell Information Systems: figure 1-4.

Courtesy of Control Data Corporation: figure 10-4.

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging in Publication Data

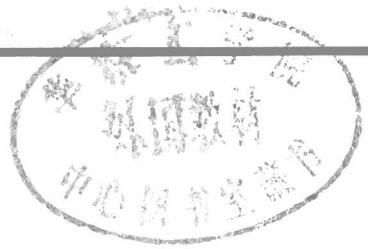
Murach, Mike.
Structured COBOL.

Includes index.
1. COBOL (Computer program language). 2. Structured programming. I. Title.
QA76.73.C25M863 001.6'424 79-22329
ISBN 0-574-21260-4

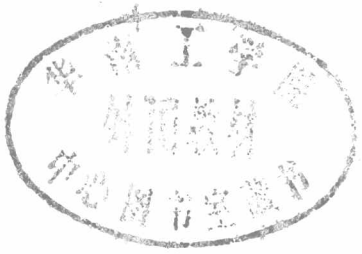
Structured COBOL

Acknowledgment

The following information is reprinted from *COBOL Edition 1965*, published by the Conference on Data Systems Languages (CODASYL) and printed by the U.S. Government Printing Office:



Contents



Preface for Instructors

This book is intended for a one- or two-semester (or quarter) course in COBOL. It presents standard COBOL as described in the 1968 and 1974 standards of the American National Standards Institute. It also presents the most current techniques of structured program development.

As I'm sure you know, dozens of COBOL books are available for college and industrial courses. Nevertheless, COBOL instruction is a continual problem because training requirements for the COBOL programmer are continually changing. As a result, instructional materials that were acceptable three years ago are no longer acceptable. To better understand the instructional problem (and the need for this book), let me briefly describe the changing nature of the COBOL environment.

The development of COBOL

COBOL is one of the oldest programming languages, first introduced in 1959. By the mid-sixties, COBOL had established itself as the most widely used language for business applications. Before 1968, however, there was no standard COBOL language. Although all versions of COBOL were based on the same set of language specifications, there were major COBOL differences as you moved from one type of computer to another.

Then, in 1968, the American National Standards Institute (ANSI) approved a standard COBOL language. Theoretically, this meant that one standard language could be used on all types of computer systems. In practice, however, the standards didn't provide for all of the capabilities that computer users wanted. As a result, each manufacturer added *extensions* to the language that provided the additional capabilities. Furthermore, the 1968 standards allowed these extensions. As long as the rest of the language conformed to the standards, manufacturers could refer to their versions of COBOL as standard no matter how many extensions were added. Needless to say, the end result was that the 1968 COBOL standards did *not* lead to widespread standardization of the COBOL language.

In 1974, the American National Standards Institute released a new set of COBOL standards. These standards deleted some of the 1968 standards, modified others, and, most important, added most of the capabilities that

computer users had wanted in the 1968 standards. COBOL that is based on the new standards is usually referred to as 1974 ANS COBOL, while 1968 ANS COBOL refers to COBOL that is based on the old standards. In contrast to 1968 ANS COBOL, 1974 ANS COBOL for one computer system is usually very similar to that for another computer system. In the best cases, you can convert a 1974 ANS COBOL program from one system to another by making only minor changes to the program. On the other hand, extensions to the standards are still allowed. So in the worst cases, extensive changes must be made to an ANS 74 COBOL program to convert it from one system to another.

Because it takes time to develop new compilers, a few years usually pass between the time a new set of standards is released and the time compilers that conform to those standards are available. For instance, IBM didn't release a 1974 ANS compiler for its most popular computer, the System/370, until 1977. And at this writing IBM still hasn't announced a 1974 ANS COBOL compiler for its Disk Operating System, which is widely used on small System/360s and System/370s.

After a new COBOL compiler becomes available, it usually takes another few years before computer users convert to it. Why? Because it costs a lot of money to make the conversion. Programmers have to be taught how to use the new language. And eventually all programs written in the earlier version of the language have to be converted to the new language so they will compile correctly when they are modified to meet changing business conditions. In general, then, computer users delay the conversion to the new standards until the new compiler gives them some capability that they want but can't get through extensions on their current compiler. Because IBM extensions to the 1968 standards gave the System/360 and System/370 user all of the significant capabilities provided by the 1974 standards, there is little reason for System/360-370 users to want to convert to the new ANS 74 compiler. As a result, most COBOL programs for the System/360-370 are still being written in 1968 COBOL. Yet already we're getting rumors about new standards, perhaps becoming official in 1984.

From an instructor's point of view, this continual change presents many problems. In general, a college course should present 1968 COBOL because that's the language that a student is most likely to encounter when entering the business world. On the other hand, most businesses will eventually switch to 1974 COBOL. So a student should also become familiar with this version of COBOL.

To complicate the instructor's problem, new programming techniques are gaining wide acceptance in industry. In particular, it has become apparent that the techniques of structured programming are replacing the traditional techniques of program development (including the technique of flowcharting). In general, this forces the COBOL instructor to include techniques such as structured design and structured coding within the COBOL course. And this by itself has destroyed the usefulness of most of the traditional instructional materials for COBOL courses.

What this book does

This book is designed to teach a novice how to develop structured programs in COBOL. In terms of COBOL, the only major content omissions are the report writer and the segmentation modules. However, research has shown that these modules are used on only a small minority of the COBOL systems in use today. As a result, we believe that a student who completes this course will have the qualifications of an entry-level programmer in industry...and probably much more.

To handle the problem of what version of COBOL to teach, 1968 or 1974, this book teaches a subset of the two sets of standards that will run on either a 1968 or 1974 compiler. Whenever language is presented that is only available under one set of standards, it is clearly identified so the presentation of both sets of standards shouldn't be confusing to the students.

Because we feel that it's impossible to teach students how to code structured programs without also teaching them how to design structured programs, this book gives extensive coverage to modern design techniques. In fact, structured design and documentation are covered in detail in chapter 3, right after the introductory subset of COBOL is presented. Thereafter, students should be able to design and code simple programs using structured techniques. In my opinion, this material on structured design is essential to a structured COBOL course even though you won't find this material in most competing books.

When you read chapter 2, you may be surprised to discover that this book teaches structured COBOL and *only* structured COBOL. As a result, you will find nested IFs and GOTO-less COBOL right from the start. Although some people don't think it's possible to teach a course this way, we've been doing it for a couple of years. So remember that a student doesn't know that it's supposed to be difficult to code a nested IF. In fact, I think you'll find that it's easier for a student to code a nested IF than it is for him to code a program on an unstructured basis. In other words, give this approach a chance; I think you'll be delighted by the results.

How this book was developed

Because this book wasn't developed in the traditional way, I think you might be interested in how it was developed. To begin with, a major portion of the content in this book is taken from two earlier books: (1) *Standard COBOL*, Mike Murach, Science Research Associates, 1975, and (2) *Structured Programming for the COBOL Programmer*, Paul Noll, Mike Murach & Associates, 1977.

Beyond this, a team of people worked on the development of this book. I was responsible for the organization of the book and the coordination of the team; I also did some of the writing. Paul Noll, a software specialist and independent consultant in industry, was the technical advisor. Judy Taylor did most of the rewriting and new writing that was required...a major effort. And Doug Lowe did some of the writing; he also wrote all of the programs that are used in this book based on the programming standards supplied by Paul Noll.

Because of this team approach, I think this book has some strengths that aren't found in competing products. First, because the educational approach used in this book is adapted from *Standard COBOL*, I'm confident that the book will be effective in terms of instruction. *Standard COBOL* has been used by thousands of students in more than 200 colleges and universities in the last few years. It has also been used in dozens of businesses for inhouse training; and it has been used by thousands of professionals for self-instruction. As a result, the method of instruction used in this book has been proven effective many times over.

Second, because the structured programming content is taken from Paul's book, *Structured Programming for the COBOL Programmer*, you can be sure that the techniques presented in this book are effective. Paul's structured programming book is currently is use in hundreds of businesses for in-house training. And many companies have adopted its principles as their standard for program development. In short, though there is considerable

debate about which are the best methods for implementing structured programming, the methods presented here are widely accepted.

Finally, because Paul was responsible for the technical excellence of this product, I'm confident that it is without equal in this respect. Paul has been responsible for the training of 150 programmers at Pacific Telephone in San Francisco; he was once the assistant manager of the COBOL group within GUIDE (an association of large IBM users); and in my opinion, he is one of the top COBOL experts in the country. To the benefit of this book, Paul has taken pains to see that the COBOL presented here is not only accurate, but that it also represents the practices that are currently in use in the best COBOL shops in America. As a result, if you compare the programs in this book with those taken from any competing book or course, I think you'll find a significant difference in program quality.

Some features of the book

Modular organization An important feature of this book is its organization, sometimes referred to as *modular* organization. In particular, the book is organized as indicated by table 1. This means that you can continue with any other part (or module) in the book after you cover the first two parts. If, for example, you want to cover file handling before table handling, you can skip to chapter 10 immediately after chapter 5. Note also that the design of part III is random. This means that the chapters in this part don't have to be studied in sequence. In short, your course can be teacher-directed or student-directed, but it will not be textbook-directed.

In addition to the teaching flexibility, there is an important educational reason for modular organization: it forces an author to present the essence, or *core content*, of the subject in just a few chapters early in the book. This in turn means that early in the course the student is shown the important relationships between the elements of the subject. Because one of the major problems of learning is the failure to see these relationships, the emphasis on core content makes learning more effective.

I might add that although many books are advertised as modular, few actually are. In most cases, a few alternative paths through the books are given to support the claim of modularity, but these seem to be afterthoughts rather than an integral part of the book's design. To be truly modular, I think the essence of the subject must be presented early in the course; and all subsequent modules must require only this core content as a prerequisite.

Educational approach In general, there are two basic approaches to teaching the COBOL language. The first teaches the COBOL elements separately until a great deal of detail has been covered; at that point, a few of these elements are combined in a complete program. A book like this is easy to identify since each early chapter covers a COBOL division: Identification, Environment, Data, and Procedure. Using this approach, the first complete program commonly isn't presented until well into the book.

Table 1 The organization of the book

Part	Chapters	Title	Prerequisite Parts	Design
I	1	Required Background	—	Sequential
II	2-5	COBOL: The Core Content	I	Sequential
III	6-9	Advanced COBOL Subjects	I, II	Random
IV	10-13	File Handling in COBOL	I, II	Sequential

The problem with this parts-to-the-whole method is twofold. First, a student doesn't have the perspective to appreciate the relationships between the parts until becoming familiar with a complete program. As a result, the student learns the parts through memory rather than through an underlying structure or concept. Second, this method is impractical in classroom teaching. Normally, it takes several weeks before a student has learned enough of the parts to be able to write a complete program. In the meantime, motivation dwindles, and what should be an exciting problem-solving class becomes a frustrating struggle to learn the massive amount of detail associated with the language. If the assignment of computer laboratory time is a fixed number of hours per week, the instructor usually must create supplementary material so students can run programs, or segments of programs, in the early weeks of the course.

The second approach to the teaching of COBOL is used in this book. After some background material in chapter 1 (some of which may be review), topic 2 of chapter 2 presents a complete program including card input, printer output, data movement, editing, arithmetic, and logic. As soon as this basic program is understood, students can begin to write significant programs of their own. Before chapter 2 is completed, though, one refinement of this first program and one additional program are presented, so three complete programs are shown in chapter 2. By this time, a full subset of the language has been presented, and a student is ready for independent work in a computer lab.

After the COBOL subset of chapter 2, chapter 3 shows a student how to design and document a program on a structured basis as preparation for structured coding. As I mentioned earlier, you can't teach structured COBOL without teaching structured design, but most competing books don't teach design. When students complete this chapter, they are ready to attack a wide range of programming problems.

To complete the core content of part II, chapter 4 provides a definitive presentation on correcting diagnostics, preparing test data, and debugging programs. Since these skills are essential to COBOL programming and to successful lab work, it is surprising that they are treated so lightly in most other texts. Then, chapter 5 presents a collection of elements and techniques that makes the book truly modular.

Once students have completed part II—in particular, chapters 2, 3, and 4—the major part of your job is done. When students understand the structure of the language and the related skills for design, documentation, testing, and debugging, it becomes easy for them to learn other COBOL elements and techniques. They become part of a pattern. If students can see how an element or technique relates to the whole task of programming, mastering the material is a manageable task.

Apparatus by topic Because learning depends on what a student does, not on what he or she sees or hears, each topic is followed by terminology lists, behavioral objectives, and, whenever relevant, problems and solutions. After reading a topic, the student should scan the terminology list to check comprehension. If the words are understood, the student can proceed. If there is not a clear understanding, applicable sections should be reread or the term noted so its meaning can be questioned in class. In any case, the list is for use as a quick review; the student shouldn't be expected to actually define the words.

The behavioral objectives describe the activities a student should be able to do upon completion of a topic. Since this book deals with programming, the primary objectives have to do with solving various types of programming problems using COBOL. In addition, there are objectives dealing with related skills such as developing a structure chart, using pseudocode, and describing a type of file organization. The intent of the objectives is to give the student a clear picture of his or her learning goals. Although some students will ignore the objectives, others will be more efficient learners because of them.

Although many instructors feel that preparing and using objectives is busywork, I prefer to see them in all textbooks. At the least, preparing objectives forces this author to focus more clearly on what he is trying to accomplish. Otherwise, he all too often concerns himself with writing a definitive work rather than concentrating on the goals of education.

At any rate, I believe objectives can contribute heavily to the success of a course. If students are convinced that the objective lists describe *all* activities they will be expected to perform, their learning will become much more directed. In every class I've taught, I have found students who wouldn't rest until they felt they could satisfy all the course objectives. On the other hand, there are some who won't believe you are telling them all that will be required, so it is important to refer to the objectives as you review a topic in class. If the objectives are prominent in all classroom activity, I am convinced that teaching has a great likelihood of success.

Since it is unlikely that two people will agree on a list of objectives for a course, you will probably want to modify those given. The objective lists, then, are only a starting point. However, if only those given are fulfilled, I would say that you have taught a highly successful course.

When the objectives deal with problem solving, they are followed by problems and solutions that provide practice in the skills described by the objectives. As much as possible, these problems are designed to show how the elements and techniques presented in the text are used in a different context. There are no multiple-choice, true/false, matching, or fill-in questions, because these have nothing to do with the important objectives of a programming course.

So there is immediacy to the problem-solving activity, solutions are given after problems. This has the advantage of letting students know when they are right, and just as important, letting them realize when they are wrong. For those students who wouldn't otherwise know how to begin solving a problem, the solutions are an essential part of the learning process. Although compiling and testing programs on a computer system has the same effect as doing the problems and checking the solutions, studying the solutions of a professional can correct many false notions and bad habits before problems are actually tried on a computer system. The expense of computer time makes this a practical consideration.

What about students who don't actually do the problems but look immediately to the solutions? The experience is still valuable. Although the best way to learn is to do the problems and compare the answers with the solutions provided, it may not be the most efficient way—particularly for the brightest students or for those with extensive experience in another language. In the interest of expediency, then, a student may read a problem, conceive a solution, and compare it with the one given. The important thing is that COBOL be viewed in the context of its application. Looking at the problems in this way, they can be seen simply as a means of presenting additional COBOL applications.

Lab problems Appendix C presents a progression of programming problems. Since they include test data listings, they are ideal for lab, although they can also be used for classroom exercises or tests. If a student can write and debug programs for all types of problems given, I feel sure that he or she is well qualified to become an entry-level programmer in industry.

DOS and OS reference manuals Because this book is intended to teach American National Standard COBOL, it does *not* give the detailed specifications required by any actual compiler. From a practical point of view, then, you will have to present some of the details demanded by your system. At the least, you will have to show your students how to use your system's job-control language to compile and test programs. In addition, you will probably have to show them how to create program and system names that are acceptable for your system.

If you are using a System/360 or 370 running under DOS or OS, two reference manuals are available with this text: one for DOS users, one for OS users. These manuals present all of the reference material your students will need for running programs on your system. They are designed specifically for use with this textbook, and they include material on creating system names, COMP-3 usage, job-control language, and so on. If you use these manuals, you won't need to supplement this book with IBM manuals, although it won't hurt to keep a set available for lab sessions.

Instructor's guide An Instructor's Guide is available with the text. Among other things, it contains pretests for chapters that might have been covered by a prerequisite course, posttests for critical chapters, test answers, solutions for selected lab problems, and masters for preparing overhead projector foils. Be sure to obtain a copy of the Guide because it can save you many hours of preparation time.

Conclusion

In this book, we have tried to do something that I don't think has been done effectively before in a college text. That is, we have tried to integrate the teaching of structured design and coding with the teaching of COBOL. Although we have used this approach with programmer trainees for a couple of years now, I think this is the first book that uses this approach at the college level. If you try this approach, I think you'll be delighted by the results.

In addition, I hope that this text will help to provide some new solutions for old teaching problems. In particular, I'm thinking about the range of aptitudes you often encounter in a programming course. For instance, some students will grasp the material by reading the text only, some will require minor assistance in addition to the text, some will require extensive help, and some just shouldn't be taking a programming course.

Because this text gets into the problem-solving aspects of COBOL in chapters 2 and 3, the aptitudes of your students should be apparent to you early in the course, in time for effective counseling. Then, the brightest students can be assigned independent work because the educational approach used in this book has already proven itself to be effective for independent study. In the meantime, the marginal students can be given full assistance and supervision. If there's such a thing as a class that has no marginal students, I believe the entire class can be taught through independent study with minor assistance and supervision from the instructor.

Each time we develop a new product, we try to improve the technical content as well as the educational effectiveness. And we will keep trying to improve. That's why I welcome your comments, criticisms, suggestions, or questions. Please write if you have anything you want to say about this book.

Mike Murach
Mike Murach & Associates, Inc.
4222 W. Alamos, Suite 101
Fresno, CA 93711

Introduction for Students

COBOL, which stands for *CO*mmon *B*usiness *O*riented *L*anguage, is the most widely used programming language for business applications. It can be used on most business computers, so COBOL programmers can use their skills on many different makes of computers. At present, there are well over 100,000 COBOL programmers working in industry. And each year, thousands of new COBOL programmers are trained. If there is one programming language the business student should become familiar with, it is COBOL.

Today, there are two basic versions of COBOL. One is based on standards provided by the American National Standards Institute (ANSI) in 1968. COBOL that conforms to these standards is called 1968 ANS COBOL, or just 1968 COBOL. The second version of COBOL is based on a revised set of ANSI standards that were released in 1974. COBOL that conforms to these standards is called 1974 ANS COBOL, or just 1974 COBOL. With few exceptions, the COBOL that is used today is based on either 1968 or 1974 ANSI standards.

This book teaches you how to develop COBOL programs in either 1968 or 1974 COBOL. When you finish this course, then, you will have the skills necessary for programming any computer system that uses COBOL. From a practical point of view, however, each computer system is likely to have some non-standard peculiarities. So you will usually have to learn these variations as you move from one computer system to another.

Before you begin to use this book, there are several things you ought to know about it:

1. This book is designed so the chapters don't have to be read in sequence. In brief, after you complete the first five chapters, you can skip to any of the other parts of the book. And you can skip to any chapter in part III without reading the chapters that precede it in that part. So don't worry if your instructor assigns chapters in an unusual way; the book is designed to be used that way.

If you are studying this book on your own, I recommend reading it in the standard book sequence, from chapter 1 to chapter 13. But don't feel

that you should rigidly adhere to this sequence. Whenever your interest in a subject is aroused, read the appropriate chapter. There is no greater assurance that learning will take place than to study a subject in search of an answer.

2. At the end of each topic or chapter are lists consisting of the new terms encountered. The intent is not that you be able to define these words but that you feel you understand them. After you read a topic, glance at the list and note any word whose meaning is unclear to you. Then reread the related material. Once the terms are fixed in your mind, continue on.

3. Following the terminology lists for each topic or chapter are one or more behavioral objectives. They describe the activities (behavior) that you should be able to perform upon completion of a topic or chapter. The theory is that you will be a more effective learner if you know in advance what you are expected to do and what you will be tested on. This contrasts the traditional course in which the student is forced to guess what he will be tested on.

In general, behavioral objectives can be classified as (1) *knowledge objectives* and (2) *application objectives*. A knowledge objective requires you to list, identify, describe, or explain aspects of a subject. For example, the first objective in chapter 1 is to be able to list the components of a typical computer system. Once you are told or have read what these components are, you should have no trouble fulfilling this objective. Although other knowledge objectives will be more involved and more difficult than this one, given the objective and a source of knowledge, you should be able to perform the activity described.

Since COBOL programming is concerned entirely with problem solving, the primary objectives of this book are application objectives—those that require you to apply knowledge to problems. In general, knowledge objectives are given only when they are a prerequisite for applying some aspect of COBOL. If only one objective were given for this entire book, it would be something like this: Given a business programming problem, solve it in COBOL.

4. Following the behavioral objectives are one or more problems for each application objective. These are intended to get you involved. There is much truth in the maxim: I hear and I forget; I see and I remember; I *do* and I understand. If there is one message coming from research in education, it is that meaningful learning depends on what the learner does—not on what is seen, heard, or read.

Because the intent of this book is to teach COBOL programming, the problems, for the most part, ask you to apply COBOL to significant programming tasks. There are no fill-in answers, no multiple-choice questions, and no true/false statements because these types of activity have nothing to do with writing COBOL programs. As much as possible, the problems are intended to stimulate the kind of thinking that would be necessary if you were actually performing the job of a programmer. Because the problems often require you to apply COBOL to situations that go beyond the applications presented in the topics themselves, I hope that at times you will experience the joy of discovery and receive the reward of deeper understanding.

Solutions are presented immediately after the problems. This lets you confirm that you are right when you are right, but it also lets you learn from being wrong. By checking the solution when you finish a problem, you can discover when you are wrong and correct false notions before they become habits.